Provenance-Aware Dynamic Analysis of JavaScript

Laurent Christophe

Promotors:

Prof. Dr. Coen De Roover

Prof. Dr. Wolfgang De Meuter

Dissertation submitted in fulfilment of the requirements for the degree of Doctor of Sciences

January 2026

President of the Jury:

Prof. Dr. Ann Nowé – Vrije Universiteit Brussel

Members of the Jury:

Prof. Dr. Michael Pradel – University of Stuttgart

Prof. Dr. Tom Van Cutsem – KU Leuven

Prof. Dr. Elisa Gonzalez-Boix – Vrije Universiteit Brussel

Prof. Dr. Jens Nicolay – Vrije Universiteit Brussel

Vrije Universiteit Brussel

Faculty of Sciences and Bioengineering

Department of Computer Science

Software Languages Lab

ISBN: 978-9-4934-6143-7

NUR: 989

THEMA: UMC

Acknowledgments:
The work of this dissertation has been funded in part by the SBO project Tearless by the Agency for Innovation and Entrepreneurship (VLAIO).

Printed by:
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel: +32 2 629 33 44
crazycopy@vub.be
www.crazycopy.be

Abstract

Dynamic program analysis, which examines a program's behavior during execution, is essential in domains such as security and software maintenance. This dissertation focuses on provenance-aware analysis, such as dynamic taint analysis, which requires tracking how data propagates during execution. To implement analyses independently from the execution environment, a common approach is to instrument the source code of the target program. However, developing robust instrumentation-based analysis tools is hindered by three key challenges in most managed languages: the syntactic complexity of the language complicates instrumentation; optimizations in the representation of immutable values make provenance tracking difficult; and the distributed nature of modern applications requires analysis orchestration.

This dissertation presents a novel modular approach to overcome these challenges for JavaScript. To simplify instrumentation, we transpile programs into a core variant. Compared to JavaScript, this core variant employs simpler constructs, reduces the number of syntactic node types by 57%, and still passes 99.8% of the official JavaScript conformance test suite. To track immutable values, we introduce a value-centric approach that promotes primitive values to transparent reference values, which requires an access-control system based on reflection. Owing to JavaScript's reflection API, this method yields high precision and transparency, albeit with a performance cost. Finally, to coordinate the analysis of distributed applications, we centralize the analysis logic in a dedicated process. This architecture provides high expressiveness and convenience for the analysis developer, but introduces additional communication overhead.

This research advances the state of the art by delivering a modular framework for building robust instrumentation-based tools for provenance-aware analyses in JavaScript.

Samenvatting

Dynamische programma-analyse, die het gedrag van een programma tijdens uitvoering onderzoekt, is essentieel in domeinen zoals beveiliging en softwareonderhoud. Dit proefschrift richt zich op herkomstbewuste analyse, zoals dynamische taint-analyse, waarbij het noodzakelijk is om te volgen hoe gegevens zich tijdens uitvoering verspreiden. Om analyses onafhankelijk van de uitvoeringsomgeving uit te voeren, is een gangbare aanpak het instrumenteren van de broncode van het doelprogramma. Het ontwikkelen van robuuste instrumentatiegebaseerde analysetools wordt echter belemmerd door drie belangrijke uitdagingen in de meeste beheerde talen: de syntactische complexiteit van de taal bemoeilijkt instrumentatie; optimalisaties in de representatie van onveranderlijke waarden maken het volgen van herkomst moeilijk; en de gedistribueerde aard van moderne toepassingen vereist coördinatie van de analyse.

Dit proefschrift presenteert een nieuw modulair kader om deze uitdagingen te overwinnen voor JavaScript. Om instrumentatie te vereenvoudigen, transpilen we programma's naar een kernvariant. In vergelijking met JavaScript gebruikt deze kernvariant eenvoudigere constructies, vermindert het aantal syntactische knooppunttypes met 57% en slaagt nog steeds voor 99,8% van de officiële JavaScript-conformiteitstest. Om onveranderlijke waarden te volgen, introduceren we een waardegerichte aanpak die primitieve waarden promoot tot transparante referentiewaarden, waarvoor een toegangscontrolesysteem gebaseerd op reflectie nodig is. Vanwege de reflection-API van JavaScript levert deze methode hoge precisie en transparantie op, zij het met een prestatie-overhead. Ten slotte, om de analyse van gedistribueerde toepassingen te coördineren, centraliseren we de analyselogica in een toegewezen proces. Deze architectuur biedt hoge expressiviteit en gemak voor de analyseontwikkelaar, maar brengt extra communicatie-overhead met zich mee.

Dit onderzoek bevordert de stand van de techniek door een modulair kader te bieden voor het bouwen van robuuste instrumentatiegebaseerde tools voor herkomstbewuste analyses in JavaScript.

Acknowledgments

This was a long time coming, and I have a long list of people to acknowledge, so I had better get going.

Thank you, Coen and Wolf, for your unwavering support. You both deserve medals for this. Although probably Coen even more so, sorry for the grey hairs. Wolf, you are a great teacher; I just wanted to put that out there. You inspired me and got me started on this journey.

Thanks to the jury members (Michael, Tom, Ann, Elisa, and Jens) for reading through my text. I am truly grateful for the time you have spent engaging with this long and sometimes dense dissertation. Special thanks go to the external members: Michael, for his thoughtful criticisms, and Tom, for his in-depth analysis of my work. It is humbling to have such prolific researchers seriously consider one's work.

Thank you, Laura, for your patience and support throughout all these years. Thank you for taking on more than your share to make this family as functional as one can hope. Thank you for taking care of me when I needed it the most.

Thank you, my three entertaining children, for teaching me the value of time. No joke, I am not sure I would have finished my PhD without you. Do enjoy life and never stop having fun.

Thank you, Mom, for your continuous support, and for never pressuring me to finish this. In my book, even if the mean is lower, raising a child alone is harder than raising three children as a couple. Rest assured, you did a great job.

Thank you, Nane, for making me swear on your deathbed that I would finish this. It is only now that I truly understand what you did for us. I hope I will one day have the opportunity to read Jules Verne to my grandchildren.

Thank you, Grandpa, for leading by example. You once told me that life deals you cards, and you have to play them to the best of your ability. This kind of stoic advice is easy to dispense but harder to live by. You actually did that.

Thank you, Cathy, Patrick, Claire, and Mathieu, for being so well-intentioned. You have always been a positive force in my life.

Thank you, Martine and André, for your encouragement and your gentle support. And thank you for having raised four wonderful people.

Thank you, Karim, Jean, and Mauricio for your tough love.

Thank you, my friends; I'm truly lucky to have met the bunch of you and especially you, J.R.

Thank you to all the softies. In no particular order: Dries, for advising me (your sharp mind was hopefully not lost on me); Theo and Viviane, for everything you have done for the lab; Maarten, for the fruitful collaboration and for accepting the dubious honor of being my GitHub heir; Scull, for taking a chance with my framework; Christophe, for supporting my application (it actually means a lot to me) and you fully deserve that Christmas tree; Aaron, Florean, and Tim, just for being cool dudes; and the swimming team (Nathalie, Eline, Laure, Reinout and Kevin) for all the fun times. And thanks to the rest!

Thank you to my new colleagues at Sirris for their interest and support; shoutout to Johan, Nicolas, Nick, Sreeraj, Mike, and Isabelle.

Thank you all!

Contents

List of Figures

List of Tables

List of Listings

Glossary

- Public (resp. private) (execution) information: Information that can (resp. cannot) be directly derived from observing the run-time behavior of the program (p. 31).
- Extrinsic (resp. intrinsic) (execution) information: Information that can (resp. cannot) be directly derived via introspection during execution (p. 31).
- Point-based (program) instrumentation: Program instrumentation that performs measurements at specific execution points via introspection (p. 34).
- Hook-based (program) instrumentation: Program instrumentation that performs measurements via global notification mechanisms (p. 34).
- (Analysis) discrepancies: Adverse situations in which the program under analysis does not behave as during baseline execution (p. 32).
- (Analysis) semantic overhead: The discrepancies directly introduced by an analysis as it modifies the semantics of the program (p. 33).
- (Analysis) performance overhead: The additional memory and time complexity introduced by the analysis, which may lead to performance-related discrepancies (p. 33).
- Value: Representation of data in managed languages involving typing and memory indirection to support mutations (p. 36).
- Value identity: The distinction between values as established by the strictest comparison operator provided by the managed language (p. 37).
- (Data) inlining: Performance optimization involving copying data items into locations in the state where a value is expected (p. 44).
- (Value) interning: Performance optimization involving caching immutable values for reuse (p. 48).
- Provenancial equality (coined term): Value identity in a naive allocation system which does not conduct data inlining nor value interning (p. 50).

Chapter 1

Introduction

We introduce this dissertation by outlining its scope and structure, providing the reader with a clear overview of the problems addressed, the approach taken, and the contributions made. This chapter is organized as follows:

- *Section 1.1 articulates the problem statement.*
- *Section 1.2 provides an overview of our approach.*
- *Section 1.3 outlines the main contributions.*
- *Section 1.4 lists supporting publications.*
- *Section 1.5 summarizes the remainder of the dissertation.*

1.1 Problem statement

Program analysis is the field that studies the behavior of programs. Traditionally, program analysis techniques are divided into two branches: static analysis and dynamic analysis. Static analysis sacrifices precision by reducing the state space through abstraction, whereas dynamic analysis maintains precision but focuses on a single execution path. In this dissertation, we focus on dynamic analysis, specifically on techniques that require information about the provenance of run-time values. Important examples of such techniques include dynamic taint analysis [84, 25], which detects violations of security policies, and concolic testing, which generates test inputs to maximize code coverage [103, 44].

In this dissertation, we focus on realizing dynamic analysis through source code instrumentation, which offers the advantages of being independent of the execution environment and providing a view of program execution at the source level. Moreover, this dissertation focuses on managed languages (such as Smalltalk, JavaScript, and Python), which execute on virtual machines that manage memory access and thereby ensure memory safety.

Implementing provenance-aware dynamic program analysis for managed languages via source code instrumentation is complex for three main reasons:

- Mainstream managed languages are complex and feature syntactic constructs that are difficult to reason about—e.g., class definitions, iterator-related constructs, destructuring assignments, variable hoisting, and list comprehensions.
- Most managed languages optimize the handling of immutable values using techniques such as inlining and interning. These optimizations hinder direct monitoring from providing precise information about the provenance of immutable values.
- Managed languages have become a prime choice for developing distributed applications. For example, JavaScript is now present on both the server and client side of web applications, and microservices are frequently built using Python frameworks such as FastAPI or Flask. Reasoning about

the shared state of these applications requires an analysis that is itself distributed and must be orchestrated. This added difficulty compounds with the two challenges mentioned above that apply to single-process applications.

The primary objective of this dissertation is to simplify the implementation of provenance-aware analyses for managed languages by proposing a modular approach that separates four key concerns:

- Analysis-specific logic, such as taint propagation for dynamic taint analysis and the collection of symbolic constraints for concolic testing.
- Managing incidental complexity arising from the presence of non-essential syntactic constructs in the target language.
- Providing provenance for run-time values, particularly immutable values.
- Orchestrating the various components of the analysis to reason about distributed applications as a whole.

We illustrate this approach using JavaScript; the feasibility of applying it to other managed languages remains an open question, which we briefly discuss throughout the dissertation.

1.2 Approach

As stated before, our approach to dynamic program analysis is based on source code instrumentation. While dynamic analysis tools implemented in this manner can potentially be independent of the execution environment, they must contend with the non-essential syntactic constructs of the target language. To address this incidental complexity, we propose transpiling the target program into a *core variant* of the language and conducting analysis on this simplified representation before transpiling it back to an executable form.

While ad-hoc instrumentation of the core variant representation of the target program is feasible, we also provide an *aspect-oriented* [60, 37] API that integrates code transformation into our approach. In aspect-oriented programming, cross-cutting concerns are encapsulated in aspects, which specify how their behavior should be integrated into the base program through a join-point model. Therefore, in our approach, an analysis is expressed as an advice. By operating on a core variant, our join-point model supports specifying advanced dynamic analyses such as shadow execution while keeping the complexity of the analysis manageable.

To enhance the precision of tracking the provenance of run-time values, we propose to *promote* immutable values to handle references. This approach effectively reverses optimizations such as value inlining, which copies the data of immutable values directly into the scope and the value stack, as well as interning, which maintains a global registry of immutable values to reduce memory consumption and enhance comparison efficiency.

To maintain semantic transparency, access to handle references must be carefully mediated; for instance, adding two objects does not have the same semantics as adding two numbers. We address this by adapting an existing security pattern known as a *membrane*, which uses value virtualization to transitively enforce an interposition layer between two regions: one region where handles are allowed to enhance the precision of provenance tracking, and one where handles are forbidden to preserve the semantic transparency of the analysis.

To further decouple analysis-specific concerns from provenance tracking, we compose the analysis-specific advice with a generic provenance-tracking advice into two largely independent layers. From the perspective of the analysis advice, the language behaves as if immutable values, such as numbers or booleans, can be differentiated based on their provenance.

To evaluate the precision of provenance tracking independently of consuming analyses, we introduce a novel *metric* that assigns each run-time value a score reflecting how much information is known about its origin.

To orchestrate the various components of an analysis targeting distributed applications, we propose *centralizing* the analysis-specific logic within a separate process. Implementing this vision requires intensive communication between the central analysis process and the processes of the application under analysis. This communication must occur in a synchronous yet non-blocking manner, which necessitated the development of a novel protocol.

1.3 Main contributions

Below, we present the main contributions of this dissertation, which span the fields of program analysis, language design, aspect-oriented programming, and communication protocols.

Core variant – Chapter 3 The idea of simplifying reasoning about a programming language by transpiling it into a core variant is not novel. However, existing approaches for JavaScript, such as λ_{JS} [46], focus on pre-Harmony JavaScript (i.e., ECMAScript 5.1), while our framework supports ECMAScript 2025. To the best of our knowledge, our work is the first to support post-Harmony JavaScript, which is significant given that the language has evolved considerably since the Harmony era.

Comprehensive join point model – Chapter 4 Aspect-oriented programming has long been leveraged to carry out dynamic analysis with approaches such as Racer [10], which is based on AspectJ, and DiSL [75], which conducts binary code instrumentation. However, as far as we are aware, our work is the first to introduce a join point model for a high-level programming language that is comprehensive enough to support full-fledged shadow execution.

Value-centric shadow values – Chapter 5 Metadata attached to run-time values is commonly referred to as *shadow values*. Traditionally, these have been implemented in a location-centric manner using a technique called *shadow execution* [72, 83, 138, 14, 105], which involves mirroring parts of the run-time state. In contrast, our approach is value-centric and relies on a combination of promoting immutable values and virtualizing compound values. The advantage of our method over shadow execution is that it does not encounter synchronization issues when transferring control to unknown built-in functions or uninstrumented code regions.

Advice layering – Chapter 5 The closest related work is found in the field of aspect-oriented programming (AOP), specifically through execution levels [118, 119], which allow aspects to advise other aspects while controlling cycles and avoiding infinite recursion. The primary distinction between execution levels and advice layering is that execution levels can be changed dynamically, which requires a runtime, whereas our approach is static and relies exclusively on code transformation. We are not aware of prior work that achieves multi-layered analysis exclusively through instrumentation.

Provenance metric – Chapter 5 Separating concerns related to tracking provenance from those related to analysis-specific logic is a central theme of this dissertation. To achieve this vision, it is essential to evaluate the precision of a provenance provider independently from consuming analyses. We believe that our provenance metric is a pioneering effort in this direction.

Central analysis of distributed applications – Chapter 7 To the best of our knowledge, the idea of centralizing all analysis-specific logic into a separate process has not been previously explored. In this dissertation, we provide a preliminary assessment of the feasibility of this idea based on remote references. Our preliminary experiment shows that the approach is highly expressive and capable of supporting shadow values; however, abundant synchronous communication incurs performance overhead.

Synchronous non-blocking remote procedure calls – Chapter 7 We propose a novel protocol for event-driven programs to synchronously invoke remote procedures in a non-blocking manner. The main feature of our protocol is to pause the processing of the current event while remaining responsive to remote procedure calls, thereby preventing deadlocks. We present a JavaScript implementation of our protocol and model it using pushdown automata.

Technical contribution The engineering effort required to develop dynamic program analysis tools capable of handling real-world programs should not be underestimated. While implementing approaches into artifacts is common in computer science, our implementation has moved beyond the prototype stage and has been extensively utilized both internally at SOFT and externally—for instance with HackYourFuture Belgium. It has facilitated dozens of Bachelor and Master theses and has been extensively employed in two PhD dissertations [94, 127]. For these reasons, we argue that our implementation is worth mentioning as a technical contribution.

Practical considerations

Most of the techniques presented in this dissertation are sensible and practical. However, there are two exceptions: our composition of advices into layers and our centralization of analysis-specific logic. Both come with significant performance overhead; the former due to instrumentation stacking and the latter because of heavy synchronous communication load. We believe that additional work is necessary to manage performance overhead before these techniques can be applied in real-world contexts. Nonetheless, due to their expressiveness, these techniques remain valuable for research, experimentation, and demonstration purposes.

1.4 Supporting publications

This dissertation is supported by several publications, which we categorize as primary or secondary. Primary publications directly support a specific chapter, while secondary publications build upon the work presented here, demonstrating extensions, practical applications, and the significance of our approach.

Primary supporting publications

- Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. "Aran: JavaScript Instrumentation for Heavyweight Dynamic Analysis". English. In: *Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop (BENEVOL)*. vol. 3941. Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop. Belgium-Netherlands Software Evolution Workshop 2024, Benevol 2024 ; Conference date: 21-11-2024 Through 22-11-2024. CEUR Workshop Proceedings, Nov. 2024, pp. 97–114. URL: https://benevol2024.github.io

  ---

  In this paper, we introduce AranLang, our core variant of JavaScript, along with our standard join point model. This paper supports much of Chapters 3 and 4; however, the evaluation in this dissertation is more comprehensive and includes deploying Aran onto Octane to provide additional insights into performance overhead.

- Laurent Christophe, Wolfgang De Meuter, Elisa Gonzalez Boix, and Coen De Roover. "Linvail: A General-Purpose Platform for Shadow Execution of JavaScript". English. In: *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER2016) ; Conference date: 14-03-2016 Through 18-03-2016. IEEE CS, Mar. 2016, pp. 260–270. ISBN: 978-1-5090-1855-0

  In this paper, we present our adaptation of the membrane pattern to preserve semantic transparency

by controlling access to promoted values. This paper supports much of Chapter 5 and Chapter 6, except for the evaluation. The evaluation in this paper used an earlier version of the Aran Linvail stack, which could not support JavaScript features added after the 2015 Harmony specification. Therefore, we tested our approach only against the SunSpider benchmark. In contrast, we provide a more comprehensive evaluation in Chapter 6 by deploying the Aran Linvail stack onto Test262, Octane, and even Aran itself.

- Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. "Orchestrating Dynamic Analyses of Distributed Processes for Full-Stack JavaScript Programs". English. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming (GPCE)*. ed. by Eric Van Wyk and Tiark Rompf. International Conference on Generative Programming: Concepts & Experience, GPCE ; Conference date: 05-11-2018 Through 06-11-2018. ACM, Nov. 2018, pp. 107–118. ISBN: 978-1-4503-6045-6. DOI: 10.1145/3278122.3278135

  In this paper, we present our approach for centralizing analysis-specific logic within a separate process. This work supports much of Chapter 7. Although the overall approach and evaluation remain largely unchanged, we revised the interface of our implementation and the description of our communication protocol.

Secondary supporting publications

- Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. "Practical Information Flow Control for Web Applications". English. In: *Proceedings of the 18th International Conference on Runtime Verification (RV)*. vol. 11237. 18th International Conference on Runtime Verification, RV ; Conference date: 11-11-2018 Through 13-11-2018. Springer, Nov. 2018, pp. 372–388. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_21

  This paper builds upon the approach of this dissertation to analyze single-process JavaScript applications for detecting violations of information policies at run-time. The specificity of this paper's approach lies in its combination of two methods for implementing shadow values to achieve high precision. When values are assigned to regular objects, value promotion is the preferred mechanism for tracking them. However, when values are assigned to exotic objects, such as DOM nodes, an alternative approach based on shadow execution is employed. This method involves maintaining a copy of the exotic object with taint information. This paper demonstrates that the approach presented in this dissertation can be integrated with traditional methods for tracking provenance.

- Maarten Vandercammen, Laurent Christophe, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. "Prioritising Server Side Reachability via Inter-process Concolic Testing". English. In: *The Art, Science, and Engineering of Programming* 5.2 (Oct. 2020). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2021/5/5

  In this paper, we build upon the approach of this dissertation to develop a concolic tester for distributed JavaScript applications. The key feature of our concolic tester is its ability to generate symbolic constraints for inputs that span different processes, thereby uncovering distributed execution paths that are guaranteed to be feasible in practice. In contrast, applying concolic testing to each individual process of a distributed application in isolation may lead to the exploration of execution paths that do not occur in production, resulting in false alarms. This paper shows that our approach is capable of lifting concolic testing, a sophisticated dynamic analysis technique, to a distributed context.

- Abel Stuker, Aäron Munsters, Angel Luis Scull Pupo, Laurent Christophe, and Elisa Gonzalez Boix. "JASMaint: Portable Multi-language Taint Analysis for the Web". English. In: *Proceedings of the 22nd International Conference on Managed Programming Languages and Runtimes (MPLR)*. Belgium-Netherlands Software Evolution Workshop 2024, Benevol 2024 ; Conference date: 21-11-2024 Through 22-11-2024. ACM, Oct. 2025. URL: https://conf.researchr.org/home/icfp-splash-2025/mplr-2025

In this paper, we build upon the approach of this dissertation to conduct dynamic taint analysis on single-process applications written in both JavaScript and WebAssembly. The key feature of this work is its ability to track the provenance of data across various languages with fundamentally different abstraction levels. In contrast to existing approaches, our method does not require manually defining taint signatures and is able to retain taint information for data that escapes through side effects. This paper demonstrates that our approach is sufficiently flexible to interface with insights derived from a foreign dynamic analysis.

1.5 Roadmap of the chapters

Figure 1.1 depicts the roadmap of this dissertation, whose remaining chapters read as follows:

- In Chapter 2, we provide an overview of the broad topic of program analysis and articulate the main problem statement of this dissertation: tracking the provenance of immutable run-time values that have been interned or inlined in complex managed languages.
- In Chapter 3, we introduce AranLang, a core variant of JavaScript, and discuss code transformations necessary for transpiling JavaScript to AranLang and subsequently back to JavaScript.
- In Chapter 4, we present a join point model for AranLang that is comprehensive enough to enable shadow execution while maintaining a manageable level of complexity. Additionally, we evaluate the semantic transparency and performance overhead of our method by applying it to Test262 (i.e., the official conformance test suite for ECMAScript) and Octane (a popular performance benchmark from Google).
- In Chapter 5, we describe a technique for tracking the provenance of values through a combination of value promotion and value virtualization. To separate concerns between provenance tracking and analysis-specific logic, we propose two additional techniques: advice layering and a novel metric to assess the precision of a provenance provider.
- In Chapter 6, we evaluate the semantic transparency, performance overhead, and precision of our approach to tracking provenance by deploying it on Test262, Octane, and our own instrumenter.
- In Chapter 7, we present an approach for analyzing distributed applications where the analysis-specific logic is executed in a separate process. We evaluate our approach by conducting symbolic execution on a minimal collaborative drawing application that remains usable during the analysis.
- In Chapter 8, we conclude the dissertation by assessing our approach against objective evaluation criteria and discussing future research directions.

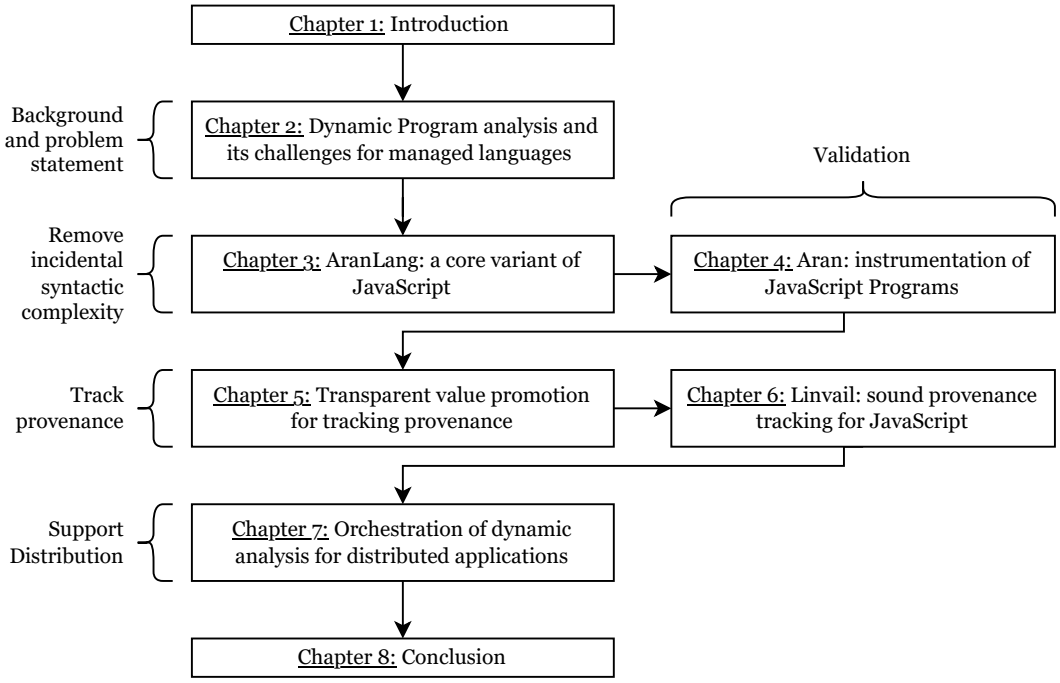


Figure 1.1: Chapter diagram of this dissertation

Chapter 2

Dynamic program analysis and its challenges for managed languages

In this chapter, we focus the dissertation on an approach to building dynamic analyses of programs written in managed languages via source code instrumentation. We emphasize support for analyses that require information about the provenance of values, which is data that cannot be derived directly from the normal execution of programs. This chapter has two primary goals: first, to establish the terminology that will be used throughout the dissertation; and second, to refine the problem statement by articulating it in terms of specific evaluation criteria. This chapter is organized as follows:

- *Section 2.1 introduces the broad topic of program analysis.*
- *Section 2.2 examines the design space for performing dynamic program analysis.*
- *Section 2.3 discusses the difficulty of tracking the provenance of immutable values in managed languages.*
- *Section 2.4 concludes the chapter and presents evaluation criteria.*

2.1 Program analysis: definitions and applications

In this section, we briefly introduce the broad topic of program analysis. The goal is twofold: first, to familiarize non-expert readers with the established terminology related to program analysis; second, to provide a sample of dynamic analyses which will be used to evaluate the expressiveness of our approach. This section is organized as follows:

- *Section 2.1.1 defines some important concepts related to program analysis and discusses its fundamental limitations.*
- *Section 2.1.2 discusses some key differences between static program analysis techniques and dynamic program analysis techniques.*
- *Section 2.1.3 presents some prominent applications of dynamic program analysis in the following four important domains: software correctness, software security, software optimization, and software maintenance.*
- *Section 2.1.4 provides a representative selection of dynamic analysis techniques.*

2.1.1 Fundamental limitations of program analysis

In this section, we restate important elements of computational theory. The goal is to provide non-expert readers with an understanding of the fundamental limitations of program analysis.

Program analysis is the field that studies the *behavior* of *programs*. The behavior of a program can be defined with respect to a discrete state machine as its *state transition system* after it has loaded the program. In the literature, this approach is usually called operational semantics [91, 135] and the state machine corresponds to the concept of an *interpreter*. For instance, Figure 2.1 illustrates the state transition system that a JavaScript interpreter will exhibit after loading the program from Listing 2.1.

```
1 const fac = (n) => n === 0 ? 1 : n * fac(n - 1);
2 const input = prompt();
3 const output = fac(parseInt(input));
4 alert(output);
```

Listing 2.1: JavaScript program for computing the factorial of a number

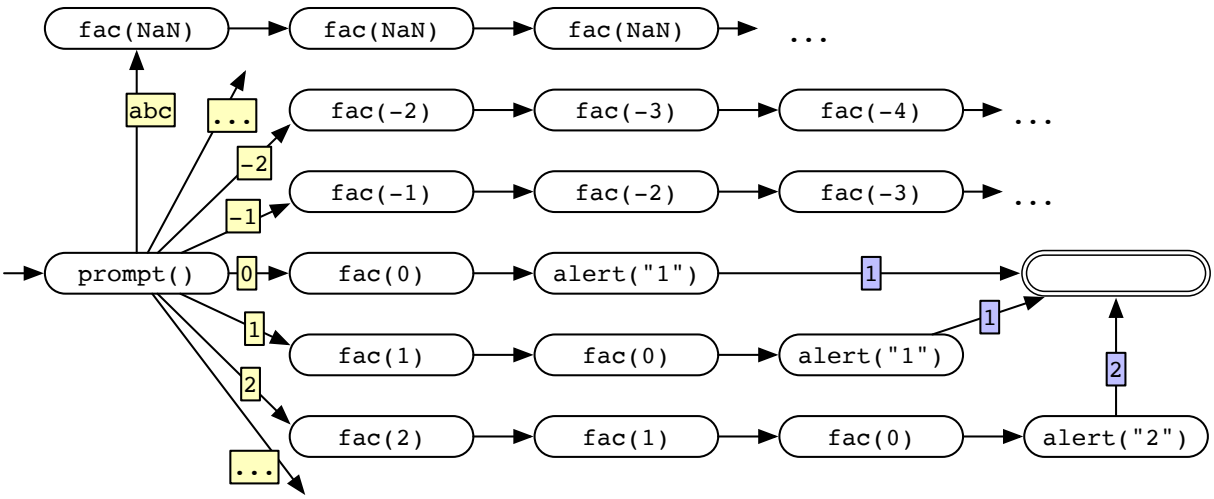


Figure 2.1: Simplified state diagram of the program from Listing 2.1

It is important to realize that the *problems* studied in the field of program analysis do not concern the programs themselves, which are always finite, but rather the transition systems they encode relative to an interpreter, which can be infinite. For instance, Listing 2.1 encodes a transition system that runs indefinitely when executed with a negative input[1]. Therefore, it is unclear which mathematically defined *function* over these infinite abstract objects can actually be computed in practice.

Unfortunately, computability theory establishes fundamental limits on what can be determined about program behavior, most notably through the undecidability of the halting problem. This theorem states that it is impossible to decide whether an arbitrary transition system (potentially infinite but encoded as a finite program) will eventually halt on a given finite input. As per Rice's theorem [97], this result has profound implications, since any non-trivial observable behavior of a program is undecidable. For example, determining whether a particular problematic state is reachable can be framed as an instance of the halting problem.

It follows that the field of program analysis is inherently limited to developing mechanical methods that can only approximate exact solutions for any given program analysis problem. We refer to these mechanical methods as *techniques*. For instance, when a program analysis technique investigates a decision problem, it may conclude `yes` when the correct answer is `no`, and conversely, it may conclude `no` when the correct answer is `yes`. The first case is termed a *false positive*, while the second case is termed a *false negative*.

[1] In practice, because the memory of interpreters is bounded, the state space is finite but enormous due to combinatorial explosion.

Perhaps the most prominent class of problems studied in program analysis involves *verifying* that the behavior of a target program adheres to a given *specification*. In simple cases, the specification defines only unwanted states—e.g., "Are divisions always performed with a non-zero divisor?" or "Will arrays always be accessed within their bounds?". More generally, elements of temporal logics such as LTL [92] can be incorporated into the specification—e.g., "A beverage cannot be dispensed until a coin has been received." Table 2.1 summarizes the terminology associated with false negatives and false positives, depending on whether the technique attempts to prove the presence or absence of defects. Bolded outcomes are undesirable and unavoidable for decision problems that are reducible to the halting problem.

	Correct answer is `yes`	Correct answer is `no`
Technique does not terminate or concludes `maybe`	**Not a decision procedure**	
Technique concludes `yes`	True positive	**False positive** **→ Unsound (safety)** **→ Incomplete (bug finding)**
Technique concludes `no`	**False negative** **→ Incomplete (safety)** **→ Unsound (bug finding)**	True negative

Table 2.1: Outcome categories for a program analysis technique on a decision problem

2.1.2 Approximation strategies for program analysis

In this section, we briefly compare the two main approaches to ensure the termination of program analysis techniques: dynamic program analysis and static program analysis. The goal is to provide non-expert readers with an understanding of the strengths and weaknesses of both approaches.

To ensure termination, the most straightforward and common approach involves analyzing a finite approximation of the program's transition system. In this context, two main strategies are commonly employed:

- The first strategy sacrifices a global view of the transition system by concentrating on a single finite execution path. Program analysis techniques based on this strategy are typically referred to as *dynamic* and involve executing the program in a run-time environment that resembles production. The goal is to *under*-approximate the behavior of the target program, ensuring that the studied execution can actually occur in production.
- In contrast, the second strategy involves abstracting multiple concrete states into a single abstract state. Techniques based on this strategy are called *static*, as they analyze programs without executing them in a production-like runtime. Often, these techniques attempt to *over*-approximate the behavior of the target program by ensuring that every single concrete transition is represented in the finite abstract representation.

When verifying that a target program adheres to a specification, static program analysis techniques are valued for their potential to prove the absence of defects, provided that they consistently over-approximate the behavior of the target program. Conversely, the strength of dynamic program analysis techniques lies in their ability to prove the presence of defects and avoid false alarms, provided that they consistently under-approximate the behavior of the program.

Table 2.2 summarizes the key elements of comparison between dynamic and static program analysis techniques. This presentation clearly demonstrates that dynamic and static techniques are complementary

approaches to solving program analysis problems. In this dissertation, we introduce an approach for developing dynamic program analysis tools. In doing so, we must consider the extent to which our tools examine an actual under-approximation of the target program's behavior, as this is directly related to the rate of false alarms. We further discuss this point in Section 2.2.1.

	Dynamic Analysis	**Static Analysis**
Study Subject	Single finite concrete execution path	Finite transition system obtained by collapsing concrete states into abstract states
Reuse Potential	*Many* components of actual runtimes and environments	*Few* components of actual runtimes and environments
Precision	Often attempts to *under*-approximate program behavior	Often attempts to *over*-approximate program behavior
Guarantee	An exact under-approximation can prove the *presence* of defects and avoid false alarms	An exact over-approximation can prove the *absence* of defects
Examples	Testing, dynamic type checking, dynamic information-flow analysis, profiling, ...	Static type checking, static information-flow analysis, k-CFAs, ...

Table 2.2: Elements of comparison between dynamic and static program analysis

2.1.3 Important applications of program analysis

In this section, we discuss several important application domains of program analysis, focusing on dynamic program analysis techniques. The goal is to introduce representative dynamic program analysis techniques that will serve as benchmarks for evaluating the applicability of our own approach.

Next, we present four important application domains of program analysis. We do not claim that this section offers a comprehensive overview; rather, it provides a fair and representative sample. For more extensive surveys of dynamic program analysis, we refer the reader to [137, 31, 26, 47].

Program analysis for software correctness

This first application we discuss is program correctness, which is perhaps the most prevalent application of program analysis. It involves ensuring that a program behaves according to a given specification.

Testing is crucial in assessing software correctness [80]. It consists of two intertwined activities: test case implementation and test case execution. A *test case* represents a scenario that provides *test input* and a *test oracle* [52] to determine whether the test passed or failed based on the output. A *test suite* is a collection of test cases. There are two main approaches to implementing test cases: *black box* and *white box* [85]. The black box approach implements test cases based solely on the software specification, without any knowledge of the implementation. In contrast, the white box approach implements test cases using information about the software's implementation. Test cases can be executed either manually or automatically within a testing framework. Some frameworks, such as the xUnit family [8], specialize in generating human-readable reports and providing metrics like code coverage to assess the quality of the test suite. Others, like Selenium[2], provide an API to interact with the program via its graphical user interface (GUI).

[2] `https://www.seleniumhq.org`

Type checking is another essential approach for ensuring software correctness. It involves verifying that the program is free of *type errors*, which can be understood as performing meaningless operations [89], such as executing an integer division on byte representations of characters. Type checking enhances program correctness by converting these silent errors into explicit errors that are easier to detect and correct. Unlike testing, type checking is often part of the language specification and can be performed either statically at compile-time or dynamically at run-time. When conducted statically, the standard approach for type checking involves constructing a formal proof that demonstrates the program is free of type errors [89]. Unfortunately, type checking is reducible to the halting problem, which means that conservative static type checkers may produce false alarms. In practice, developers often do not investigate the validity of these alarms; instead, they modify their programs to eliminate all such warnings indiscriminately. Compilers typically enforce this practice by refusing to compile programs that fail type checking. When performed dynamically, type checking usually involves attaching type information to values and verifying that built-in operations receive values of the correct type. In this case, type errors are detected just before they occur.

Whereas type checking enforces structural and representational constraints, *contracts* [77] specify behavioral and semantic constraints. Contracts typically consist of preconditions, which must hold before an operation is invoked; postconditions, which must hold after it completes; and invariants, which must hold throughout the lifetime of a component. Contracts are much more expressive than types, but they cannot, in general, be checked statically and therefore may require runtime mechanisms to detect violations.

Finally, we discuss *model checking*, an approach that is mostly used to ensure program correctness in safety-critical systems [6]. Model checking investigates whether the behavior of a given program satisfies a specified property expressed in a formal framework such as LTL [92] and CTL [24]. These frameworks can encode temporal properties such as "A program will always eventually terminate", "Arrays are never accessed outside their boundaries", and "A beverage cannot be dispensed before a coin has been inserted". Since model checkers operate on finite models, they require finite approximations of program behaviors. While most approaches focus on providing static over-approximations, dynamic under-approximations have also been proposed [40, 50, 49].

Program analysis for software security

A second important application of program analysis is software security. Software security is closely related to software correctness but primarily focuses on three aspects: *confidentiality*, *integrity*, and *availability* [76]. Confidentiality involves protecting against the unintentional disclosure of sensitive information. Integrity ensures protection against the unintentional alteration of sensitive information. Availability guarantees that authorized access to resources remains reliable.

A common approach to studying the integrity and confidentiality of programs is *information-flow analysis*, which investigates the propagation of data through programs[3]. Information-flow analysis relies on the analyst to specify sources and sinks, which are parts of their programs that can produce or consume data. Each source and sink should then be associated with a sensitivity level. Confidentiality can be defined as preventing data from higher-sensitivity sources from flowing to lower-sensitivity sinks. Conversely, integrity can be defined in terms of the absence of data from lower-sensitivity sources flowing to higher-sensitivity sinks. A common example of an integrity violation is SQL injection, where an attacker inputs data that ends up inside a SQL request without proper sanitization, enabling them to perform unauthorized data manipulation. Information-flow analysis can be conducted statically, for instance, by constructing a formal proof that demonstrates the program cannot exhibit forbidden information flow [28]. Alternatively, it can be performed dynamically by attaching taint information to values, a technique akin to dynamic type checking [25].

Automatic test case generation represents another significant application of program analysis within the realm of software security. Indeed, software vulnerabilities can be considered software defects that attackers can exploit to compromise the confidentiality, integrity, and/or availability of the program. To uncover defects that reside in corner cases, automated methods for defining test cases have proven to be

[3]In the literature, information-flow analysis is sometimes treated as a special case of data-flow analysis. However, we find it useful to distinguish between the two: data-flow analysis focuses on the content of values, whereas information-flow analysis focuses on their provenance.

beneficial. The most straightforward method for automatically generating test inputs is random generation. Although simple, this technique has proven useful for catching hard-to-find bugs [23]. However, random testing may miss corner cases that require precise input conditions. To address this limitation, it is necessary to derive information from the target program to guide input generation in order to improve test metrics such as code coverage. In this context, *symbolic execution* has proven effective; it involves gathering mathematical expressions known as *path constraints* on the program's inputs, which, when satisfied, will lead the program along a specific execution path.

The original technique for performing symbolic execution involves statically constructing a symbolic execution tree [63]. For each path in the symbolic execution tree, path constraints are collected and fed to an SMT solver to obtain an input that will lead the program down that particular execution path. More recently, dynamic techniques have been proposed [42, 104]. During program execution, inputs are tracked so that the constraints associated with the current execution path can be gathered. Then, one of the path constraints is negated, and the result is fed to an SMT solver. If a solution is found, providing it as input will steer the program to another execution path, provided that sources of indeterminism have been eliminated. More recent approaches achieve better results by combining symbolic execution with fuzzing techniques [11].

Program analysis for software optimization

A third important application domain of program analysis is software optimization. It is the process of transforming a program so that it behaves more efficiently regarding some aspects such as execution time, memory usage, or energy consumption.

Compilers often perform some form of software optimization, which requires them to analyze the program at hand. In that context, they often employ *data-flow analysis* [2], which investigates the possible values that a given expression in the program can take. The traditional approach for conducting data-flow analysis involves decorating the control-flow graph with data-flow equations and iteratively solving them until the system reaches a fixed point [61]. For example, constant propagation is a common software optimization technique used by compilers. It involves replacing expressions, whose value can be determined at compile-time, with their actual value. In practice, this optimization is facilitated by a program analysis technique known as *reaching definitions*, a data-flow analysis method that conservatively gathers the variable definitions in effect.

Developers also manually optimize their programs. To implement effective optimizations, they need to understand their programs' behavior to identify performance bottlenecks. In this context, *program profiling* is commonly used [114]. This technique involves extracting performance-related information during the execution of the target program, which can include memory consumption, execution time, and the frequency of function calls.

Program analysis for software maintenance

Lastly, a fourth important application of program analysis is software maintenance [90]. Software maintenance involves modifying a program after its release to correct defects, optimize performance, or ensure usability in a changing environment.

In complex systems, even small changes can have unforeseen effects. These unintended effects are known as *ripple* effects and can lead developers to introduce new defects during maintenance tasks. The detection of ripple effects is the focus of dependence-based change impact analysis, a subfield of change impact analysis [12]. In practice, dependence-based change impact analysis is performed using various approaches, most of which rely on some form of *control-flow analysis*, which investigates the control-flow graph of a program [70]. For most low-level programming languages, the control-flow graph is typically explicit in the source code [4]. For instance, in the programming language C, the only source of uncertainty in constructing the control-flow graph is the use of function pointers, which is generally rare. In higher-level programming languages, two features complicate the construction of the control-flow graph:

dynamic method dispatching and higher-order functions. *k-CFA* algorithms were developed to address this issue; they simulate the program to gain insights into types and function values [111].

Another aspect of software maintenance that benefits from program analysis is debugging, the process of identifying, localizing, and correcting defects in a program to restore intended behavior. Debugging requires the developer to understand the inner workings of the program to identify the root cause of the defect and ultimately correct it. Whether the developer is stepping through a debugger, proofreading the source code, or inserting print statements, they are performing program analysis. One approach that helps developers locate the root cause of a defect is *program slicing*, which investigates the statements that may be involved in the computation of a given expression in the program. In the context of debugging, program slicing can be used to eliminate parts of the program that cannot be responsible for the defect, thereby narrowing the search for the root cause. Originally, slices were conservatively computed using a form of static data-flow analysis [133]. Later, another approach was proposed that relied on executing the program to observe the statements that were actually involved for a given input [65].

2.1.4 Sample of dynamic program analysis techniques

In the previous section, we introduced several domains *of program analysis by providing a high-level overview of the problems they address, along with established approaches to solve them and key applications. In this section, we present a representative sample of dynamic program analysis techniques drawn from this overview.*

Table 2.3 summarizes our sample of dynamic program analysis techniques. This collection is by no means exhaustive; for a comprehensive overview of dynamic program analysis approaches, we refer the reader to the surveys by Yang et al. [137], Egele et al. [31], and Cornelissen et al. [26]. However, we believe this collection offers a representative sample of the technical challenges encountered when building dynamic program analysis tools, making it suitable for evaluating both the **cross-analysis applicability** of our approach and its **expressiveness**.

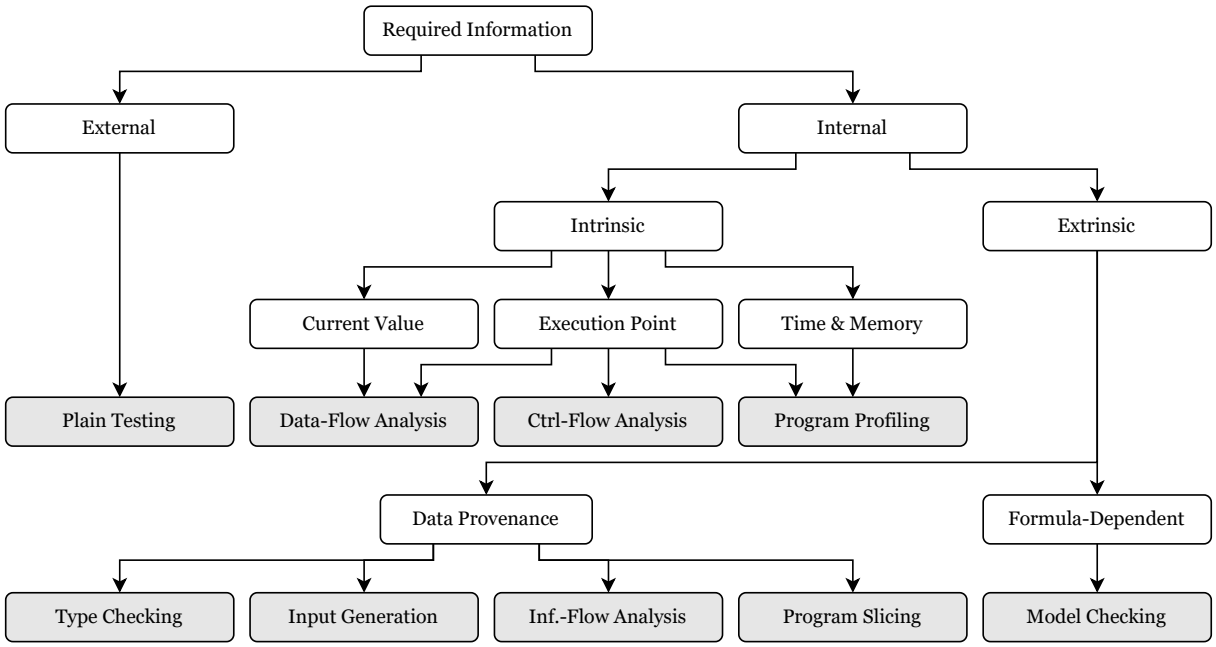


Figure 2.2: Categorization of the program analysis approaches listed in Table 2.3

Domain	Prototypical Problem	Standard Approach	Typical Application
Plain Testing	Will a test scenario pass?	Run the program with the test scenario input and check whether the output meets the specification (dynamic).	Bug detection and validation (correctness).
Type Checking	Is the program free of type errors?	Use a type system to construct a formal proof demonstrating that the program is free of type errors (static). Alternatively, attach type information to values to detect type errors just before they occur (dynamic).	Virtually all compilers and interpreters perform some form of type checking (correctness).
Model Checking	Does the program yield a transition system that satisfies a given property?	Obtain a finite model by over-approximation (static) or by under-approximation (dynamic). Subsequently, apply techniques from automata theory to verify that the model satisfies the specified property.	Verification of safety-critical programs (correctness).
Input Generation	What sequence of test inputs will maximize code coverage?	Construct a symbolic execution tree and use an SMT solver to identify an input for each path (static). Or, execute the program while collecting path constraints and solve them to guide the next execution (dynamic).	Detection of hard-to-find bugs that may be exploitable by an attacker (correctness, security).
Inf.-Flow Analysis	Can information flow from a given source in the program to a specified sink?	Utilize a data-flow analysis to construct a proof demonstrating that the program cannot perform forbidden information flow (static). Alternatively, attach taint information to values and flag any forbidden information flow just before it occurs (dynamic)	Detection and prevention of program vulnerabilities (security).
Data-Flow Analysis	What potential values can an expression take at run-time?	Derive data-flow equations from a control-flow graph and solve the system (static). Monitor the run-time values at the given code point (dynamic).	Support for compiler optimizations (optimization) and detection of invariant violations (correctness).
Program Profiling	Which parts of the program consume the most resources during execution?	Sample execution points, timing, and memory usage during execution (dynamic).	Identify performance bottlenecks (optimization, maintenance).
Ctrl.-Flow Analysis	Which functions may be called at a given call site?	Control-flow analysis (CFA) algorithms which often rely on abstract interpretation (static). Or, execute and trace the program to uncover parts of the control-flow graph (dynamic).	Identify ripple effects when modifying an existing program (maintenance).
Program Slicing	Which parts of the program affect the value of a given variable?	Conduct backward data-flow analysis to trace dependencies between statements (static) or monitor the statements involved during execution (dynamic).	Narrow the search for the root cause of a defect (maintenance).

Table 2.3: Sample of program analysis approaches

Figure 2.2 categorizes dynamic analysis techniques based on their information needs, which significantly influence their implementation. We classify information about program execution as either *public* or *private*, depending on whether it can be directly derived by observing the program's interaction with its environment. Techniques consuming private information require modifying the execution of the target program, adding complexity compared to analyses that only consume public information, such as plain testing.

We further divide private information into two categories, based on whether it requires additional state management. On the one hand, *intrinsic information* can be directly derived from the internal state of the interpreter at any given execution point, without relying on prior knowledge. On the other hand, *extrinsic information* cannot be obtained from the current state alone and requires information from earlier execution steps. Techniques that depend on extrinsic information thus require additional bookkeeping mechanisms, which add complexity compared to analyses that rely solely on intrinsic information.

Techniques such as dynamic data-flow analysis, dynamic control-flow analysis, and dynamic program profiling primarily rely on intrinsic information—e.g., the current program location, the value of the active expression, the current time, or memory usage. While these techniques may employ sophisticated methods to reduce performance overhead and generate high-quality reports, they fundamentally operate by logging information that is immediately accessible during execution.

This is not the case for dynamic analysis techniques that require extrinsic information. For instance, in an untyped language such as assembly, a dynamic type checker must determine whether performing integer addition on two sequences of bytes is type-safe. Making such a decision requires context about how these bytes were created. The situation is similar for input generation, which requires understanding how data used for branching relates to inputs; dynamic information-flow analysis, which needs to assess the sensitivity of data; and program slicing, which requires knowledge of the code locations where data were created. We refer to the context of data creation as *provenance*. Model checking also necessitates managing stateful analysis data, but it depends on the specifics of the formula. For example, the property "never dispenses a beverage before receiving a coin" requires knowledge of whether a coin has been received.

Due to the additional run-time machinery required to manage stateful analysis data, analysis tools that require extrinsic information typically incur a higher performance overhead than those that rely solely on intrinsic information. As we will discuss in Section 2.3, maintaining **accurate** extrinsic information is not always straightforward and may sometimes require approximation.

2.2 Overview of the design space to conduct dynamic program analysis

In this section, we review the design space for conducting dynamic program analysis. The goal is to outline our design decisions and position our approach within the broader field of dynamic program analysis. This section is organized as follows:

- *Section 2.2.1 discusses software instrumentation in general and emphasizes the importance of transparency.*
- *Section 2.2.2 examines the abstraction levels at which programs can be analyzed.*
- *Section 2.2.3 explores the primary approaches to software instrumentation.*
- *Section 2.2.4 contrasts live analysis with postmortem analysis.*

2.2.1 Software instrumentation and transparency

In this section, we explain why dynamic program analysis may require instrumentation and emphasize the importance of ensuring that this process remains transparent.

As mentioned in Section 2.1.2, the process by which dynamic program analysis gathers information about program behavior resembles execution in production. Figure 2.3 depicts a simplified execution model for programs. As noted in Section 2.1.4, in simple cases, program analysis techniques only examine the external behavior of programs, which is represented as input/output in the figure. However, in more complex scenarios, dynamic program analysis techniques require modifications to how programs are executed in order to access their internal behavior. This process, known as *instrumentation*, involves altering either the target program, the language runtime, the operating system or the hardware. Generally, hardware instrumentation is too costly and inflexible, making software instrumentation the preferred approach.

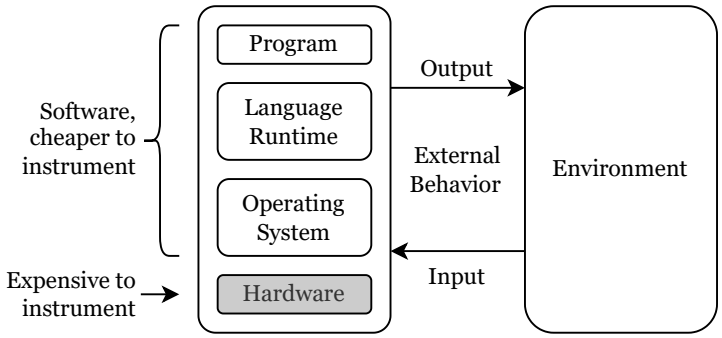


Figure 2.3: High-level overview of the execution of programs in practice

After software instrumentation, the behavior of the program on the hardware is *augmented* with analysis logic. Conceptually, when dynamically analyzing programs, two layers coexist: the *base layer*, which performs the computations of the target program, and the *meta layer*, which performs analysis-related computations. It is important for the meta layer to remain **transparent** [72, 83, 13] to the base layer; otherwise, the outcome of the analysis may become unreliable. As soon as the meta layer interferes with the base layer, the meta layer risks no longer studying an exact under-approximation of the behavior of the target program. In other words, the examined states and transitions may no longer be possible during the baseline execution of the program. We refer to this type of adverse situation as a *discrepancy*.

More formally, as stated in Section 2.1.1, the behavior of a program can be modeled by a state transition system. In this formalism, interactions with the environment are typically represented by labeling state transitions with actions, such as receiving user input from a text box or logging a string to the console. Transparency can be assessed by comparing the labeled transition system of the baseline execution of the program with that of its augmented execution.

In the literature, the strictest comparison between labeled transition systems is called *bisimulation* [6], which is based on a relation between the states of the two systems. It requires that each labeled transition in the first system be represented in the second system and vice versa. Thus, two bisimilar labeled transition systems interact with the environment in exactly the same way. However, as illustrated in Figure 2.4, bisimulation is too strict for describing transparency and must be relaxed to accommodate two types of divergences:

- First, by incorporating analysis-related logic, the instrumentation is bound to introduce intermediary states in the augmented execution that do not exist in the baseline execution. In the literature, this equivalence is referred to as *stuttering bisimulation* [6], which involves introducing a "silent" label (`e` in the figure) that does not translate to any observable effect.
- Second, the insights of the analysis must be communicated to the outside world, but in a controlled manner so as not to affect base-level interactions. This can be modeled by separating state transition labels into two groups based on whether they describe interactions in the base layer (`bX` in the figure)

or in the meta layer (`mX` in the figure). We can then consider meta labels as silent for the purpose of establishing a stuttering bisimulation.

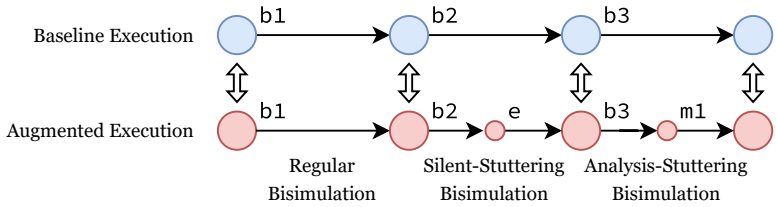


Figure 2.4: Transparent instrumentation modeled as stuttering bisimulation

Importantly, this formalism for modeling the transparency of instrumentation does not account for the unavoidable *performance overhead* introduced by the analysis. For instance, memory overhead can lead to out-of-memory errors, and time-complexity overhead can result in event interleavings that may not occur in the baseline execution. We qualify these discrepancies as *performance-related* , whereas the discrepancies modeled by our formalism are qualified as *semantics-related* and collectively referred to as *semantic overhead* .

To summarize, instrumentation is necessary for conducting dynamic program analysis techniques that consume private information about program execution. To preserve the primary advantage of dynamic analysis (i.e., its ability to under-approximate program behavior), instrumentation must remain transparent and not introduce discrepancies, which can be either semantics-related or performance-related.

2.2.2 Abstraction levels at which programs can be analyzed

In this section, we present the three primary abstraction levels at which programs can be analyzed and discuss their trade-offs.

In practice, programs can be represented at three primary abstraction levels. The lowest level is machine code, which is executed directly by the hardware. The next level is bytecode, which serves as an intermediate representation that can either be executed by a virtual machine, such as the JVM, or compiled into machine code, as is the case with LLVM IR. Finally, the highest level is source code, typically written in a structured programming language that includes variables, loops, and procedures. Source code can be executed by a virtual machine (e.g., JavaScript, Python, Smalltalk, Ruby) or compiled into either bytecode (e.g., Java, Kotlin, C#) or machine code (e.g., C, Rust, Go).

Each of these abstraction levels defines a different granularity at which a program's execution can be analyzed. At the machine code level, the state space aligns with that of a register machine, with each transition corresponding to the execution of a single machine instruction. At the bytecode level, the state space is typically that of either a register machine or a stack machine, and each transition results from executing a single bytecode instruction. At the source code level, the state space and transitions are determined by a high-level virtual machine capable of interpreting the code. It's worth noting that such a machine can always be implemented, even for programs written in traditionally compiled languages such as C [36, 53].

The appropriate abstraction level for a program analysis depends on the problem being studied. For example, analyzing performance may require examining cache misses, which are best studied at the machine code level rather than at the source code level. Conversely, analyses related to program comprehension are often more effectively conducted at the source code level. When the analysis examines a low-level execution trace, providing high-level insights to the programmer becomes more difficult. While techniques like source mapping can help mitigate this issue, they cannot fully eliminate it. For instance, consider taint analysis at the bit level; it is unclear how a programmer should address the issue when a sensitive bit is transferred to an untrusted sink. In contrast, when the analysis operates on a high-level trace that includes concepts like closures in a SECD machine [66], the insights can be linked to the source code more easily. We refer to this ability as **source-adjacency**.

Another important differentiating criterion between abstraction levels is **portability**, which refers to the capacity of the analysis to be deployed across different execution setups with minimal effort. Analyzing programs at the machine code level is inherently tied to a specific hardware instruction set architecture; however, it can analyze programs compiled from various source languages. In contrast, analyzing programs at the source code level is closely linked to the specifications of the source language but can potentially remain independent of the rest of the execution setup. Finally, analyzing languages at the bytecode level enables the examination of programs from different sources across various hardware platforms; however, it requires the program to be represented in the targeted bytecode format within the execution setup.

In this dissertation, we focus on analyzing programs at the source code level, aiming to preserve the strengths associated with this abstraction level: source-adjacent insights and portability across runtimes and hardware.

2.2.3 Main approaches to software instrumentation

In this section, we review the three main instrumentation strategies that we identified in the literature.

Dynamically analyzing programs at the source code level requires either instrumenting the source code or, for managed languages, the interpreter—i.e., the virtual machine responsible for directly executing the program's source code.

First, we focus on interpreter instrumentation. The main advantage of interpreter instrumentation over program instrumentation is its ability to derive information that is difficult to obtain from within the language by bridging the semantic gap between the guest language and the underlying runtime. For instance, as discussed in Section 2.3, modifying the allocation system provides a straightforward way to track provenance. The main weakness of interpreter instrumentation is its coupling to the language runtime, which poses a maintainability challenge.

The least amount of coupling occurs when the interpreter exposes an interface for customization. For instance, Truffle [134] is a highly customizable interpreter capable of conducting a wide range of dynamic analyses. Sometimes, the customization interface is already tailored toward dynamic analysis. Information-flow analyses are common in this context; for example, Perl's taint mode [19], Ruby's security levels [120], and the taint mode for JavaScript in Netscape Navigator 3.0 [30].

The research community has also modified interpreters in an ad-hoc manner without a stable customization interface. For instance, [130] proposes performing information-flow analysis of client-side JavaScript programs using their modified SpiderMonkey virtual machine, which they acknowledge required a "considerable engineering effort." Another example is jsprobes [15], which implements a probe-like mechanism in browsers by utilizing a forked version of Firefox. Compared to using a customization interface, ad-hoc interpreter instrumentation is more generic but ties the analysis to not only a particular language runtime but also to a particular version. Because interpreters are often complex and rapidly evolving, ad-hoc instrumentation without a stable customization API is difficult to maintain.

Besides the interpreter, the other medium for instrumentation is the program itself. We identified two main trends in program instrumentation, which we refer to as *point-based instrumentation* and *hook-based instrumentation*. The point-based approach relies on introspection, which is the ability of a program to examine its own run-time structure, such as the call stack and the object graph. This approach typically requires extensive program transformations to insert the necessary measurement logic. In contrast, the hook-based approach relies on global intercession, which enables a language to modify its run-time behavior on a global scale—e.g., aspect-oriented programming [60, 37] and the `doesNotUnderstand` method in Smalltalk. This approach generally involves minimal program transformation and registers global hooks to perform measurements.

Point-based instrumentation can be challenging to implement because it requires careful consideration of the semantics of each transformed syntactic node to preserve transparency. In contrast, hook-based instrumentation does not require complex code transformations but depends on the intercession capabilities of the language. Notably, intercession mechanisms are rarely provided for primitive data types.

A final technical consideration in program instrumentation concerns its deployment. The first option involves transforming the program statically before execution. This approach is straightforward and completely independent of the execution environment; however, it can only handle static code. Consequently, dynamically loaded libraries and dynamically generated code remain invisible to the analysis. To address this limitation, dynamic instrumentation can be employed; however, it requires customization of the language runtime through preload hooks for hook-based instrumentation and import hooks for point-based instrumentation.

Table 2.4 summarizes the key elements for comparing the three main approaches to dynamic program analysis; using green, yellow, and red markers to indicate ideal, less-than-ideal, and undesirable situations, respectively. In this dissertation, we focus on point-based program instrumentation, whose primary advantage is its **applicability**; it makes minimal assumptions about the reflective capabilities of the languages and is independent of the runtime. While the deployment of the instrumentation (whether static or dynamic) will be discussed, it is not the primary focus of this dissertation.

Medium	**Bridging**	**Trend**	**Requirement**	**Timing**	**Coupling**
Interpreter	Host and guest	Interface	Customizable	Static	Runtime (stable)
		Ad-hoc	Open source	Static	Runtime (version)
Program	Guest only	Hook	Global intercession	Static	None
				Dynamic	Preload hook
		Point	Local introspection	Static	None
				Dynamic	Import hook

Table 2.4: Elements of comparison for instrumentation of managed languages

2.2.4 Evaluation strategies for deriving analysis insights

In this section, we discuss the evaluation strategies for deriving insights from the execution information.

Derivation of analysis insights occurs on a spectrum. On one end of the spectrum, insights are eagerly updated as new measurements become available; this approach is often termed *live*. Conversely, at the other end of the spectrum, measurements are collected during augmented execution, and conclusions are computed afterward. This strategy is often referred to as *postmortem*.

At first glance, each analysis technique can be implemented on both ends of this spectrum. However, dynamic analysis with an extrinsic information diet requires inspecting both the current and past states. While this can be done post-mortem by tracing sufficient information, it typically necessitates a *replay* phase, as demonstrated by Jalangi [105]. The live strategy simplifies this process by directly maintaining extrinsic information during the augmented execution of the target program. Furthermore, the postmortem strategy is not applicable if the analysis needs to modify the behavior of the target program, which is often necessary for security mechanisms intended for deployment in production, such as sandboxing.

An advantage of the postmortem strategy over the live strategy is the ability to examine the same execution path multiple times. This is particularly useful when dealing with sources of indeterminism in programs, such as random numbers, access to the current time, or time-sensitive event interleaving.

For instance, during debugging, one may want to step through the program's execution multiple times to accurately diagnose the root cause of a bug.

The two evaluation strategies have different performance implications. The live strategy requires maintaining the analysis state during execution, which can be costly. In contrast, the postmortem strategy necessitates logging measurements, which can be substantial for record and replay tools. Ultimately, the performance implications of these strategies are largely dependent on the specific analysis being conducted.

As stated in Section 2.1.4, we focus on dynamic analysis with an extrinsic information diet, which makes the live strategy more natural. However, the approach we describe is flexible enough to also implement the postmortem strategy. This is consistent with our findings in the literature that successful general-purpose frameworks for dynamic program analysis, such as ATOM [114], CIL [81], Pin [72], and Valgrind, have been designed with the live strategy in mind.

2.3 Tracking the provenance of values in managed languages

In this section, we outline the main problem statement of this dissertation by examining the limitations of using value identity in managed languages for provenance tracking. This section is organized as follows:

- *Section 2.3.1 presents the common granularity levels at which data can be represented and introduces the concept of value identity.*
- *Section 2.3.2 introduces a toy interpreter written in Haskell for a Scheme-like language, which has the unique ability to support different allocation systems.*
- *Section 2.3.3 presents a case study which consists of diagnosing the appearance of `NaN` in a simple program that solves quadratic equations.*
- *Section 2.3.4 discusses this case study in a first allocation system where all data types are naively stored.*
- *Section 2.3.5 examines the case study in a second allocation system where primitive data types are inlined into values.*
- *Section 2.3.6 explores the case study in a third allocation system where primitive values are interned for reuse.*
- *Section 2.3.7 concludes this section by formalizing the concept of provenancial equality.*

2.3.1 The identity of values in managed languages

In this section, we examine the granularity of data representation within programs and explain that precise value provenance cannot be derived solely through built-in comparison means.

Table 2.3 demonstrates that many dynamic program analysis techniques, such as taint analysis and symbolic execution, rely on the ability to track the provenance of data during program execution. We can distinguish different levels of granularity at which data is represented and manipulated. At the lowest level, hardware represents data as bits. At the highest level, managed languages like JavaScript and Python manipulate data at the *value* level. Unlike raw bits, values carry semantic meaning, and their memory representation may be of arbitrary length. For instance, a string value can contain any number of characters and should not be manipulated with numeric operations.

Note that the type systems of low-level languages like C provide some abstraction for manipulating data at the value level. However, this value abstraction can be circumvented with arbitrary pointer arithmetic, which effectively renders C a weakly typed language in practice. For instance, Listing 2.2 shows a C program that uses an `int` value as a group of four `char`s. This demonstrates that robust value abstraction requires the language to manage memory.

```c
#include <stdio.h>
int main () {
  int i = 0;
  int* p1 = &i;
  char* p2 = (char *) p1;
  *(p2+0) = 'a';
  *(p2+1) = 'b';
  *(p2+2) = 'c';
  *(p2+3) = 'd';
  printf("%i\n", i); // ❌ 1684234849
}
```

Listing 2.2: Direct memory access in C

As identified in our survey, tracking the provenance of data has been extensively studied at the byte level, as demonstrated by tools such as TaintBochs [18], Dytan [25], Valgrind [83], and Sage [43]. Most of these approaches can be adapted to operate at the bit level, with some specifically designed to operate at this granularity, such as Flayer [29] and BitTracker [136]. In this dissertation, we propose to track the provenance of data at the value level, an area that has received less attention in prior research and provides insights that are closer to the source code. In doing so, we will explore how high-level mechanisms such as referential equality and virtualization can be leveraged to track values.

The most precise built-in mechanism for tracking the provenance of values is the strictest comparison operator provided by the language, which defines the *identity* of values. Unfortunately, in mainstream managed languages, this comparison operator is often insufficiently strict, resulting in imprecise provenance tracking.

For instance, in JavaScript, the strictest comparison operator is provided by `Object.is(x, y)`[4], which defers to an axiomatic equality referred to as "is" in the specification, as described in Section 5.1.7[5] and inlined in Quotation 2.5. The specification uses the term "identity" differently than we do; it states that only certain data types, such as objects, possess identity, while others, such as numbers, are subject to structural comparison. Listing 2.3 demonstrates that `Object.is` can differentiate identity-provided data types, such as objects, based on their origin, but it cannot do so for other data types, such as numbers.

> In this specification, both specification values and ECMAScript language values are compared for equality. When comparing for equality, values fall into one of two categories. Values without identity are equal to other values without identity if all of their innate characteristics are the same – characteristics such as the magnitude of an integer or the length of a sequence. Values without identity may be manifest without prior reference by fully describing their characteristics. In contrast, each value with identity is unique and therefore only equal to itself. Values with identity are like values without identity but with an additional unguessable, unchangeable, universally-unique characteristic called identity. References to existing values with identity cannot be manifest simply by describing them, as the identity itself is indescribable; instead, references to these values must be explicitly passed from one place to another. Some values with identity are mutable and therefore can have their characteristics (except their identity) changed in-place, causing all holders of the value to observe the new characteristics. A value without identity is never equal to a value with identity.
> ...
> The ECMAScript language values without specification identity and without language identity are `undefined`, `null`, Booleans, Strings, Numbers, and BigInts. The ECMAScript language values with specification identity and language identity are Symbols not produced by `Symbol.for` and Objects. Symbol values produced by `Symbol.for` have specification identity, but not language identity.

Figure 2.5: Excerpt of Section 5.2.7 from the ECMAScript 2025 specification

```c
console.log(Object.is({}, {}));   // ✅ false (provenance-based differentiation)
console.log(Object.is(123, 123)); // ❌ true  (content-based differentiation)
```

Listing 2.3: Examples of calling `Object.is` in JavaScript

[4] `https://262.ecma-international.org/16.0/index.html#sec-object.is`
[5] `https://262.ecma-international.org/16.0/index.html#sec-identity`

2.3.2 Program interpretation using a CESK machine

> *In this section, we present an interpreter for a Scheme-like language written in Haskell. The goal is to understand why mainstream managed languages intentionally do not differentiate certain data types based on their origin.*

To illustrate the concept of value identity, we developed an interpreter written in Haskell for a toy language with customizable *allocation* systems. The goal is to explore why managed languages intentionally lack a suitable comparison operator for tracking the provenance of data and discuss the significant modifications that must be applied to an interpreter in order to provide such an ideal operator. If these changes were trivial, the motivation for this dissertation would be diminished. Our toy language can be seen as a core subset of Scheme, featuring three important deviations:

- The environment is immutable; thus, variable assignments are not possible, and there is no special form `(set! <var> <expr>)`.
- Garbage collection is not implemented, which means that the program's memory usage is monotonic and will never decrease during execution.
- Tail-call optimization is not implemented, resulting in linear memory consumption for iterations implemented via function recursion.

The complete interpreter consists of fewer than 2,000 lines of code and is available on GitHub[6]. Our interpreter implements a CESK machine [35], which is an abstract machine suitable for expressing the semantics of many managed programming languages [125]. In the remainder of this section, we highlight important aspects of our interpreter.

Expressions

Listing 2.4 defines the types of expressions that constitute our toy language. An expression can be one of the following:

- `Literal`: Creates a primitive value—e.g., `123`.
- `Variable`: Reads a value from the scope—e.g., `x`.
- `Condition`: Branches the control flow—e.g., `(if test cons alt)`.
- `Let`: Extends the scope—e.g., `(let (x right) body)`.
- `Lambda`: Creates a closure—e.g., `(lambda (param0 param1) body)`.
- `Application`: Calls a built-in function or a closure—e.g., `(fct arg0 arg1)`.

```haskell
data Expression
  = Literal Primitive
  | Variable Variable
  | Condition Expression Expression Expression
  | Let Variable Expression Expression
  | Lambda [Variable] Expression
  | Application Expression [Expression]
  deriving (Eq, Show)
```

Listing 2.4: Abstract grammar of our toy language

Data domain

Listing 2.5 defines the data domain of our interpreter. Note that it is parameterized by the type variable `v`, which represents values. This parameterization is necessary because our allocation systems require different value types. A data item can be one of the following:

[6] `https://github.com/lachrist/cesk-labo`

- `Primitive`: An atomic value.
- `Builtin`: A value defined by its name in the initial environment that can be applied.
- `Pair`: A pair comprising two values.
- `Closure`: A value created by a lambda expression, characterized by its environment, parameters, and body.

```haskell
data Primitive
  = Null
  | Boolean Bool
  | Number Double
  | String String
  deriving (Eq, Show)

data Domain v
  = Primitive Primitive
  | Builtin BuiltinName
  | Pair v v
  | Closure (Environment v) [Variable] Expression
  deriving (Eq, Show)

```

Listing 2.5: Data domain of our toy language

State space

Listing 2.6 defines the state space of our interpreter, which is parameterized by two type variables: `d` for the items in the store and `v` for the values in the environment and continuation. As previously mentioned, our interpreter operates as a CESK machine, meaning its states are composed of the following:

- Control: Represents the current expression.
- Environment: Mapping from variables to values.
- Store: Mapping from addresses to items[7].
- K(C)ontinuation: Represents what remains to be done after computing a value; this is analogous to the value stack in many virtual machines.

```haskell
data Continuation v
  = Bind (Environment v) Variable Expression (Continuation v)
  | Apply (Environment v) [Expression] [v] (Continuation v) Location
  | Branch (Environment v) Expression Expression (Continuation v)
  | Finish
  deriving (Eq, Show)

data State d v
  = Ongoing Expression (Environment v) (Store d) (Continuation v)
  | Success (Store d) v
  | Failure (Store d) String
  deriving (Eq, Show)
```

Listing 2.6: The state space of our toy interpreter (CESK machine)

Allocation system

Listing 2.7 defines the `Alloc` type class, which specifies how the interpreter interacts with values (i.e., `v`) that are opaque representations, items in the store (i.e., `d`) which are also opaque, and elements from the data domain (i.e., `Domain v`) which, in contrast, can be inspected and created. It features three methods:

- `new`: Creates a new value from a data item, which may involve updating the store.

[7] In our toy interpreter, the store is technically immutable and is entirely copied at every mutation, which is extremely inefficient.

- `get`: Retrieves a data item from a value.
- `set`: Either mutates a value by updating the store or fails with a message.

```haskell
class Alloc d v where
  new :: Store d -> Domain v -> (Store d, v)
  get :: Store d -> v -> Domain v
  set :: Store d -> v -> Domain v -> Either Message (Store d)
```

Listing 2.7: Allocation system required by our interpreter

Listing 2.8 exemplifies how our interpreter utilizes the allocation system by depicting the implementation of the `cons` and `set-car!` built-in functions. The `cons` built-in function creates a new pair by invoking `new`. While the `set-car!` built-in function mutates the first element of a pair in place, it first invokes `get` to access the second element of the pair, followed by invoking `set` to update the store. Note that this last operation may fail if mutations are not permitted within the current allocation system.

```haskell
applyBuiltin :: (Alloc d v) => Builtin -> [v] -> Store d -> Either Msg (Store d, v)
applyBuiltin (Builtin "cons") [car, cdr] store = Right $ new store (Pair car cdr)
appplyBuiltin (Builtin "set-car!") [cons, car] store = case get store cons of
  (Pair _ cdr) -> (cons, ) <$> set cons (Pair car cdr)
  _  -> Left "First argument of set-car! is not a pair"
```

Listing 2.8: Implementation of the `cons` and `set-car!` built-in functions

Our toy interpreter has 13 call sites for `new`, 14 call sites for `get`, and 7 call sites for `set`. This should suffice to demonstrate that the allocation system is a deeply embedded component of interpreters and cannot be easily patched. If it were, a portable tool performing provenance-aware analysis could be developed by patching a wide range of virtual machines. However, as it stands, developing and maintaining such a tool would require significant engineering effort due to the complexity and rapidly evolving nature of virtual machines.

Identity comparison and structural equality

Listing 2.9 presents the implementation of the `eq?` and `equal?` built-in functions, which define identity comparison and structural equality, respectively. Identity comparison does not resolve indirection and relies on the allocation system to implement the code `(==)` from the `Eq` type class for the value data type. It corresponds to the "is" axiomatic comparison defined in the ECMAScript 2025 specification, as referenced in Quotation 2.5. In contrast, structural equality resolves indirection to compare the underlying data of the values being compared.

```haskell
applyBuiltin :: (Eq v, Alloc d v) => Builtin -> [v] -> Store d -> Either Msg (Store d, v)
applyBuiltin (Builtin "eq?") [x, y] store = new store (Boolean $ x == y)
applyBuiltin (Builtin "equal?") [x, y] store = new store (Boolean $ structEq (x, y) store)

structEq :: (Eq v, Alloc d v) => (v, v) -> Store d -> Bool
structEq (x, y) store = case (get store x, get store y) of
  (Pair x1 x2, Pair y1 y2) -> structEq (x1, y1) store && structEq (x2, y2) store
  (Primitive p, Primitive q) -> p == q
  _ -> x == y
```

Listing 2.9: Implementation of the `eq?` and `equal?` built-in functions

2.3.3 Case study: solving quadratic equations

> *In this section, we present a program that will serve as a case study to examine the impact of different allocation systems on value identity.*

Listing 2.10 presents our case study program, which solves quadratic equations using the well-known formula from Equation 2.1. However, the program does not always behave in a user-friendly manner: in certain cases, it produces `NaN` values.

$$\forall\ a, b, c \in \mathbb{R} : \forall\ x \in \mathbb{R} :\quad a \cdot x^2 + b \cdot x + c = 0 \quad \Leftrightarrow \quad x = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a} \tag{2.1}$$

```
; delta.scm
(let solve
  (lambda (a b c)
    (let d (sqrt (- (expt b 2) (* 4 (* a c))))
      (cons
        (/ (- (- b) d) (* 2 a))
        (/ (+ (- b) d) (* 2 a)))))
  (let prompt
    (lambda (n)
      (begin
        (display (string-append "Enter a number for '" n "'...\n"))
        (string->number (read-line))))
    (let a (prompt "a")
      (let b (prompt "b")
        (let c (prompt "c")
          (let p (solve a b c)
            (display (string-append
              "Sol1 = " (number->string (car p))
              ", "
              "Sol2 = " (number->string (cdr p))
              "\n")))))))))
```

↓ ↓ ↓

```
> ./cesk-labo naive-alloc delta.scm
Enter a number for 'a'...
1
Enter a number for 'b'...
-7
Enter a number for 'c'...
10
Sol1 = 2.0, Sol2 = 5.0 ✅ Intended interaction
```

```
> ./cesk-labo naive-alloc delta.scm
Enter a number for 'a'...
1
Enter a number for 'b'...
2
Enter a number for 'c'...
3
Sol1 = NaN, Sol2 = NaN ❌ Not user-friendly
```

Listing 2.10: A program written in our toy language for solving quadratic equations

To diagnose the root cause of the appearance of `NaN` values, we manually insert calls to `trace`, a built-in function of our language that executes a call while logging both its arguments and result. We will examine the resulting traces in various allocation systems in the following sections. The usage of `trace` is illustrated in Listing 2.11.

```
(trace + 2 3 "@1:1")
```

↓ ↓ ↓

```
5.0 <- (+  2.0  3.0)  @1:1
```

Listing 2.11: Usage of `trace` to log an application of the built-in addition operator

2.3.4 Naive allocation system

> *In this section, we examine the identity of run-time values within a first allocation system that naively stores the entire data domain in new locations.*

Our naive allocation system is defined in Listing 2.12 as an instance of the `Alloc` type class. In this system, all values within the environment and the continuation are references pointing to locations in the store. The listing reads as follows:

- Comparing values determines whether they refer to the same location in the store.

- Inspecting a value reveals the location in the store, preceded by the character `&`.
- The `new` method stores the data item in a new location in the store and returns a value that references this location.
- The `get` method retrieves the data item referenced by the value.
- The `set` method always succeeds in updating the store at the location referenced by the value.

```
1
2  newtype StoreItem = StoreItem (Domain Value)
3
4  newtype Value = ReferenceValue Int
5
6  instance Eq Value where
7    (ReferenceValue addr1) == (ReferenceValue addr2) = addr1 == addr2
8
9  instance Show Value where
10   show (ReferenceValue addr) = "&" ++ show addr
11
12 instance Alloc StoreItem Value where
13   new store item =
14     let (store', addr) = push store (StoreItem item)
15      in (store', ReferenceValue addr)
16   get store (ReferenceValue addr) =
17     let StoreItem item = load store addr
18      in item
19   set store (ReferenceValue addr) item =
20     Right $ save store (addr, StoreItem item)
```

Listing 2.12: Naive allocation system without optimizations

Listing 2.13 exemplifies an interaction with the naive allocation system. The structural equality provided by `equal?` functions as expected, allowing differentiation based on the content of the data underlying the value. In contrast, the identity comparison offered by `eq?` is more surprising, enabling differentiation based on the provenance of the value. Inspecting values using the `inspect` built-in, which internally calls `show` from the `Show` type class, demonstrates how this differentiation is achieved, as each newly created number is stored in a fresh location.

```
1 (equal? 123 123)      ; ✅ #t    (same content)
2 (equal? 123 456)      ; ✅ #f    (different content)
3 (eq? 123 123)         ; ✅ #f    (different provenance)
4 (let x 123 (eq? x x)) ; ✅ #t    (same provenance)
5 (inspect 123)         ; ✅ "&38" (unique identity)
6 (inspect 123)         ; ✅ "&39" (unique identity)
```

Listing 2.13: Comparisons and inspections in the naive allocation system

Note that addresses will always increase during execution. This occurs because our interpreter lacks garbage collection, ensuring that each store location is used for only one value throughout the program's execution. In practice, garbage collection may lead to the reuse of addresses, preventing them from serving as unique identifiers for values. This issue can be addressed by disabling garbage collection, which can be achieved at the language level by maintaining a collection of values whose addresses should be protected. However, this approach can lead to memory leaks and may result in out-of-memory issues when analyzing long executions.

Listing 2.14 presents the trace of our quadratic equation solver program when executed with the inputs `2`, `3`, and `2` in this naive allocation system. Aside from user interactions highlighted in blue, each line in the trace corresponds to a built-in application. To diagnose the occurrence of `NaN`, we can use this trace to create a *value flow graph*. In a value flow graph, each node represents a value, while each edge represents a dependency, which can be labeled by the role the upstream value played in generating the downstream value. For example, value `&74` is used as the left-hand side of a division to produce value `&77`.

Figure 2.6 illustrates the value flow graph defined by Listing 2.14. The occurrence of `NaN` can be easily diagnosed by tracing the dependency links back to the source, which is the computation of the square root of a negative number. It is now clear that the program can be improved by adding a case clause to handle situations where a negative value is assigned to `d`. Our example demonstrates that the naive allocation system enables straightforward instrumentation to accurately track value provenance.

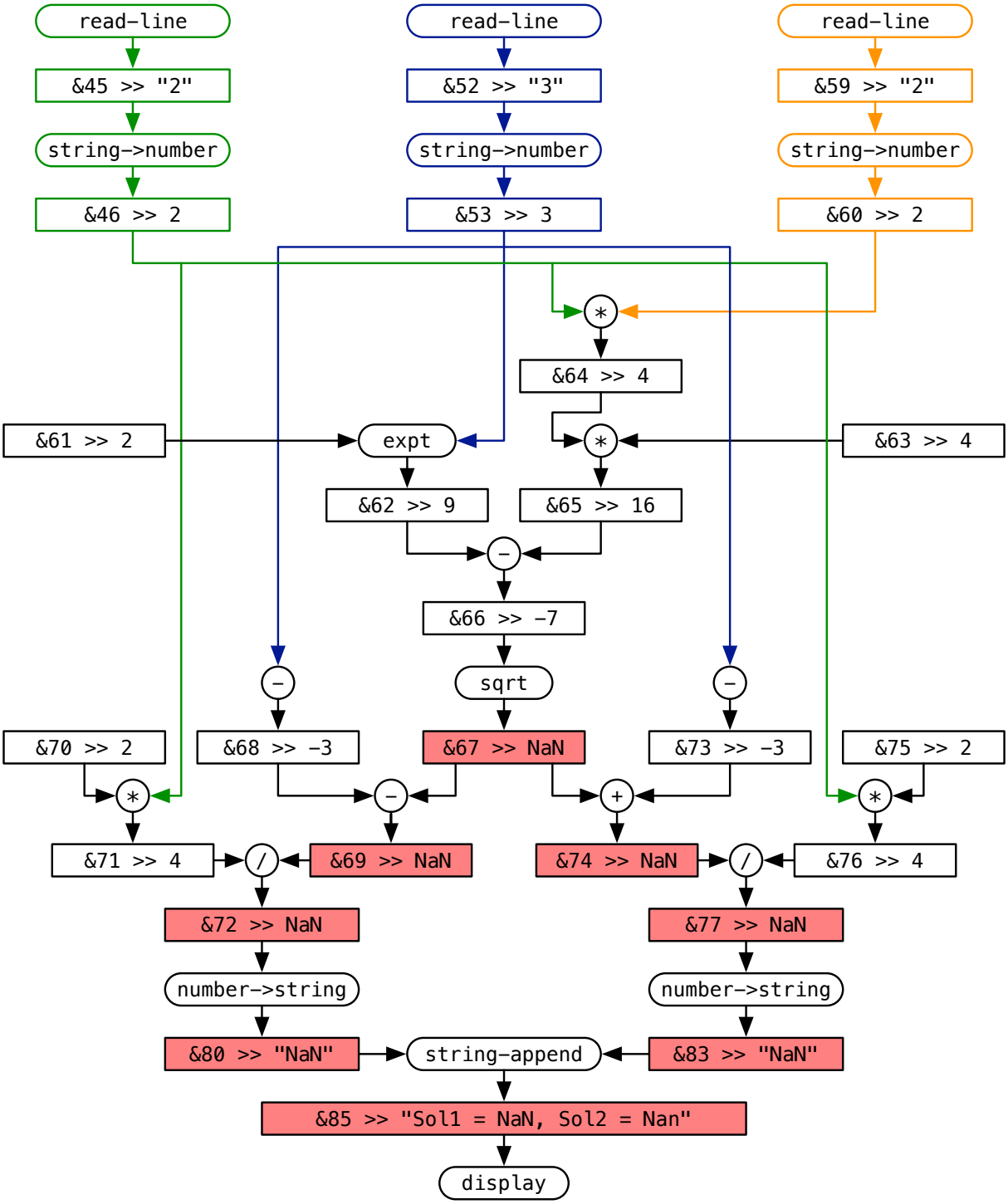


Figure 2.6: Visualization of the value flow graph defined by the trace in Listing 2.14

```
> ./cesk-labo naive-alloc delta-trace.scm
Enter a number for 'a'...
2
&45 <- (read-line)  @11:31
&46 <- (string->number  &45)  @11:9
Enter a number for 'b'...
3
&52 <- (read-line)  @11:31
&53 <- (string->number  &52)  @11:9
Enter a number for 'c'...
2
&59 <- (read-line)  @11:31
&60 <- (string->number  &59)  @11:9
&62 <- (expt  &53  &61)  @3:33
&64 <- (*  &46  &60)  @3:61
&65 <- (*  &63  &64)  @3:50
&66 <- (-  &62  &65)  @3:24
&67 <- (sqrt  &66)  @3:12
&68 <- (-  &53)  @5:27
&69 <- (-  &68  &67)  @5:18
&71 <- (*  &70  &46)  @5:42
&72 <- (/  &69  &71)  @5:9
&73 <- (-  &53)  @6:27
&74 <- (+  &73  &67)  @6:18
&76 <- (*  &75  &46)  @6:42
&77 <- (/  &74  &76)  @6:9
&72 <- (car  &78)  @17:47
&80 <- (number->string  &72)  @17:25
&77 <- (cdr  &78)  @19:47
&83 <- (number->string  &77)  @19:25
&85 <- (string-append  &79  &80  &81  &82  &83  &84)  @16:28
Sol1 = NaN, Sol2 = NaN
&86 <- (display  &85)  @16:13
```

Listing 2.14: Execution of the tracing variant of Listing 2.10 in the naive allocation system

However, as illustrated in Quotation 2.5, mainstream managed languages define value identity in a way that allows optimizing how certain data types are allocated. While such optimizations lead to significant improvements in speed and memory usage, they hinder provenance tracking. We will examine two of these optimizations in the next two sections.

2.3.5 Allocation optimization #1: data inlining

> *In this section, we examine the identity of run-time values within a second allocation system that implements an optimization involving inlining parts of the data domain into values, which effectively renders this subdomain immutable.*

The first allocation optimization that hinders provenance tracking involves *inlining* portions of the data domain directly into values. For instance, *pointer tagging* exploits the fact that some addresses are invalid, using these otherwise-unused bits to encode small amounts of data. This first optimization is illustrated in Listing 2.15 as a second instance of the `Alloc` type class. In this system, stored items consist of pairs, built-ins, and closures. A value is either an immediate primitive or a reference to a stored item. The listing reads as follows:

- Comparing values determines whether they inline the same primitive or if they refer to the same location in the store.
- Inspecting a value either returns the string representation of the inlined primitive or it reveals the location in the store.
- The `new` function returns an immediate if the data item is a primitive; otherwise, it pushes a new item onto the store and returns a reference to that item.
- The `get` method directly returns the wrapped primitive if the value is immediate; otherwise, it loads the item from the store and converts it.
- The `set` method returns an error message if the left value is immediate or the right data item is a primitive; otherwise, it mutates the reference value in the store to the converted item.

```haskell
data StoreItem
  = PairItem Value Value
  | BuiltinItem BuiltinName
  | ClosureItem (Environment Value) [Variable] Expression
  deriving (Eq, Show)

data Value
  = ImmediateValue Primitive
  | ReferenceValue Address

instance Eq Value where
  (ImmediateValue prim1) == (ImmediateValue prim2) = prim1 == prim2
  (ReferenceValue addr1) == (ReferenceValue addr2) = addr1 == addr2
  _ == _ = False

instance Show Value where
  (ImmediateValue prim) = show prim
  (ReferenceValue addr) = "&" ++ addr

toDomain :: StoreItem -> Domain Value -- boilerplate

toItem :: Domain Value -> StoreItem -- boilerplate

instance Alloc StoreItem Value where
  -- new --
  new store (Primitive primitive) = (store, ImmediateValue primitive)
  new store datum =
    let (store', address) = push store (toItem datum)
     in (store', ReferenceValue address)
  -- get --
  get _ (ImmediateValue primitive) = Primitive primitive
  get store (ReferenceValue address) = toDomain $ load store address
  -- set --
  set _ (ImmediateValue _) _ = Left "Cannot mutate an immediate value"
  set _ _ (Primitive _) = Left "Reference value cannot be a primitive"
  set store (ReferenceValue address) datum = Right $ save store (address, toItem datum)
```

Listing 2.15: Allocation system with data inlining

Listing 2.16 describes a potential interaction enabled by the inline-optimized allocation system. Unfortunately, the identity comparison provided by `eq?` can no longer differentiate numeric values based on their provenance because the content of the data has been inlined into the value.

```scheme
(eq? 123 123) ; ❌ #t    (different provenance)
(inspect 123) ; ❌ "123" (no unique identity)
```

Listing 2.16: Comparisons and inspections in the inline-optimized allocation system

It should be clear that handling immediate values is more efficient than handling reference values, primarily because immediate values avoid the need for indirection in memory. This reduction in memory indirection not only speeds up data access but also reduces memory usage, as there is no need to allocate separate memory for objects and references. As a result, the overall memory footprint is smaller, which in turn alleviates pressure on the garbage collector.

Note that real-world interpreters typically design their values to fit within a CPU word size, such as 32 bits or 64 bits. This means that elements from the inlined data domain must be encodable in strictly less than 32 or 64 bits. For example, although strings are typically immutable in high-level programming languages, they cannot be immediate values due to their potentially arbitrary length. As shown in Listing 2.17, the reference interpreter of Ruby known as MRI inlines integers in the range $[-2^{62}, 2^{62} - 1]$ on a 64-bit architecture[8]. In contrast, integers outside of this range are represented as `Bignum` and are not inlined.

Another example of an interpreter with immediate values is SpiderMonkey, the JavaScript engine of Firefox. Listing 2.18 highlights parts of `Value.h`[9] that define how SpiderMonkey represents JavaScript values. A JavaScript value can be tested for `null` or `undefined` and can be converted to a `bool`, an `int32_t`, a `double`, a pointer to a `JS::BigInt`, a pointer to a `JS::String`, a pointer to a `JS::Symbol`, and a pointer to a `JS::Object`. Thus, the data types `undefined`, `null`, `boolean`, and `number` are represented as immediate values, whereas the data types `object`, `string`, `symbol`, and `bigint` are represented as reference values.

[8] https://ruby-doc.org/core-1.8.6/Fixnum.html

[9] `https://hg.mozilla.org/mozilla-central/file/FIREFOX_NIGHTLY_68_END/js/public/Value.h`

```ruby
# Ruby 2.3.7p456

puts 123.object_id # ❌ 247
puts 123.object_id # ❌ 247

FIXNUM_MAX = (2**(0.size * 8 -2) -1)

puts FIXNUM_MAX.class       # Fixnum
puts (FIXNUM_MAX + 1).class # Bignum

puts FIXNUM_MAX.object_id # ❌ 9223372036854775807
puts FIXNUM_MAX.object_id # ❌ 9223372036854775807

puts (FIXNUM_MAX + 1).object_id # ✅ 70131054129240
puts (FIXNUM_MAX + 1).object_id # ✅ 70131054129140
```

Listing 2.17: Inlining of `Fixnum` values in MRI (Ruby)

This is made possible by a technique called *NaN-boxing*, which is based on the encoding of floating-point numbers in double precision.

```cpp
# define JSVAL_TAG_SHIFT 47                                                    //  39
union Value {                                                                  // 328
  private:                                                                     // 329
    uint64_t asBits_;                                                          // 330
    double asDouble_;                                                          // 331
  public:                                                                      // 547
    bool isUndefined() const {                                                 // 570
      return asBits_ == JSVAL_SHIFTED_TAG_UNDEFINED;                           // 574
    }                                                                          // 586
    bool isNull() const {                                                      // 578
      return asBits_ == JSVAL_SHIFTED_TAG_NULL;                                // 582
    }                                                                          // 584
    int32_t toInt32() const {                                                  // 703
      MOZ_ASSERT(isInt32());                                                   // 704
      return int32_t(asBits_);                                                 // 708
    }                                                                          // 710
    double toDouble() const {                                                  // 712
      MOZ_ASSERT(isDouble());                                                  // 713
      return asDouble_;                                                        // 714
    }                                                                          // 715
    JSString* toString() const {                                               // 722
      MOZ_ASSERT(isString());                                                  // 723
      return reinterpret_cast<JSString*>  (asBits_ ^ JSVAL_SHIFTED_TAG_STRING);  // 727
    }                                                                          // 729
    JS::Symbol* toSymbol() const {                                             // 731
      MOZ_ASSERT(isSymbol());                                                  // 732
      return reinterpret_cast<JS::Symbol*> (asBits_ ^ JSVAL_SHIFTED_TAG_SYMBOL); // 736
    }                                                                          // 738
    JS::BigInt* toBigInt() const {                                             // 740
      MOZ_ASSERT(isBigInt());                                                  // 741
      return reinterpret_cast<JS::BigInt*> (asBits_ ^ JSVAL_SHIFTED_TAG_BIGINT); // 745
    }                                                                          // 747
    JSObject* toObjectOrNull() const {                                         // 761
      MOZ_ASSERT(isObjectOrNull());                                            // 762
      return reinterpret_cast<JSObject*> (asBits_ ^ JSVAL_SHIFTED_TAG_OBJECT);   // 771
    }                                                                          // 773
    bool toBoolean() const {                                                   // 788
      MOZ_ASSERT(isBoolean());                                                 // 789
      return bool(int32_t(asBits_));                                           // 793
    }                                                                          // 795
}                                                                              // 893
```

Listing 2.18: Highlight of `Value.h` from Firefox 68 for a 64-bit architecture

Similarly to Listing 2.14, Listing 2.19 presents the trace of our quadratic equation solver program when executed with the inputs `2`, `3`, and `2` in the inline-optimized allocation system. Note that only pairs are still displayed as store addresses whereas all primitive data are directly displayed in the trace. Unfortunately, as shown in Figure 2.7, this trace results in a significantly collapsed value flow graph. Specifically, all `NaN` value nodes have been merged into a single node, making it difficult to diagnose the root cause of the error. Inlining primitive values such as `NaN` has complicated data provenance analysis, as it is unclear how to restore the value graph from Figure 2.6 from this lossy representation. Note that even though the program has no mutations, the collapse of value nodes has created many cycles.

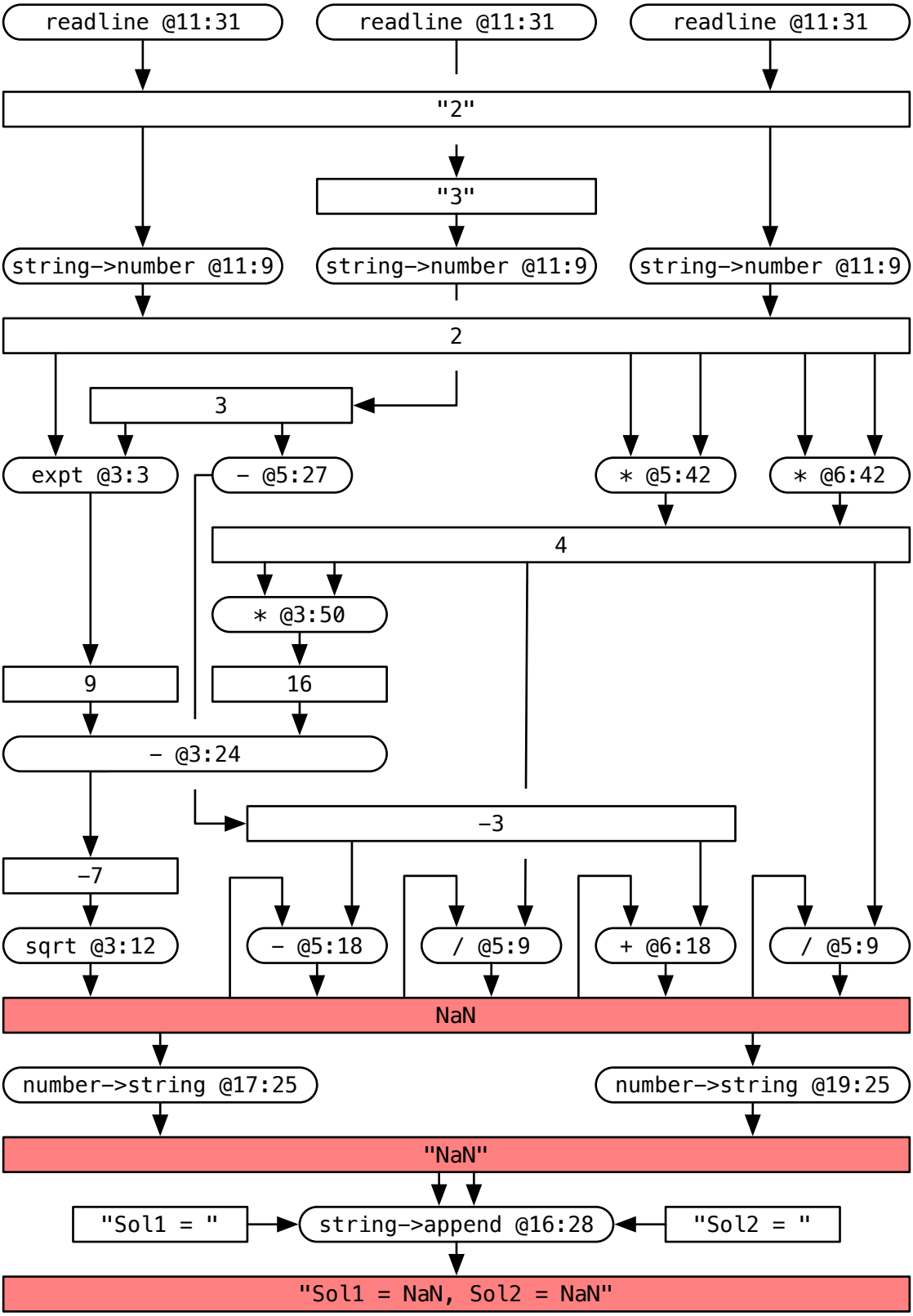


Figure 2.7: Visualization of the value flow graph defined by the trace in Listing 2.19

```
./cesk-labo hybrid-alloc delta-trace.scm
Enter a number for 'a'...
2
"2" <- (read-line)  @11:31
2.0 <- (string->number  "2")  @11:9
Enter a number for 'b'...
3
"3" <- (read-line)  @11:31
3.0 <- (string->number  "3")  @11:9
Enter a number for 'c'...
2
"2" <- (read-line)  @11:31
2.0 <- (string->number  "2")  @11:9
9.0 <- (expt  3.0  2.0)  @3:33
4.0 <- (*  2.0  2.0)  @3:61
16.0 <- (*  4.0  4.0)  @3:50
-7.0 <- (-  9.0  16.0)  @3:24
NaN <- (sqrt  -7.0)  @3:12
-3.0 <- (-  3.0)  @5:27
NaN <- (-  -3.0  NaN)  @5:18
4.0 <- (*  2.0  2.0)  @5:42
NaN <- (/  NaN  4.0)  @5:9
-3.0 <- (-  3.0)  @6:27
NaN <- (+  -3.0  NaN)  @6:18
4.0 <- (*  2.0  2.0)  @6:42
NaN <- (/  NaN  4.0)  @6:9
NaN <- (car  &41)  @17:47
"NaN" <- (number->string  NaN)  @17:25
NaN <- (cdr  &41)  @19:47
"NaN" <- (number->string  NaN)  @19:25
"Sol1 = NaN, Sol2 = NaN\n" <- (string-append
  "Sol1 = "  "NaN"  ", "
  "Sol2 = "  "NaN"  "\n"
)  @16:28
Sol1 = NaN, Sol2 = NaN
#n <- (display  "Sol1 = NaN, Sol2 = NaN\n")  @16:13
```

Listing 2.19: Execution of the tracing variant of Listing 2.10 in the allocation system with inlining

2.3.6 Allocation optimization #2: value interning

> *In this section, we examine the identity of run-time values within a third allocation system that employs an optimization technique known as interning. This technique involves indexing portions of the data domain in the store for reuse, which reduces memory consumption and accelerates comparison.*

To reduce memory consumption and improve comparison speed, mainstream managed languages often implement a second optimization known as *interning*. This technique involves indexing portions of the data domain in the store for reuse. Because interned data remains in the store, interning does not technically prevent mutations. However, in practice, mutations are prohibited on interned values because they would introduce impractical global dependencies. This second optimization is illustrated in Listing 2.20 as a third instance of the `Alloc` type class. It closely resembles our first allocation system from Listing 2.12, with the key difference being that it reuses store locations for primitive data.

Listing 2.21 depicts a potential interaction within the intern-optimized allocation system. Unfortunately, the identity comparison provided by `eq?` can no longer distinguish numeric values based on their provenance due to address reuse.

Compared to inlining, interning is slower due to memory indirection. However, it has the distinct advantage of being applicable to data types with arbitrarily long encodings, such as strings. This makes it a crucial optimization in JavaScript, where it improves property lookup by comparing interned property keys based on their addresses rather than their content. CPython is another interpreter that implements interning. As demonstrated in Listing 2.22, integers ranging from -5 to 256 are always interned[10], while literal integers have a larger interning range.

Listing 2.23 shows the result of executing the tracing variant of Listing 2.10 in the intern-optimized allocation, using the inputs `2`, `3`, and `2`. Although different in nature from data inlining, value interning

[10] `https://docs.python.org/release/3.7.4/c-api/long.html#c.PyLong_FromLong`

```haskell
newtype StoreItem = StoreItem (Domain Value) deriving Eq

newtype Value = ReferenceValue Address

instance Eq Value where
  (ReferenceValue addr1) == (ReferenceValue addr2) = addr1 == addr2

instance Show Value where
  show (ReferenceValue addr) = "&" ++ show addr

allocate datum store = (store', Value address)
  where (store', address) = push store (StoreItem datum)

instance Alloc StoreItem Value where
  new store datum@(Primitive _) = case find store (StoreItem datum) of
    Just address -> (store, Value address)
    Nothing -> allocate store datum
  new store datum = allocate store datum
  get store (Value address) = datum where (StoreItem datum) = load store address
  set store (Value address) datum = Right $ save store (address, StoreItem datum)
```

Listing 2.20: Allocation system with value interning

```scheme
(eq? 123 123) ; ❌ #t    (different provenance)
(inspect 123) ; ❌ "&38" (no unique identity)
(inspect 123) ; ❌ "&38" (no unique identity)
```

Listing 2.21: Comparisons and inspections in the intern-optimized allocation system

```python
# Pos. Integer #
i = 0
j = 0
while i is j:
  i += 1
  j += 1
print(i) # 257

# Neg. Integer #
i = 0
j = 0
while i is j:
  i -= 1
  j -= 1
print(i) # -6

# Literal Integer #
print(1000 is 1000)       # True
print(1000 is 500 + 500)  # True (due to constant folding)
print(1000 is pow(10, 3)) # False
```

Listing 2.22: Integer interning in Python 3.9.6

results in the same collapse of node values. Indeed, this trace yields a value flow graph similar to that depicted in Figure 2.7, with nodes represented as addresses rather than primitive data items.

```
cesk-labo reuse-alloc delta-trace.scm
Enter a number for 'a'...
2
&46 <- (read-line)  @11:31
&47 <- (string->number  &46)  @11:9
Enter a number for 'b'...
3
&50 <- (read-line)  @11:31
&51 <- (string->number  &50)  @11:9
Enter a number for 'c'...
2
&46 <- (read-line)  @11:31
&47 <- (string->number  &46)  @11:9
&54 <- (expt  &51  &47)  @3:33
&55 <- (*  &47  &47)  @3:61
&56 <- (*  &55  &55)  @3:50
&57 <- (-  &54  &56)  @3:24
&58 <- (sqrt  &57)  @3:12
&59 <- (-  &51)  @5:27
&60 <- (-  &59  &58)  @5:18
&55 <- (*  &47  &47)  @5:42
&61 <- (/  &60  &55)  @5:9
&59 <- (-  &51)  @6:27
&62 <- (+  &59  &58)  @6:18
&55 <- (*  &47  &47)  @6:42
&63 <- (/  &62  &55)  @6:9
&61 <- (car  &64)  @17:47
&66 <- (number->string  &61)  @17:25
&63 <- (cdr  &64)  @19:47
&66 <- (number->string  &63)  @19:25
&70 <- (string-append  &65  &66  &67  &68  &66  &69)  @16:28
Sol1 = NaN, Sol2 = NaN
&45 <- (display  &70)  @16:13
```

Listing 2.23: Execution of the tracing variant of Listing 2.10 in the allocation system with interning

2.3.7 Provenancial equality

> *In this section, we introduce an ideal comparison operator for differentiating run-time values based on their provenance.*

We observed in Section 2.3.4 that an allocation system which naively stores every data item provides a referential equality that gives rise to a value flow graph that is suitable for provenance-based analysis. We term this ideal comparison the *provenancial equality* operator. In the following, we present a formal framework to express various value comparisons and how they relate with one another.

Equation 2.2 defines the signature in a many-sorted logic of various comparison operators. Given an interpreter, the primary sorts are S, denoting its set of states, and V, denoting its set of values. Equality on V is defined as an equivalence relation. In addition, we assume the existence of two subsorts, V_{prim} and V_{ref}, which are subsets of values denoting primitive values and reference values, respectively. In this context, a value comparison is defined as a relation between two values parameterized by an interpreter state. This signature ensures that these value comparisons can be made available to the implemented language as built-in functions. Note that including the interpreter state in the input of comparison relations is crucial because addresses are meaningless without state to contextualize them.

$$
\begin{aligned}
&\text{Sorts:} && S, \quad V \\
&\text{Subsorts:} && V_{\mathsf{prim}} \subseteq V, \quad V_{\mathsf{ref}} \subseteq V \\
&\text{Predicate symbols:} && = \ \subseteq V \times V \\
& && \overset{\mathsf{prov}}{=\!=} \ \subseteq S \times V \times V \\
& && \overset{\mathsf{struct}}{=\!=} \ \subseteq S \times V \times V \\
& && \overset{\mathsf{ref}}{=\!=} \ \subseteq S \times V \times V
\end{aligned}
\tag{2.2}
$$

We assume that the interpreter implements the naive allocation system described in Section 2.3.4, which means that provenancial equality is determined by value identity, as indicated by Equation 2.3.

$$\mathsf{ProvEq} := \forall\ s \in S\ :\ \forall\ v_1, v_2 \in V :\quad v_1 \overset{\mathsf{prov}}{\underset{s}{=\!=}} v_2 \quad \Leftrightarrow \quad v_1 = v_2 \tag{2.3}$$

We do not provide a precise definition of structural equality, as it is language-dependent. However, it must adhere to the requirement of Equation 2.4, which indicates that two identical values must be structurally equal. Note that a value may be considered invalid for a given state (e.g., a missing address in the store). We can address this situation by defining structural equality to hold if either value is invalid.

$$\mathsf{StructEq} := \forall\ s \in S :\ \forall\ v_1, v_2 \in V :\quad v_1 \overset{\mathsf{struct}}{\underset{s}{=\!=}} v_2 \quad \Leftarrow \quad v_1 = v_2 \tag{2.4}$$

Referential equality is defined as a hybrid between provenancial equality and structural equality, as presented in Equation 2.5. If the compared values are both primitives, then referential equality defaults to structural equality. If the compared values are both references, referential equality defaults to provenancial equality. Otherwise, the compared values do not share the same types and are not referentially equal. This distinction between reference values and primitive values applies to many managed languages, with some exceptions. For instance, in Python, integers outside the range $[-5, 256]$ are regarded as primitive values, yet they are compared using provenancial equality.

$$\mathsf{RefEq} := \forall\ s \in S :\ \forall\ v_1, v_2 \in V :\quad v_1 \overset{\mathsf{ref}}{\underset{s}{=\!=}} v_2 \quad \Leftrightarrow \quad \vee \begin{cases} v_1 \in V_{\mathsf{prim}} \wedge v_2 \in V_{\mathsf{prim}} \wedge v_1 \overset{\mathsf{struct}}{\underset{s}{=\!=}} v_2 \\ v_1 \in V_{\mathsf{ref}} \wedge v_2 \in V_{\mathsf{ref}} \wedge v_1 \overset{\mathsf{prov}}{\underset{s}{=\!=}} v_2 \end{cases} \tag{2.5}$$

From Equations 2.3, 2.4, and 2.5, we directly derive the strictness relation from Equation 2.6: provenancial equality is a stricter relation than referential equality, which in turn is stricter than structural equality. In a sense, provenancial equality can be viewed as an extension of referential equality to primitive value types.

$$\mathsf{Strictness} := \forall\ s \in S : \forall\ v_1, v_2 \in V :\quad v_1 \overset{\mathsf{prov}}{\underset{s}{=\!=}} v_2 \quad \Rightarrow \quad v_1 \overset{\mathsf{ref}}{\underset{s}{=\!=}} v_2 \quad \Rightarrow \quad v_1 \overset{\mathsf{struct}}{\underset{s}{=\!=}} v_2 \tag{2.6}$$

However, to enable language runtimes to inline small data items and intern values for reuse, the semantics of most managed languages intentionally do not define provenancial equality. To retain our formalism and define provenancial equality precisely, the simplest approach is to model mainstream interpreters as if they implement a naive allocation system, without exposing provenancial equality to the guest language. Concealing value identity from the guest language ensures that the allocation systems discussed in this section have no observable effect beyond performance. In this sense, provenancial equality represents an ideal value comparison that, while inaccessible to the guest language, remains formally definable [11].

As stated in Section 2.2.3, we focus on code instrumentation, meaning that the interpreter states and values cannot be altered. However, it is possible to conduct instrumentation such that the mechanism for approximate provenancial equality is applied only to a subset of values, without this restriction being noticeable in the original program. We explore this approach in Chapter 5. Equation 2.7 defines additional signatures that complement those in Equation 2.2. It introduces the subsort $\tilde{V}$, which denotes a subset of values upon which the approximation of provenancial equality $\overset{\mathsf{prov}}{\approx\!\approx}$ is defined. The function α allows for extracting the general value in V that is represented by the restricted value in $\tilde{V}$.

[11] An alternative approach that would more closely describe reality would be based on a bisimulation relation between an interpreter with naive allocations that provides exact provenance equality and one with optimized allocations and instrumentation that approximates provenance equality. However, this is technically challenging for three main reasons. First, to accommodate instrumentation, the bisimulation must be stuttering invariants. Second, to translate values between executions, the bisimulation must also align values and be of type: $BSim \subseteq S \times V \times \tilde{S} \times \tilde{V}$. Third, behavioral divergences should be allowed when provenancially comparing values, as the approximation may yield different results than the exact relation.

$$\begin{array}{ll} \text{Subsorts:} & \tilde{V} \subseteq V \\ \text{Function symbols:} & \alpha : \tilde{V} \to V \\ \text{Predicate symbols:} & \overset{\mathsf{prov}}{\approx\!\approx} \;\subseteq\; S \times \tilde{V} \times \tilde{V} \end{array} \tag{2.7}$$

We now have all the tools necessary to define the soundness and completeness of an approximation of provenancial equality. Equation 2.8 states that a *sound* approximation of provenancial equality is satisfied only if the values represented by the operands are indeed provenancially equal. Conversely, Equation 2.9 states that a *complete* approximation of provenancial equality holds whenever the values represented by the operands are indeed provenancially equal.

$$\mathsf{SoundProvEq} \;:=\; \forall\, s \in S :\; \forall\, \tilde{v}_1, \tilde{v}_2 \in \tilde{V} :\quad \tilde{v}_1 \overset{\mathsf{prov}}{\underset{s}{\approx\!\approx}} \tilde{v}_2 \quad\Rightarrow\quad \alpha(\tilde{v}_1) \overset{\mathsf{prov}}{\underset{s}{=\!=}} \alpha(\tilde{v}_2) \tag{2.8}$$

$$\mathsf{ComplProvEq} \;:=\; \forall\, s \in S :\; \forall\, \tilde{v}_1, \tilde{v}_2 \in \tilde{V} :\quad \tilde{v}_1 \overset{\mathsf{prov}}{\underset{s}{\approx\!\approx}} \tilde{v}_2 \quad\Leftarrow\quad \alpha(\tilde{v}_1) \overset{\mathsf{prov}}{\underset{s}{=\!=}} \alpha(\tilde{v}_2) \tag{2.9}$$

As noted in Equation 2.6, referential equality is a weaker relation than provenancial equality. Therefore, if we set $\tilde{V}$ to V and α to the identity function, referential equality provides a straightforward complete approximation of provenancial equality without requiring code instrumentation.

On a related note, since we assume that referential equality is accessible to the guest language, it should be straightforward to provide a complete approximation of provenancial equality for references. This is expressed in Equation 2.10, which states that a referentially complete approximation of provenancial equality will only be unsatisfied if the values represented by the operands are not referentially equal. This condition is weaker than the full completeness requirement stated in Equation 2.9.

$$\mathsf{RefComplProvEq} \;:=\; \forall\, s \in S :\; \forall\, \tilde{v}_1, \tilde{v}_2 \in \tilde{V} :\quad \tilde{v}_1 \overset{\mathsf{prov}}{\underset{s}{\approx\!\approx}} \tilde{v}_2 \quad\Leftarrow\quad \alpha(\tilde{v}_1) \overset{\mathsf{ref}}{\underset{s}{=\!=}} \alpha(\tilde{v}_2) \tag{2.10}$$

In the remainder of this dissertation, particularly in Chapters 5 and 6, we examine how to achieve a sound and referentially complete approximation of provenancial equality through point-based code instrumentation.

2.4 Conclusion

In this chapter, we provided background information on the extensive field of dynamic program analysis. We then outlined the primary problem statement of this dissertation: tracking data provenance in managed programming languages. We conclude by reiterating the scope of the dissertation and its problem statement, presenting key evaluation criteria, and highlighting the main contributions of the chapter.

Scope of the dissertation

Throughout this chapter, we progressively refined the scope of this dissertation in five successive steps:

1. In Section 2.1.2, we focused on dynamic program analysis, which provides a precise, though narrow, view of program execution. Later, in Section 2.2.1, we noted that dynamic analysis techniques often require instrumentation, which must remain **transparent** to preserve the primary advantage of dynamic program analysis: enabling the study of exact under-approximations of program behavior.
2. In Section 2.1.4, we identified a representative sample of dynamic analysis techniques against which the **expressiveness** and the **cross-analysis applicability** of an approach for designing dynamic

analysis can be evaluated. Of particular interest are analyses that rely on the provenance of values, which can be challenging to acquire **accurately** for immutable values.

3. In Section 2.2.2, we narrowed our focus to analyzing program execution at the source code level. This abstraction level offers two main advantages that should be preserved. First, it has the potential to be **portable** across various runtimes and hardware. Second, it provides insights that are **adjacent** to source code, which benefit analyses focused on program comprehension.

4. In Section 2.2.3, we narrowed our focus to point-based instrumentation, which relies on extensive code transformation and the inherent ability of programs to observe their own behavior. To retain advantages over alternative methods, a point-based approach to designing dynamic analysis should make minimal assumptions about both the reflective capabilities of the language and the runtime, ensuring that it remains **applicable** across different languages and runtimes.

5. In Section 2.3.1, we narrowed our focus on analyzing the execution of programs written in managed languages that enforce value abstraction. This restriction is important for maintaining a clear understanding of values and their provenance. Additionally, as demonstrated in Chapter 5, value abstraction enables direct support for **selective** instrumentation, which mitigates the primary weaknesses of our approach: run-time overhead.

To summarize, this chapter narrows the focus of this dissertation to an approach for conducting dynamic analysis at the source code level of programs written in managed languages through point-based instrumentation, emphasizing support for analyses that require information about the provenance of values.

Evaluation criteria

We now present the key evaluation criteria for such an approach, most of which have been previously introduced throughout the chapter:

- Applicability: The range of scenarios to which our approach can be applied.

  The value of our approach depends on its ability to support a wide range of scenarios. Its applicability is shaped by the constraints it imposes on the kinds of analyses it can implement, the programming languages it can target, and the runtime environments it can operate in.

  - Cross-Analysis Applicability: The range of dynamic program analysis techniques that can be implemented with our approach (p. 29).

    In Section 2.1.4, we introduced several representative dynamic program analysis techniques to evaluate the capacity of any general-purpose approach to represent a diverse range of dynamic analyses. Specifically, our approach focuses on dynamic analysis techniques that require provenance information that is extrinsic to program execution and must be managed by the analysis. Important examples of these techniques include dynamic type checking, dynamic symbolic execution, dynamic taint analysis, and dynamic program slicing.

  - Cross-Platform Applicability: The requirements imposed by our approach on the execution environments in which it can operate (p. 35).

    In Section 2.2.3, we discussed that the primary advantage of program instrumentation over runtime instrumentation is its decoupling from the execution environment. To remain competitive, our approach must preserve this strength, which means it is expected to impose minimal constraints on the execution environment.

  - Cross-Language Applicability: The requirements imposed by our approach on the programming languages it can target (p. 35).

    In Section 2.2.3, we discussed that the primary advantage of point-based program instrumentation over hook-based program instrumentation is its ability to overcome the limitations of the reflection mechanisms of the target language. To remain competitive, our approach should not depend on the presence of reflection mechanisms that could eliminate the necessity for point-based instrumentation entirely.

- Expressiveness: The extent to which our approach supports the specification of dynamic program analysis in a clear and concise manner.

Applicability is meaningless without expressiveness; if our approach lacks robust abstractions, it will fail to address any actual specification issues experienced by analysis implementers. The primary goal of this dissertation is to provide these abstractions while preserving applicability.

  - Cohesion: The extent to which our approach exposes an interface with a small number of clear and well-defined entry points.

    An important requirement for expressiveness is to offer a concise and simple interface for defining the analysis logic. This interface should feature a limited number of entry points to minimize the cognitive load on the analysis implementer, with each entry point having clear semantics to ensure that the analysis specification is easily understandable.

  - Support for Extrinsic Information: The extent to which our approach supports maintaining information extrinsic to the baseline program execution.

    In Section 2.1.4, we observed that significant dynamic program analysis techniques, such as dynamic model checking and provenance-aware dynamic analyses, require information that cannot be directly obtained from the run-time states of the target program during its baseline executions. Instead, this information must be maintained in a separate layer during the augmented execution of the target program. Our approach should facilitate this process.

  - Support for Distribution: The extent to which our approach supports the holistic analysis of distributed applications.

    Modern applications are frequently distributed. Analyzing these applications comprehensively necessitates orchestrating the analysis of their individual components. In other words, reasoning about the shared state of a distributed application requires an analysis that is itself distributed. Given that an analysis may involve complex logic, adapting it to a distributed context can be challenging and error-prone. Our approach should facilitate this process.

- Precision: The extent to which our approach supports generating precise insights about the runtime behavior of the program under analysis.

  The value of dynamic program analysis is contingent upon its ability to provide a narrow but precise view of program behavior. This strength must be preserved, which entails that our approach should be both *transparent* and *accurate*.

  - Transparency: The extent to which our approach supports non-interference with the execution of the program under analysis (p. 32).

    In Section 2.2.1, we discussed how a lack of transparency can cause an analysis tool to create discrepancies, which are adverse situations where the analysis examines a program trace that cannot occur during normal execution. Transparency is crucial because the strength of dynamic analysis lies in its ability to always under-approximate program behaviors, thereby avoiding false alarms when evaluating safety properties.

  - Accuracy: The extent to which our approach supports maintaining accurate extrinsic information (p. 31).

    As demonstrated in Figure 2.2, data provenance is a crucial piece of extrinsic information for many dynamic analyses. In Section 2.3, we discussed how two significant optimizations (i.e., data inlining and value interning) complicate the retrieval of precise information about data provenance. To ensure precision, our approach must provide an accurate approximation of provenancial equality.

- Interpretability: The extent to which our approach supports producing insights that are both understandable and actionable.

  Program analysis is valuable only when its insights can be effectively interpreted to enhance program comprehension and facilitate maintenance. Because of how generic our approach is, this responsibility primarily lies with the analysis implementer. However, we identified two important aspects of interpretability that remain within the scope of our approach: *source-adjacency* and *selectivity*.

  - Source-Adjacency: The extent to which our approach supports producing insights that can be traced back to the source code (p. 33).

    In Section 2.2.2, we discussed how analyzing programs at the source level yields insights that are closely linked to the source code. This connection allows programmers to effectively utilize these insights to understand and potentially modify the target program. If our approach fails to provide source-adjacent insights, it could be argued that it does not offer any advantages

over analyzing lower-level representations of programs, which benefit from being independent of the programming language used for the source code.

  - Selectivity: The support of our approach to selectively analyze programs.

    Selective analysis is an important aspect of interpretability, as it enables the generation of insights that focus on parts of the program known to be of interest prior to the analysis. Point-based instrumentation, by virtue of being local, offers a built-in preliminary filtering process that can be leveraged to implement static selectivity. Our approach should preserve this strength.

- Practicality: The engineering effort required to implement our approach across its range of applicable scenarios.

  The engineering effort required to develop a robust, general-purpose platform for dynamic program analysis should not be underestimated. Therefore, the practicality of our approach hinges on the engineering efforts necessary to implement it across its various application scenarios.

  - Implementability: The simplicity of implementing our approach for a specific language and platform.

    For the approach to be practical, it should be reasonably straightforward to implement. In particular, it is important to address the challenges of analyzing large and complex applications written in mainstream programming languages.

  - Portability: The effort necessary to port an existing implementation of our approach to other languages and platforms (p. 34).

    Dynamic program analysis tools employing point-based program instrumentation at the source code level can be largely agnostic to the execution platform, which typically includes the language runtime, operating system, and hardware. Our approach should support the implementation of such portable tools. If it does not, it may provide no advantages over instrumenting the language runtime, which has the distinct benefit of overcoming language barriers—a significant limitation for evaluating data provenance.

Problem statement

Section 2.3.4 demonstrated that conducting provenance-aware analysis in a minimalist managed language with naive allocation can be as simple as logging function applications. The problem statement of this dissertation consists of restoring this simplicity in real-world scenarios. In this regard, we identified three independent challenges:

- To enhance expressiveness, real-world managed languages feature many complex syntactic constructs, such as classes and iterables, which must be carefully considered during point-based source code instrumentation. Chapters 3 and 4 address this challenge.
- To optimize performance, real-world managed languages implement optimizations such as value inlining and interning, which hinder the tracking of value provenance. Chapters 5 and 6 address this challenge.
- Applications written in managed languages are often distributed, necessitating that the analysis also be distributed to reason about the state of the application as a whole; this requires significant engineering to orchestrate the various parts of the analysis. Chapter 7 addresses this challenge.

Main contributions

The problem of attaching metadata to the data manipulated by the program under analysis is well-studied. Traditionally, this issue is framed in a location-centric manner, where metadata is attached to locations within the run-time state of the target program [72, 83, 138, 14, 105]. This framing directs the solution toward shadow execution, which involves maintaining a structure that contains metadata mirroring the run-time state of the target program. In this chapter, by introducing provenancial equality, we framed the problem in a value-centric manner that decouples provenance tracking from metadata

tagging. The remainder of this dissertation demonstrates that for languages enforcing value abstraction, such a framing facilitates clearer separations of concerns within the analysis implementation.

Chapter 3

AranLang: a core variant of JavaScript

In this chapter, we introduce AranLang, a core variant of JavaScript designed to simplify the development of dynamic program analysis tools. This chapter is organized as follows:

- *Section 3.1 introduces the need to insert AranLang into the instrumentation pipeline.*
- *Section 3.2 presents the syntax of AranLang and highlights several points of interest in the language.*
- *Section 3.3 describes how AranLang programs can be retropiled to JavaScript.*
- *Section 3.4 describes how JavaScript programs can be transpiled to AranLang.*
- *Section 3.5 concludes the chapter by summarizing the results and discussing related work.*

3.1 Introduction to AranLang

In this section, we introduce AranLang, a minimal variant of JavaScript. This section is organized as follows:

- *Section 3.1.1 discusses why JavaScript is a suitable language for presenting our approach.*
- *Section 3.1.2 outlines the rationale for introducing AranLang.*
- *Section 3.1.3 provides a motivating example for introducing AranLang.*

3.1.1 Selecting JavaScript as the target language

In this section, we justify our choice of JavaScript as both the implementation and target language for presenting our approach.

In the previous chapter, we provided a general overview of the challenges associated with conducting provenance-aware dynamic analysis of managed languages. To make our presentation more concrete, we have selected JavaScript as both the implementation and target language. In Sections 3.5 and 5.6, we speculate how our approach could be ported to other popular managed languages, such as Python and Ruby. In the remainder of this section, we outline three main reasons why JavaScript is a suitable choice for demonstrating our approach.

First and foremost, JavaScript is one of the most popular programming languages. As shown in Figure 3.1[1], it has held the top position in popularity on GitHub for many years. It was only recently dethroned by Python in 2024. Several factors contribute to this popularity, with one of the most significant being its status as the de facto standard language of the web. By creating a tool that targets such a widely used language, we hope to attract greater attention to our research.

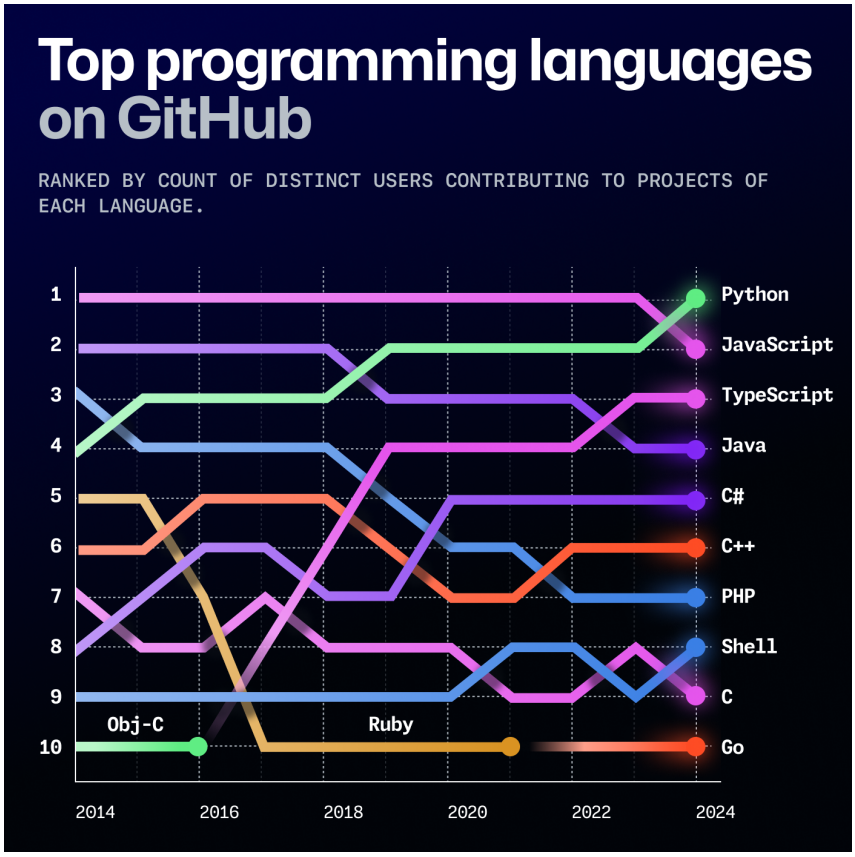


Figure 3.1: Language popularity on GitHub according to Octoverse 2024 [41]

Secondly, we argue that the core of JavaScript is well-suited to express many constructs found in other programming languages. On one hand, JavaScript was born from the promise made by Netscape in 1995 to allow Brendan Eich to implement Scheme in its navigator [95]. However, in the end, the Netscape management required that the syntax of its scripting language be similar to that of Java [95]. In response, Brendan essentially sugar-coated Scheme with a Java-like syntax. On the other hand, Scheme (a dialect of Lisp) is known for its ability to express complex language constructs despite being minimalist. As shown in Quotation 3.2, the book "Structure and Interpretation of Computer Programs" [1], widely used to teach fundamental principles of computer programming, has chosen Scheme as its framework language. Taken together, these arguments suggest that the core of JavaScript provides a suitable basis for discussing the challenges of supporting constructs found in other programming languages.

> If Lisp is not a mainstream language, why are we using it as the framework for our discussion of programming? Because the language possesses unique features that make it an excellent medium for studying important programming constructs and data structures and for relating them to the linguistic features that support them.

Figure 3.2: Excerpt from Structure and Interpretation of Computer Programs [1]

Third and finally, JavaScript has evolved into a complex language over the years, presenting a representative sample of the technical challenges involved in adapting our approach to other managed languages. It began in 1996 when Netscape decided to standardize JavaScript and requested ECMA International to host the standard. This standard is known as ECMA-262 [95] and is managed by the 39th Technical

[1] https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages

Name	Publ. Date	New Major Language Features
ES1	June 1997	First edition
ES2	August 1998	Editorial changes
ES3	December 1999	- Regular expressions - Exception handling
ES4	**Abandoned** (July 2008)	Ambitious modifications; some were completely dropped, some were included in ES2015.
ES5	December 2009	- Strict mode - Property accessors
ES5.1	June 2011	Editorial changes
ES2015 a.k.a., harmony	June 2015	- Classes - Native modules - Block scoping - `Proxy` and `Reflect` - Destructuring assignments - Arrow functions - Iteration and Promise protocols - Generator functions
ES2016	June 2016	Few minor features
ES2017	June 2017	- Asynchronous functions - Shared memory
ES2018	June 2018	- Asynchronous iteration - Rest/Spread properties
ES2019	June 2019	- Optional binding in `catch`
ES2020	June 2020	- Dynamic import - `import.meta` - BigInt - Optional chaining - Nullish coalescing
ES2021	June 2021	- Weak reference - Logical assignment
ES2022	June 2022	- Top level `await` - Class fields - Private fields and methods
ES2023	June 2023	Few minor features
ES2024	June 2024	Few minor features
ES2025	June 2025	Import attributes

Table 3.1: A brief history of ECMAScript versions

Committee of ECMA, also referred to as TC39. Due to trademark reasons, the language standardized by the ECMA-262 specification is called ECMAScript rather than JavaScript. As shown in Table 3.1, the TC39 committee has incorporated numerous major features into JavaScript over the years. Consequently, JavaScript has transformed from a relatively small language inspired by Scheme into the complex language we know today. This evolution demonstrates the expressiveness of the core of JavaScript.

As will be evident, the primary goal of this chapter is to restore the core simplicity of JavaScript to facilitate the development of analyses. Consequently, this chapter is quite technical and assumes that the reader is familiar with JavaScript. However, we will do our best to introduce lesser-known JavaScript concepts as they arise.

3.1.2 Design objectives of AranLang

In this section, we outline the rationale for introducing a new language named AranLang, which is a minimal variant of JavaScript.

ESTree[2] is the de facto standard abstract syntax tree for JavaScript, utilized by popular tools such as ESLint[3], Prettier[4], and Babel[5]. Its abstract grammar comprises 76 interfaces, of which 57 are recursive. Moreover, the meaning of these syntactic nodes can vary significantly based on their context. For example, a `MemberExpression` can denote a get operation in an expression context, a set operation on the left-hand side of an assignment, or a method invocation in the callee position. Therefore, we argue that directly instrumenting ESTree node types is not a *cohesive* approach to implementing dynamic analyses for JavaScript.

The main goal behind introducing AranLang is to simplify point-based instrumentation by expressing non-fundamental language constructs in terms of more fundamental ones. For instance, `x += 1` can be expressed as `x = x + 1`. Simplifying semantics for the purpose of facilitating program analysis or code transformation is a well-explored idea. One of the seminal works in this context is CIL [81], which is an intermediate representation of C. The research community has investigated this idea for managed languages such as λ_{JS} [46] for JavaScript pre-harmony (i.e., ES 5.1) and λ_{π} [93] for Python. To the best of our knowledge, our work is the first to support post-harmony JavaScript (i.e., ES2015); specifically, our work supports ES2025. This is significant because many challenging syntactic features have been added after harmony, including block-scoped variables, modules, coroutines, classes, and more.

Related are also compatibility transpilers, which aim to enable programs utilizing modern language features to run on legacy runtimes. In contrast to the aforementioned research endeavors, some of these transpilers are industrial-strength and actively maintained to support the latest language features. However, because their goal is to enhance compatibility rather than simplify program analysis and code transformation, they have different design objectives, resulting in different design decisions. For instance, the most prominent compatibility transpiler for JavaScript is Babel[6]. Since version 7.0, Babel converts generator functions and asynchronous functions into regular functions to make them available to legacy engines, while AranLang retains both constructs. In contrast, variable increment is not transpiled by Babel, as it has been part of the earliest JavaScript release. In summary, although Babel is a similar technology, it serves a different purpose and targets different language constructs for lowering.

While Babel targets JavaScript developers, language implementers have also recently shown interest in simplifying JavaScript. The rapid introduction of new features has made it challenging for them to provide ECMAScript-compliant runtimes that remain secure and performant. This issue was raised in a TC39 meeting in October 2024[7], where Shu-yu Guo, an engineer from Google, proposed essentially dividing JavaScript into two parts[8]: JS0, which would consist of a core subset of stable JavaScript features, and JSSugar, which would include fast-evolving and non-essential features that can be expressed in terms of JS0. To clarify, no additional information about JS0 has been provided, making it impossible to compare it with AranLang.

To achieve its primary objective of making our approach more cohesive, AranLang should be **minimalist**. Although, in theory, any Turing-complete language can express JavaScript semantics, AranLang should remain sufficiently **expressive** to convey most of the semantics of JavaScript without requiring excessive transpilation expansion that could harm the *source-adjacency* of our approach. If we push minimalism to the extreme, we could design AranLang to resemble an assembly-like language. In this case, we could utilize ahead-of-time JavaScript compilers such as Hopc [107, 108], Porffor[9], or Javy[10]. However, this

[2] https://github.com/estree/estree
[3] https://eslint.org
[4] https://prettier.io
[5] https://babeljs.io
[6] https://babeljs.io
[7] https://github.com/tc39/agendas/blob/main/2024/10.md
[8] https://docs.google.com/presentation/d/1ylROTu3N6MyHzNzWJXQAc7Bo1O0FHO3lNKfQMfPOA4o/edit#slide=id.p
[9] https://porffor.dev
[10] https://github.com/bytecodealliance/javy

approach would significantly lower the abstraction level of the insights, completely undermining source-adjacency. Moreover, it would incur substantial performance costs, as the low-level code would need to be rebuilt into JavaScript.

To clarify this tension between the minimalism of AranLang and its expressiveness, we separate the latter into two categories: **coverage** and **proximity**. Given a transpilation from JavaScript to AranLang, coverage evaluates how much of JavaScript semantics is preserved by the transformation, while proximity assesses the complexity of the transformation itself. Figure 3.3 illustrates the design space of AranLang. By definition, JavaScript has perfect coverage and proximity but does not help improve the cohesion of the approach. At the extreme, a low-level language like WebAssembly is very minimalist and capable of representing the entirety of JavaScript's semantics, but at the cost of heavy transpilation expansion. AranLang strikes a delicate balance among these three conflicting criteria, with a preference for coverage, then minimalism, and finally proximity.

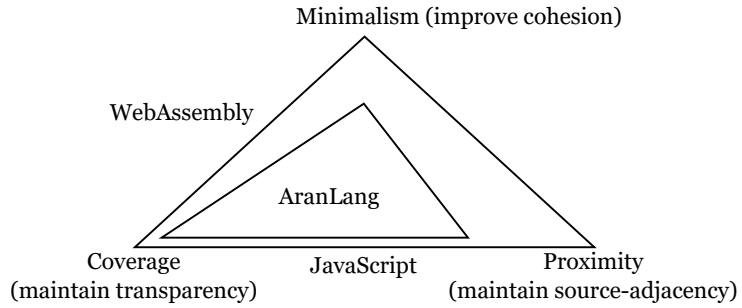


Figure 3.3: Position of AranLang within its design space

3.1.3 Motivating example for introducing AranLang

In this section, we illustrate how AranLang simplifies the analysis of JavaScript through a motivating example. The goal is to help the reader understand how AranLang fits within the overall approach.

Listing 3.1 provides a complete account of the instrumentation pipeline of our approach. It assumes that the analysis implementer has defined a global function named `__APPLY__`. For instance, in a taint analysis, this function would be responsible for propagating the taint from the input to the output of the call. Our overall approach prescribes three transformations involving four representations:

- Original JavaScript Program: The original program is a simple JavaScript module that logs the result of incrementing a local variable named `xyz`.
- Transpiled AranLang Program: Initially, the program is transpiled into AranLang. There are two main points of interest. First, the `log` variable has been removed, and its occurrences have been replaced with an inline import expression. Second, the increment operation has been substituted with a call to `%aran.performBinaryOperation%`.
- Instrumented AranLang Program: The instrumentation is applied to the AranLang program instead of the JavaScript program, yielding two key benefits. First, due to the desugaring of the increment operation, the analysis can propagate taint during this operation in the same manner as it would for a call to a built-in function. Second, with import inlining, the analysis no longer needs to track the connection between the `log` variable and the export of the same name from the `node:console` module.
- Instrumented JavaScript Program: To conduct the analysis, the AranLang program must be transpiled back into JavaScript so that it can be executed by standard JavaScript runtimes. This process is straightforward and assumes the existence of a global variable named `__intrinsic` that holds built-in values such as the `%Reflect.get%` function or custom values like the `%aran.performBinaryOperation%` function.

```
import { log } from "node:console";
let xyz = 1;
xyz++;
log(xyz);
"completion";
```

⇓ ⇓ ⇓

```
"module"; // Program directive
import 'log' from 'node:console'; // Import declaration without local binding
{
  let xyz = %aran.deadzone_symbol%; // Explicit hoisting
  xyz = 1; // Initialization
  xyz = %aran.performBinaryOperation%("+", xyz, 1); // No binary operation
  (import 'log' from 'node:console')(xyz); // Inline import expression
  return "completion"; // Explicit completion value
}
```

⇓ ⇓ ⇓

```
"module"
import 'log' from 'node:console';
{
  let xyz = %aran.deadzone_symbol%, _apply;
  _apply = %Reflect.get%(%aran.global_object%, "__APPLY__");
  xyz = 1;
  xyz = _apply(%aran.performBinaryOperation%, %undefined%, %Array.of%("+", xyz, 1));
  _apply(import 'log' from 'node:console', %undefined%, %Array.of%(xyz));
  return "completion";
}
```

⇓ ⇓ ⇓

```
import {'log' as __log } from 'node:console';
let xyz = __intrinsic["aran.deadzone_symbol"], _apply;
_apply = __intrinsic["Reflect.get"](__intrinsic["aran.global_object"], "__APPLY__");
xyz = 1
xyz = _apply(
  __intrinsic["aran.performBinaryOperation"],
  __intrinsic["undefined"],
  __intrinsic["Array.of"]("+", xyz, 1),
);
_apply(__log, __intrinsic["undefined"], __intrinsic["Array.of"](xyz));
"completion";
```

Listing 3.1: An example of the successive transformations performed by our approach

3.2 Syntax and design of AranLang

In Section 3.1, we outlined the motivation for introducing AranLang. Here, we provide a detailed description of the language and its design in relation to its objectives. This section is organized as follows:

- *Section 3.2.1 focuses on the syntax of AranLang.*
- *Sections 3.2.2 to 3.2.13 discuss several key points of the language's design.*

3.2.1 The syntax of AranLang

In this section, we present the syntax of AranLang by detailing its abstract grammar and concrete production rules.

Listing 3.2 provides an abstract grammar for AranLang, defined as an algebraic datatype in Haskell and named `AranTree`. Internally, Aran utilizes a TypeScript type definition[11] which is more verbose but describes a similar data structure.

Table 3.2 and Table 3.3 provide a context-free grammar for AranLang. The grammar is split into two tables for readability. The first table focuses on literals, programs, and blocks, while the second table focuses on statements, effects, and expressions. Elements in black are grammar variables, elements in blue are terminal symbols, and elements in orange are grammar combinators such as `?` for zero or one occurrence (optionality), `|` for alternation (or), and `*` for zero or more occurrences (Kleene star). Some grammar variables are omitted from the tables and are defined as follows:

- `Identifier`: Regular JavaScript identifier, such as `MyClass`.
- `JavaScriptKeyword`: Enumeration of reserved JavaScript identifiers defined in the specification, such as `var` and `while`.
- `ImplicitParameter`: Enumeration of reserved dot-separated identifiers defined in Section 3.2.4, such as `import.meta` or `super.get`.
- `JsonLiteral`: Regular JSON primitive literal, such as `null`, `true`, `123`, or `"foo"`.
- `BigIntLiteral`: Regular JavaScript big integer literal, such as `123n`.
- `SingleQuoteStringLiteral`: Regular JavaScript single quote literal, such as `'foo'`.

[11] `https://github.com/lachrist/aran/blob/f97595d1/lib/lang/syntax.d.ts`

```haskell
-- Synonym --
type ExternalVariable = String
type InternalVariable = String
type InternalVariableOrImplicitParameter = String
type InternalLabel = String
type Asynchronous = Bool
type Delegate = Bool
type Specifier = String
type Source = String
type Intrinsic = String
-- Primitive --
data Primitive
  = NullPrimitive
  | BooleanPrimitive Bool
  | NumberPrimitive Double
  | BigIntPrimitive Integer
  | StringPrimitive String
data LetExternalDeclaration = LetExternalDeclaration ExternalVariable
data VarExternalDeclaration = VarExternalDeclaration ExternalVariable
type ExternalDeclaration = Either LetExternalDeclaration VarExternalDeclaration
data ModuleHeader
  = ImportHeader (Maybe Specifier) Source
  | ExportHeader Specifier
  | AggregateHeader (Maybe Specifier) (Maybe Specifier) Source
data Program
  = ModuleProgram [ModuleHeader] RoutineBlock
  | ScriptProgram [ExternalDeclaration] RoutineBlock
  | GlobalEvalProgram [VarExternalDeclaration] RoutineBlock
  | RootLocalEvalProgram [VarExternalDeclaration] RoutineBlock
  | DeepLocalEvalProgram RoutineBlock
-- Block --
data Declaration = Declaration InternalVariable Intrinsic
data ControlBlock = ControlBlock [InternalLabel] [Declaration] [Statement]
data RoutineBlock = RoutineBlock [Declaration] [Statement] Expression
data GeneratorBlock
  = GeneratorBlock [Declaration] [Effect] [Statement] Expression
-- Statement --
data Statement
  = EffectStatement Effect
  | BreakStatement InternalLabel
  | DebuggerStatement
  | BlockStatement ControlBlock
  | IfStatement Expression Statement Statement
  | WhileStatement Expression Statement
  | TryStatement ControlBlock ControlBlock ControlBlock
-- Effect --
data Effect
  = ExpressionEffect Expression
  | ConditionalEffect Expression [Effect] [Effect]
  | ExportEffect Specifier Expression
  | WriteEffect InternalVariableOrImplicitParameter Expression
-- Expression --
data Expression
  = PrimitiveExpression Primitive
  | IntrinsicExpression Intrinsic
  | ImportExpression (Maybe Specifier) Source
  | ReadExpression InternalVariableOrImplicitParameter
  | ArrowExpression Asynchronous RoutineBlock
  | FunctionExpression Asynchronous RoutineBlock
  | GeneratorExpression Asynchronous GeneratorBlock
  | AwaitExpression Expression
  | YieldExpression Delegate Expression
  | SequenceExpression [Effect] Expression
  | ConditionalExpression Expression Expression Expression
  | EvalExpression Expression
  | ApplyExpression Expression Expression [Expression]
  | ConstructExpression Expression [Expression]
```

Listing 3.2: Abstract grammar for AranLang defined as Haskell data types

Primitive	JsonLiteral | BigIntLiteral
InternalLabel	Identifier (. Identifier)* ! JavaScriptKeyword
InternalVariable	Identifier (. Identifier)* ! JavaScriptKeyword ! ImplicitParameter
ExternalVariable	Identifier ! JavaScriptKeyword
Intrinsic	% Identifier (. Identifier)* (@get|@set)? %
Specifier	SingleQuoteStringLiteral
Source	SingleQuoteStringLiteral
ModuleHeader	ImportHeader | ExportHeader | AggregateHeader
ImportHeader	import (Specifier | *) from Source ;
ExportHeader	export Specifier ;
AggregateHeader	export * from Source ;
AggregateHeader	export * as Specifier from Source ;
AggregateHeader	export Specifier as Specifier from Source ;
Program	ModuleProgram | ... | DeepLocalEvalProgram
ModuleProgram	"module" ; ModuleHeader* RoutineBlock
ScriptProgram	"script" ; ((var | let) ExternalVariable ;)* RoutineBlock
GlobalEvalProgram	"eval-global" ; (var ExternalVariable ;)* RoutineBlock
RootLocalEvalProgram	"eval-local-root" ; (var ExternalVariable ;)* RoutineBlock
DeepLocalEvalProgram	"eval-local-deep" ; RoutineBlock
Declaration	let InternalVariable = Intrinsic ; let InternalVariable ;
SegmentBlock	(InternalLabel :)* { Declaration* Statement* }
RoutineBlock	{ Declaration* Statement* return Expression ; }
GeneratorBlock	{ Declaration* ([Effect (, Effect)*] ;)? Statement* return Expression ; }

Table 3.2: Production rules for AranLang (1/2)

Statement	EffectStatement | ... | TryStatement
EffectStatement	Effect ;
BreakStatement	break InternalLabel ;
DebuggerStatement	debugger ;
BlockStatement	SegmentBlock
IfStatement	if (Expression) SegmentBlock else SegmentBlock
WhileStatement	while (Expression) SegmentBlock
TryStatement	try SegmentBlock catch SegmentBlock finally SegmentBlock
Effect	ExpressionEffect | ... | WriteEffect
ExpressionEffect	Expression
ConditionalEffect	Expression ? (Effect*) : (Effect*)
ExportEffect	export Specifier = Expression
WriteEffect	(InternalVariable | ImplicitParameter) = Expression
Expression	PrimitiveExpression | ... | ConstructExpression
PrimitiveExpression	Primitive
IntrinsicExpression	Intrinsic
ReadExpression	InternalVariable | ImplicitParameter
ImportExpression	import (Specifier | *) from Source
ArrowExpression	async? arrow RoutineBlock
FunctionExpression	async? function RoutineBlock
MethodExpression	async? method RoutineBlock
GeneratorExpression	async? generator GeneratorBlock
AwaitExpression	await Expression
YieldExpression	yield *? Expression
SequenceExpression	((Effect ,)+ Expression)
ConditionalExpression	Expression ? Expression : Expression
EvalExpression	eval Expression
ApplyExpression	Expression ( )
ApplyExpression	Expression (Expression (, Expression)*)
ApplyExpression	Expression (that Expression (, Expression)*)
ConstructExpression	new Expression ( )
ConstructExpression	new Expression (Expression (, Expression)*)

Table 3.3: Production rules for AranLang (2/2)

3.2.2 Explicit program kind through directives

In this section, we present the directives that must be included in AranLang programs.

Unlike JavaScript, AranLang requires that programs be annotated with a directive specifying their kind. The kind of a program defines the context in which the AranLang program will be executed after transpilation back to JavaScript. As is detailed in Section 3.3, this information is necessary to accurately transpile AranLang back to JavaScript. Table 3.4 lists the five kinds of AranLang programs, including their execution contexts and allowed top-level declarations.

Program Kind	Execution Context	Top-Level Declaration
`ModuleProgram`	Native ECMAScript module	none (ES modules cannot declare global variables)
`ScriptProgram`	Top-level script	`var` & `let` (global `const` cannot be expressed)
`GlobalEvalProgram`	Indirect `eval` call	`var` (`let` & `const` are scoped)
`RootLocalEvalProgram`	Direct `eval` call within JS	`var` (`let` & `const` are scoped)
`DeepLocalEvalProgram`	Direct `eval` call within AranLang code that has been transpiled back to JS	none (deep `var` cannot be expressed, and both `let` and `const` are scoped)

Table 3.4: The five kinds of programs in AranLang

3.2.3 Intrinsic literal expressions for stable retrieval

In this section, we detail the mechanism in AranLang for reliably fetching intrinsic values.

In JavaScript, intrinsic values are predefined objects and functions specified by the language and denoted using percentage notation. Many intrinsics are exposed as properties of the global object, such as the `%Math%` object, while others, like the constructor of asynchronous functions `%AsyncFunction%`, are not available in the global scope. Often, transpilation expansion requires access to these intrinsic values. For instance, as depicted in Listing 3.3, naively transpiling a regular expression literal into a call to the external variable `RegExp` does not guarantee transparency, as the variable may have been shadowed. To safeguard against shadowing and mutations of the global object, we introduce a global immutable cache of intrinsic values, which can be syntactically accessed via intrinsic expressions.

```
1 new RegExp("pattern", "flags"); // ❌ Dynamic evaluation (JavaScript)
```

```
1 new %RegExp%("pattern", "flags"); // ✅ Static evaluation (AranLang)
```

Listing 3.3: Syntactic intrinsic expression to safeguard against shadowing

Intrinsic values can be top-level properties of the global object, such as `%Array%`, or deeply nested, like `%Array.prototype.concat%`. In these cases, AranLang adheres to the conventions established by the JavaScript specification. Additionally, intrinsic values can be accessed as accessor functions for properties of global objects. In this context, AranLang employs either the `@get` or `@set` suffix, which is not part of the JavaScript specification. For instance, `%Function.prototype.arguments@get%` denotes the initial value of the `get` accessor for the `arguments` property of the `%Function.prototype%` intrinsic.

In addition to these globally available intrinsic values, AranLang defines several custom intrinsic values. Table 3.5 lists all the intrinsic values introduced by AranLang, categorizing them into five groups:

1. Intrinsic values that are specified in JavaScript but cannot be retrieved from the global object. This category includes only the prototypes of generators and asynchronous generators.

2. Custom intrinsic values that are not functions: a symbol to mark the temporal dead zone, the global object, and the global declarative record.

3. Custom intrinsic functions that cannot be expressed in AranLang. These functions are essential for transpiling JavaScript to AranLang; their removal would necessitate the introduction of new language constructs. For instance, AranLang does not support binary operations; instead, it utilizes the `%aran.performBinaryOperation%` intrinsic function.

4. Custom intrinsic functions that are not essential for transpiling JavaScript to AranLang but help reduce transpilation expansion. For example, `%aran.getValueProperty%` implements regular property lookup, which can also be achieved using `%Reflect.get%`, but it requires converting the target into an object first. For instance, as shown in Listing 3.4, `%aran.getValueProperty%` enables a more concise transpilation of `obj[key]` compared to `%Reflect.get%`.

5. Custom intrinsic functions that enable support for direct calls to `%eval%`. By default, these functions are placeholders that throw an instance of `%SyntaxError%` upon invocation. To support direct calls to `%eval%`, they should be overridden with the appropriate exports from our implementation of AranLang. This design decision aims to minimize performance overhead for programs that do not utilize dynamic code evaluation and do not require bundling with Aran.

```
// %aran.getValueProperty%
%aran.getValueProperty%(obj, key);
// %aran.performBinaryOperation%
%Reflect.get%(
  %aran.performBinaryOperation%("==", obj, null)
    ? obj
    : %Object%(obj),
  key,
);
```

Listing 3.4: Two possible transpilations for the expression `obj[key]`

Custom intrinsic functions are not without cost, as they impose a cognitive load on the analysis that must account for them. The alternative for essential custom intrinsic functions is to integrate them into AranLang as language constructs. For example, `%aran.performBinaryOperation%` could be replaced by reintroducing binary operations into AranLang. However, this would necessitate the creation of multiple instrumentation join points: one for the left argument, one for the right argument, and one for the result. Additionally, this approach would not allow for the handling of potential exceptions thrown by the operation. Therefore, to enhance the minimalism of AranLang and maximize expressiveness of our approach, we prioritize essential custom intrinsic functions over language constructs whenever possible.

In contrast, non-essential custom intrinsic functions were introduced to reduce transpilation expansion, thereby enhancing source-adjacency. These functions illustrate the tension between the minimalism of AranLang and its proximity to JavaScript. We determined that a custom intrinsic function is always warranted when its implementation requires looping over a data structure, which is particularly detrimental to source-adjacency in an expression context because it necessitates the invocation of an inline function. A few non-essential custom intrinsic functions that do not require looping over a data structure, such as `%aran.getValueProperty%`, have been added due to their frequent recurrence in transpilation expansion.

Intrinsic	Description
Hidden Global Intrinsics	
`%aran.GeneratorFunction.prototype.prototype%`	The prototype of generator functions
`%aran.AsyncGeneratorFunction.prototype.prototype%`	The prototype of async generator functions
Non-Function Custom Intrinsics	
`%aran.deadzone_symbol%`	The symbol for denoting the TDZ of a variable
`%aran.global_object%`	The value used by Aran as the global object Default: `%globalThis%`
`%aran.global_declarative_record%`	The object for reifying the global decl. record
Essential Custom Intrinsics	
`%aran.declareGlobalVariable%`	Declares a `var` global variable e.g., `var x` ≡ `%aran.declareGlobalVariable%("x")`
`%aran.readGlobalVariable%`	Reads a global variable e.g., `x` ≡ `%aran.readGlobalVariable%("x")`
`%aran.typeofGlobalVariable%`	Retrieves the type of a global variable Does not throw if the variable is missing e.g., `typeof x` ≡ `%aran.typeofGlobalVariable%("x")`
`%aran.discardGlobalVariable%`	Deletes a global variable e.g., `delete x` ≡ `%aran.discardGlobalVariable%("x")`
`%aran.writeGlobalVariableStrict%`	Writes to a global variable in strict mode e.g., `x=123` ≡ `%aran.writeGlobalVariableStrict%("x",123)`
`%aran.writeGlobalVariableSloppy%`	Writes to a global variable in sloppy mode e.g., `x=123` ≡ `%aran.writeGlobalVariableSloppy%("x",123)`
`%aran.performUnaryOperation%`	Performs a unary operation e.g., `!x` ≡ `%aran.performUnaryOperation%("!",x)`
`%aran.performBinaryOperation%`	Performs a binary operation e.g., `x+y` ≡ `%aran.performBinaryOperation%("+",x,y)`
`%aran.throwException%`	Throws an exception e.g., `throw e;` ≡ `%aran.throwException%(e)`
`%aran.isConstructor%`	Indicates whether a value is a constructor
Non-Essential Custom Intrinsics	
`%aran.createObject%`	Creates an object from a flat list of entries e.g., `{__proto__:p,[k]:x}` ≡ `%aran.createObject(p,k,x)%`
`%aran.getValueProperty%`	Looks up the property value of an object e.g, `o[k]` ≡ `%aran.getValueProperty(o,k)%`
`%aran.toPropertyKey%`	Converts any value to a key.
`%aran.toArgumentList%`	Converts an array to array-like object similar to the `arguments` object.
`%aran.sliceObject%`	Removes a set of properties from an object
`%aran.listForInKey%`	List the enumerable string keys of an object
`%aran.listIteratorRest%`	List the remainder values in an iterator
Eval-Related Custom Intrinsics	
`%aran.transpileEvalCode%`	Convert JavaScript code into AranLang AST
`%aran.retropileEvalCode%`	Revert AranLang AST into JavaScript code

Table 3.5: Custom intrinsic values defined in AranLang

3.2.4 No explicit parameters to simplify scope analysis

In this section, we describe the mechanism in AranLang for retrieving context-dependent values that are predefined in programs, closures, and catch blocks.

JavaScript features both explicit parameters, such as the formal parameters of a function, and implicit parameters, such as the implicit `this` parameter of a function. We decided to eliminate explicit parameters from AranLang in favor of implicit parameters only. This approach simplifies scope analysis by establishing a clear distinction between implicit parameters, which are initialized with a context-dependent value, and internal variables which begin as undefined or reside within the temporal dead zone.

In AranLang, implicit parameters cannot be declared but can be read from and written to, similar to internal variables. The ability to overwrite implicit parameters is a key feature of AranLang, enabling analyses to intercept and modify them. In contrast, JavaScript often treats implicit parameters as keywords; for instance, `import.meta` is an expression that provides information about the current module but cannot appear on the left-hand side of an assignment.

Table 3.6 lists all implicit parameters specified in AranLang. The middle column indicates the context in which each implicit parameter is defined. Below the double division line are implicit parameters defined in the "eval" context, which denote the generation of AranLang code to be fed to a direct `eval` call present in arbitrary JavaScript code. This interoperability between AranLang and JavaScript requires 12 dedicated implicit parameters, while the remaining 7 parameters are more straightforward and resemble standard JavaScript.

Note that most implicit parameters defined in AranLang include a dot character. Similar to existing implicit parameters in JavaScript, such as `new.target` and `import.meta`, these dots should not be interpreted as property access; instead, they serve as a semantic grouping between parameters.

Similarly to custom intrinsic functions, implicit parameters should be introduced sparingly, as they impose a cognitive load on the analysis implementer. However, similar to intrinsic values, the alternative would be to integrate them into AranLang as additional language constructs, which would significantly diminish the expressiveness of our approach.

During the design of AranLang, certain functionalities fluctuated between being exposed as implicit parameters or intrinsic functions. For example, `%aran.readGlobalVariable%` was initially available as `scope.read` for global programs, rather than just local ones. Ultimately, we chose to expose functionalities independent of the execution context as intrinsic values instead of implicit parameters. This decision clarifies that they possess fixed semantics and prevents the creation of multiple versions of the same function.

Implicit Variable	Context	Description
`import`	program	(almost regular JS)
`import.meta`	module & eval	(regular JS)
`this`	function & generator & program	(regular JS)
`new.target`	function & generator & eval	(regular JS)
`function.arguments`	closure	The argument values in an array
`function.callee`	closure	The function being called
`catch.error`	catch	The caught exception
`super.get`	eval	Looks up a `super` property e.g., `super[k]` ≡ `super.get(k)`
`super.set`	eval	Assigns a `super` property e.g., `super[k]=v` ≡ `super.set(k,v)`
`super.call`	eval	Calls the `super` constructor e.g., `super(x1,x2)` ≡ `super.call(%Array.of%(x1,x2))`
`private.check`	eval	Checks that all the elements of an array are valid private properties
`private.get`	eval	Looks up a private property e.g., `o#k` ≡ `private.get(o,"k")`
`private.has`	eval	Indicates if a private property exists e.g., `#k in o` ≡ `private.has(o,"k")`
`private.set`	eval	Assigns a private property e.g., `o.#k=v` ≡ `private.set(o,"k",v)`
`scope.read`	eval	Reads an external variable e.g., `x` ≡ `scope.read("x")`
`scope.writeStrict`	eval	Writes to an ext. var. (strict) e.g., `x=v` ≡ `scope.writeStrict("x",v)`
`scope.writeSloppy`	eval	Writes to an ext. var. (sloppy) e.g., `x=v` ≡ `scope.writeSloppy("x",v)`
`scope.typeof`	eval	Retrieves an external variable's type e.g., `typeof x` ≡ `scope.typeof("x")`
`scope.discard`	eval	Discards an external variable e.g., `delete x` ≡ `scope.discard("x")`

Table 3.6: The implicit parameters of AranLang

3.2.5 Mandatory variable hoisting

In this section, we detail how AranLang enforces variable hoisting, thereby eliminating the temporal dead zone.

In JavaScript, variables are conceptually declared before their explicit declaration in the code. For newer variable types (e.g., `let`, `const`, and `class`) the explicit declaration aligns with their initialization. This behavior, known as *hoisting*, creates a *temporal dead zone* (TDZ) between the point of declaration and initialization. During this period, accessing the variable results in a `%ReferenceError%`. Consequently, variable hoisting complicates scope analysis.

In AranLang, hoisting is enforced: variables must be declared at the beginning of blocks—i.e., `ControlBlock`, `RoutineBlock`, and `GeneratorBlock`. Additionally, only `let` variables are allowed, which further simplifies scope analysis by enforcing a single, consistent variable type.

The production rule `Declaration` specifies that variables can be initialized with any intrinsic value. However, in practice, variables from programs transpiled from JavaScript should only be initialized with `%undefined%` or `%aran.deadzone_symbol%`. Note that if the intrinsic initializer is omitted, it serves as a shorthand for `%undefined%`; thus, `let x;` is equivalent to `let x=%undefined%;`.

The variables described above are classified as *internal* because their access is entirely controlled by AranLang. The language also includes declarations for *external* variables, whose bindings reside in a scope frame that is not managed by AranLang. Unlike internal variables, external variables must be valid JavaScript identifiers and cannot contain a dot character. These variables are hoisted to the beginning of script programs and eval programs as part of the `ScriptHeader`. External variables come in two kinds (i.e., `var` and `let`) and do not have initializers. The handling of external variables in AranLang is a delicate matter, which we discuss further in Section 3.4.4.

3.2.6 Safe lexical scoping

In this section, we explain why variable access in AranLang is both safe and atomic.

AranLang is carefully designed to ensure both the atomicity and safety of variable access. This important feature simplifies scope analysis and reduces the number of instrumentation join points, as read operations only require an "after" notification with the result, while write operations only require a "before" notification with the new value.

On one hand, external variables whose bindings escape the control of AranLang have been separated from internal variables, which are accessed via either intrinsic functions such as `%aran.readGlobalVariable%` or implicit bindings such as `scope.read`. This separation ensures that internal variables are always lexically scoped, whether they are explicitly declared variables or implicitly declared parameters. Such lexical scoping guarantees the atomicity of access to internal variables, as it does not trigger any property lookup, unlike dynamic scope frames in JavaScript.

On the other hand, access to these lexically scoped internal variables is safe because all sources of errors have been eliminated from AranLang. First, the temporal dead zone is no longer present due to mandatory hoisting. Second, the language does not feature constant variables, as all implicitly and explicitly declared variables are mutable.

3.2.7 Modeling assignments without results

In this section, we present AranLang's effect system, which is designed to optimize the instrumentation of write and export expressions when their results are discarded.

In JavaScript, write expressions return the new value assigned to variables; for instance, `(x=123)` is an expression that evaluates to `123` at run-time. However, write expressions often appear in a statement context, where their results are directly discarded. In this case, analyses that mirror the value stack would require an additional discard notification, which could be avoided by representing the write operation as an *effect*, which can be seen as an expression without a result value. This is the original motivation behind the `Effect` nodes presented in Table 3.3.

Table 3.7 compares two JavaScript instrumentation strategies for implementing an analysis responsible for mirroring the value stack: one where `writePeek` peeks into the value stack, and one where `writePop` pops the top value off the value stack. The "peek" strategy works best when the value of the write expression is used but requires an additional discard notification when the value of the write expression is discarded. In contrast, the "pop" strategy is more effective when the value of the write expression is discarded but necessitates a temporary variable to restore the last value of the stack if needed.

Strat.	**Expression Context**	**Statement Context**
	`f(x = 123);`	`x = 123;`
Peek	`f(x = writePeek("y", push(123)));`	`discard(x = writePeek("x", push(123)));`

Pop	`f((`
	`  tmp = writePop("tmp", push(123)),`
	`  x = writePop("x", read("tmp", tmp)),`
	`  read("tmp")`
	`));`
		`x = writePop("x", primitive(123));`

Table 3.7: Comparison of the "peek" and "pop" strategies for mirroring the value stack

We chose the "pop" strategy over the "peek" strategy for two primary reasons. First, the "pop" strategy permits the value assigned to the variable to differ from the resulting value of the assignment, which is not feasible with the "peek" strategy. This flexibility is advantageous for analyses that treat scope and stack values differently. Second, our observations indicate that write expressions occur more frequently in a statement context than in an expression context, indicating that their results are often discarded. Consequently, we believe the "pop" strategy will be more efficient in practice.

Effects were introduced in AranLang to facilitate the "pop" instrumentation strategy. In AranLang, both effects and expressions are transpiled into JavaScript expressions. However, while the result of a transpiled AranLang effect is always discarded, the result of a transpiled AranLang expression can only be discarded if it appears within an `ExpressionEffect`. Besides `ExpressionEffect`, which explicitly discards the result of an expression, there are two other primary effects: `WriteEffect` for internal variable assignments and `ExportEffect` for updating exports.

Effects can be combined into a `ConditionalEffect`, where each branch consists of a list of effects. There is no need to introduce a syntactic node to chain effects together, as they always appear in groups within the grammar—whether in the head of a `GeneratorBlock`, as initial items of a `SequenceExpression`, or as branches of a `ConditionalEffect`. The only exception is the `StatementEffect`, which can contain only a single effect. However, this limitation is inconsequential, as statements also always appear in groups within blocks.

3.2.8 Explicit `this` argument

In this section, we explain why the `this` argument is explicit in AranLang.

An essential aspect of object-oriented programming in JavaScript involves passing a hidden argument that is assigned to the implicit `this` parameter within functions. In AranLang, this argument is made explicit inside `ApplyExpression`, as in `f(that t, x)`, where the value of `t` is assigned to `this` inside `f`. For conciseness, it can be omitted if it is `%undefined%`. Therefore, `f(x)` serves as shorthand for `f(that - %undefined%, x)`.

As shown in Listing 3.5, the explicit `that` argument may require some transpilation expansion to represent JavaScript method invocations transparently. If the receiver is pure, the transpilation is straightforward. However, when side effects are possible, a temporary transpilation variable must be introduced.

```
// o.m(x);
%aran.getPropertyValue%(o, "m")(that = o, x);

// f().m(x);
tmp = f();
%aran.getPropertyValue%(tmp, "m")(that = tmp, x);
```

Listing 3.5: Transpilation of JavaScript method invocations

To avoid introducing a transpilation variable for the receiver, one might be tempted to create a dedicated syntax for method invocations that explicitly includes the receiver object, the method name, and the arguments. However, as illustrated in Listing 3.6, this syntax cannot be instrumented transparently:

- Lines 2–7 declare a simple object whose particularity is to produce a side effect when accessing its `method` property.
- At line 8, the method is invoked syntactically: first, the `method` property is accessed, followed by the evaluation of the arguments.
- At line 9, the analysis is notified through the `invoke` global function, which is straightforward but unfortunately does not preserve the evaluation order.
- In contrast, at line 10, the analysis is notified via the `apply` global function, which may introduce a transpilation variable but preserves the evaluation order.

```
import { log } from "node:console";
const receiver = {
  get method () {
    console.log("get");
    return () => {};
  }
};
receiver.method(log("arg"));                    // ✅ get >> arg (original ordering)
invoke(receiver, "method", [log("arg")]);       // ❌ arg >> get (incorrect ordering)
apply(receiver.method, receiver, [log("arg")]); // ✅ get >> arg (correct ordering)
```

Listing 3.6: Comparison of the reflective `apply` and the simpler `invoke` functions

3.2.9 Mandatory return statement

In this section, we emphasize that return statements are mandatory in AranLang, both in programs and function bodies.

In JavaScript, function bodies can contain multiple and arbitrarily deep `return` statements, or even no `return` statement at all. Additionally, programs have an implicit return value known as *completion*, which is determined by the last evaluated expression statement. This behavior can be difficult to reason about

and includes corner cases, such as when the completion value occurs within a `try` statement. In contrast, AranLang mandates that every program and closure must have a unique, mandatory return statement at the end of its body, which is either a `RoutineBlock` or a `GeneratorBlock`. This explicit return value simplifies control-flow analysis.

3.2.10 Four function types

In this section, we detail the four function types of AranLang, with particular emphasis on the decision to retain method functions.

AranLang includes four types of expressions that evaluate to a closure: `ArrowExpression`, which is transpiled into a JavaScript arrow function; `FunctionExpression`, which is transpiled into a JavaScript plain function; `MethodExpression`, which is transpiled into a JavaScript method function; and `GeneratorFunction`, which is transpiled into a JavaScript generator function. All four function types can be made asynchronous and may embed `AwaitExpression`. None of them feature explicit parameters, and they all implicitly define `function.callee` and `function.arguments`. Additionally, `FunctionExpression` and `GeneratorFunction` implicitly define `this` and `new.target`, while `GeneratorFunction` can embed `YieldExpression`.

Closure generating expressions have been retained in AranLang more than any other JavaScript feature. This prominence arises from JavaScript's origins in Scheme, which emphasizes closures to emulate constructs from various programming paradigms, including object-oriented programming. This aligns with the primary objective of AranLang (already stated in Section 3.1.1) which is to restore the core simplicity of JavaScript.

To enhance the expressiveness of Aran, it may be tempting to simplify AranLang by merging all function types. In what follows, we explain why arrow functions and methods must be retained to preserve transparency. In Section 3.2.11, we justify our decision to retain asynchronicity and generators to preserve source-adjacency.

For the most part, arrow and method functions can be simulated using transpilation expansion with plain functions. This involves adding an instruction at the beginning to throw a type error if the function is invoked as a constructor, and at the end to convert the result if necessary. Unfortunately, as shown in Listing 3.7, there are four semantic discrepancies that cannot be easily addressed:

- Methods do not contain a `prototype` property, whereas functions have a `prototype` property that can be overwritten but not removed.
- Methods cannot be extended into a class.
- Methods cannot be passed as the third argument to `%Reflect.construct%`.
- It is not possible to intercept the construct operation on methods with proxies.

```
const f = function () {} // function
const { m } = { m () {} }; // method

/* 1) Functions have a non-configurable prototype property, and method do not */
Object.hasOwn(m, "prototype"); // 🆗 false
Reflect.getOwnPropertyDescriptor(f, "prototype");
  // 🆗 { value: {}, writable: true, enumerable: false, configurable: false }

/* 2) Methods cannot be extended */
class fClass extends f {} // 🆗
class mClass extends m {} // 🆗 TypeError

/* 3) Methods cannot be provided as new.target */
Reflect.construct(class {}, [], f); // 🆗
Reflect.construct(class {}, [], m); // 🆗 TypeError

/* 4) Proxified methods cannot intercept construct */
const fProxy = new Proxy(f, { construct: () => ({foo: 123}) });
const mProxy = new Proxy(m, { construct: () => ({foo: 123}) });
new fProxy(); // 🆗 { foo: 123 }
new mProxy(); // 🆗 TypeError
```

Listing 3.7: Behavioral divergences between methods and plain functions

3.2.11 Retention of coroutines

> *In this section, we discuss the design decisions to retain asynchronous and generator functions in AranLang.*

Both asynchronous and generator closures are retained in AranLang. As demonstrated in Section 4.1.3, developing analyses for these closures is manageable; however, their elimination would enhance Aran's cohesion. We have identified two approaches for their removal, but we believe that neither presents a satisfactory trade-off.

The most straightforward approach is continuation-passing style (CPS), which involves wrapping the remaining control flow into a "continuation" function that represents the next steps of computation. For simple cases, this method works well and produces understandable code. For instance, Listing 3.8 defines an asynchronous function `addAsync` and its promise-based CPS transpilation `addPromise`. Both functions return a promise that resolves to the sum of the resolved values of `promiseA` and `promiseB`. The only difference is that `addPromise` makes the control flow more explicit by registering `then` callbacks, whereas `addAsync` uses the more ergonomic `await` keyword.

```
const addAsync = async (promiseA, promiseB) =>
  await promiseA + await promiseB;
const addPromise = (promiseA, promiseB) =>
  promiseA.then(
    (a) => promiseB.then(
      (b) => Promise.resolve(a + b)
    )
  );
```

Listing 3.8: Simple promise-based CPS transpilation of `async` functions

Unfortunately, CPS transpilation becomes more complex when `await` expressions are embedded within control flow, as seen in `sumAsync` in Listing 3.9. In such cases, loops must be transformed into recursion, and the state of the control flow must be explicitly represented as arguments.

Not only is a fully-fledged CPS transformation technically challenging to implement, but it also results in code bloat and performance overhead. Moreover, CPS transpilation generates numerous additional closures that must be accounted for in the analysis, making insights more difficult to interpret and trace back to the source code. We believe this trade-off is not favorable.

The second approach is more advanced and was developed to efficiently handle `await` and `yield` expressions that are deeply nested within the control flow. This method involves transpiling coroutines into an

```
const sumAsync = async (promises) => {
  let total = 0;
  for (let index = 0; index < promises.length; index++)
    total += await promises[index];
  return total;
};
const sumPromise = (total, index, promises) =>
  index === promises.length
    ? Promise.resolve(total)
    : promises[index].then(
        (item) => sumPromise(total + item, index + 1, promises)
      );

// test //
console.log([
  await sumAsync([Promise.resolve(2), Promise.resolve(3)]),
  await sumPromise(0, 0, [Promise.resolve(2), Promise.resolve(3)]),
]);
```

Listing 3.9: Promise-based CPS transpilation of `async` functions with state management

explicit state machine and has been implemented in a library called Regenerator[12], which is utilized by the popular JavaScript compatibility transpiler Babel. As shown in Listing 3.10, the transformation is not straightforward and requires a runtime named `regeneratorRuntime`, which is responsible for advancing the state machine implemented by the transpiled generator.

```
// Original //
function* repeat123 () {
  while (true)
    yield 123;
}
console.log(Array.from(g())); // {value: 123, done: false}
```

⇓ ⇓ ⇓

```
// Regenerator //
var _marked = /*#__PURE__*/regeneratorRuntime.mark(repeat123);
function repeat123() {
  return regeneratorRuntime.wrap(function repeat123$(_context) {
    while (1) {
      switch (_context.prev = _context.next) {
        case 0:
          if (!true) {
            _context.next = 5;
            break;
          }
          _context.next = 3;
          return 123;
        case 3:
          _context.next = 0;
          break;
        case 5:
        case "end":
          return _context.stop();
      }
    }
  }, _marked);
}
console.log(repeat123().next()); // {value: 123, done: false}
```

Listing 3.10: Example of generator transpilation by the Regenerator library

In summary, CPS transpilation is straightforward but becomes complex when execution is paused deeply within the control flow. In contrast, Regenerator transpilation handles this situation more effectively, although it requires advanced machinery. We believe that the increased cohesion gained by removing coroutines does not justify the loss of source-adjacency incurred by either approach. As a result, we decided to retain coroutines in AranLang.

[12]https://github.com/facebook/regenerator

3.2.12 Simplified module imports and exports

In this section, we explain how AranLang simplifies the process of importing and exporting values within modules.

On the one hand, in a JavaScript module, exports from peer modules are imported as local variables. These variables are locally immutable, but any mutations within the peer module are reflected in the imports. Consequently, analyzing imports necessitates maintaining the link between import variables and their corresponding peer locations. This requirement can be alleviated by inlining the source and specifier of the import whenever the import variable is utilized. This approach has been implemented in AranLang through the combination of `ImportHeader` and `ImportExpression`.

On the other hand, in a JavaScript module, an export specifier can be attached to a local variable as follows: `export {x as y}`. When `x` is modified, this change is reflected in the peer modules that import `y`. Similar to import variables, analyzing this behavior requires maintaining the link between local variables and their export specifiers. This requirement can be alleviated by inlining the export specifier whenever the local variable is modified. This approach has been implemented in AranLang through the combination of `ExportHeader` and `ExportEffect`.

Listing 3.11 provides an example of an AranLang module that depends on a peer module located at `./math.mjs` and exports `1-pi` and `2-pi`. It demonstrates that module declarations are hoisted to the beginning of `ModuleProgram` as `ModuleHeader` nodes. The set of `ImportHeader` nodes defines the `ImportExpression` nodes that can occur at any depth within the module, while the set of `ExportHeader` nodes defines the `ExportEffect` nodes that can also occur at any depth. Additionally, exports from peer modules can be re-exported using `AggregateHeader`. A beneficial side-effect of this approach is that it provides a clear view of the module interface and its dependencies to the analysis. An example of the transpilation of an AranLang module into a JavaScript module is presented in Section 3.3.5.

```
1  "module";
2  // Head //
3  import 'pi' from './math.mjs';           // ImportHeader
4  export 'pi' as '1-pi' from './math.mjs'; // AggregateHeader
5  export '2-pi';                           // ExportHeader
6  // Body //
7  {
8    let pi, double_pi;
9    pi = import 'pi' from './math.mjs'; // ImportExpression
10   double_pi = %aran.performBinaryOperation%("*", 2, pi);
11   export '2-pi' = double_pi; // ExportEffect
12   return %undefined%;
13 }
```

Listing 3.11: A simple AranLang module that computes $2 \cdot \pi$

3.2.13 Explicit direct `eval` calls

In this section, we emphasize how AranLang makes direct calls to `eval` explicit.

The `%eval%` function is a dangerous and often misused feature of JavaScript. It evaluates dynamically generated strings as code. When `%eval%` is invoked in a direct eval call (i.e., a call expression where the callee is the identifier `eval`), the code executes within the caller's scope. In sloppy mode, it even permits the declaration of new variables in the caller's scope. Note that a direct eval call can only be determined at run-time, as the variable `eval` does not necessarily point to `%eval%`.

Despite its implications for security, maintainability, and performance, dynamic code evaluation remains an important aspect of JavaScript and should be supported in AranLang. In particular, direct calls to `eval` are represented as `EvalExpression` nodes in AranLang. The two major differences between these syntactic structures are:

- AranLang offers a clearer syntax than a direct eval call, as an `EvalExpression` node guarantees dynamic code evaluation, whereas the semantics of a direct eval call depend on the resolution value of the `eval` variable.
- The argument of an `EvalExpression` node is expected to be a valid `DeepLocalEvalProgram` node, while the argument of a direct eval call is expected to be a string corresponding to valid JavaScript code.

Note that the discussion from Section 3.2.6 regarding safe and lexical scoping requires a run-time mechanism to ensure that all free internal variables of the `DeepLocalEvalProgram` are declared within the caller's scope. This check should be performed as the initial step in evaluating the dynamic code, signaling unbound variables as early syntactic errors. However, in our implementation, this check was omitted because `DeepLocalEvalProgram` nodes are generated on an as-needed basis, and are guaranteed not access unscoped internal variables.

3.3 Transpilation from AranLang to JavaScript

While AranLang resembles JavaScript, it is not a strict subset and requires lightweight code transformations to become executable JavaScript. In this section, we qualitatively describe this phase, which we call retropilation. *The goal is to clarify the semantics of AranLang constructs that differ slightly from those in JavaScript. This section is organized as follows:*

- *Sections 3.3.1 to 3.3.7 detail the retropilation of several AranLang constructs to JavaScript.*

3.3.1 Evaluating intrinsic literals

In this section, we describe the necessity of a global registry for intrinsic literals and clarify the implementation of some custom intrinsic functions.

Transpiling intrinsic literals back to JavaScript requires a global registry of intrinsic values, as illustrated in Listing 3.12. The code is divided into two parts: the setup, which defines the intrinsic registry and its members, and the main section, which represents the actual output of the retropilation. The setup should be executed only once per JavaScript environment.

Most custom intrinsic functions, such as `%aran.performBinaryOperation%` defined at Lines 3–9, are straightforward to implement. However, functions that access the global scope, such as `%aran.readGlobalVariable%` defined at Lines 10–24, require special attention. These functions must query the global scope for arbitrary variables passed as string values, which generally necessitates a global call to `%eval%`. This approach is extremely slow and unacceptable for simply accessing a global variable. To optimize this process, we first create a registry of closures to access global variables, allowing `%eval%` to be called only once for each new variable queried. Second, we statically analyze the AranLang code to collect static strings passed to `%aran.readGlobalVariable%`, converting them into optimized arrows that bypass `%eval%` altogether.

```
"script";
{
  %aran.readGlobalVariable%("static_variable");
  %aran.readGlobalVariable%(%aran.performBinaryOperation%("+", "dyn", "var"));
  return 123;
}
```

⇓ ⇓ ⇓

```
// @ts-nocheck

// setup //
__intrinsic = {
  "aran.performBinaryOperation": (operator, left, right) => {
    switch (operator) {
      case "+": return left + right;
      // ...
    }
    throw new TypeError("Invalid operator");
  },
  "aran.readGlobalVariable": ((() => {
    const registry = {__proto__: null};
    const evalGobal = globalThis.eval;
    return (name, optimization) => {
      if (optimization) {
        return registry[name] = optimization;
      }
      if (!(name in registry)) {
        registry[name] = evalGobal(`(() => ${name});`);
      }
      return registry[name]();
    };
  }) ()),
  // ...
};
// main //
__intrinsic["readGlobalVariable"]("static_variable", () => static_variable);
__intrinsic["readGlobalVariable"]("static_variable");
__intrinsic["readGlobalVariable"](
  __intrinsic["aran.performBinaryOperation"]("+", "dyn", "var"),
);
123;
```

Listing 3.12: Retropilation of intrinsic literals

3.3.2 Initializing implicit bindings

In this section, we clarify how to assign appropriate values to AranLang's implicit bindings.

Three constructs in AranLang introduce implicit bindings:

- Programs: AranLang programs introduce several implicit bindings; however, most of their values already exist, such as `import.meta`, or are one-liner functions, such as `super.get`, which performs a simple lookup: `(k) => super[k]`. Only the values of scope-related implicit bindings, such as `scope.read`, are slightly more complex. They employ the same optimization technique as custom intrinsic functions that access the global scope, such as `%aran.readGlobalVariable%`.
- `try` statement: The only implicit binding introduced by AranLang's `try` statements is `catch.error`, which simply requires a protected name for the parameter of the `catch` clause.
- Closure generating expressions: The values of the implicit bindings introduced by closure-generating expressions already exist or are straightforward. The only exception is the value for `function.callee`, which requires wrapping the closure in an arrow function to enable it to reference itself from within its body. This is illustrated in Listing 3.13.

```
function {
  log("function.callee", function.callee);
  log("new.target", new.target);
  log("this", this);
  log("function.arguments", function.arguments);
  return null;
};
```

⇓ ⇓ ⇓

```

((() => {
  const __callee = function (...$function$arguments) {
    let $this = this;
    let $new$target = new.target;
    let $function$callee = __callee;
    log("function.callee", $function$callee);
    log("new.target", $new$target);
    log("this", $this);
    log("function.arguments", $function$arguments);
    return null;
  };
  return __callee;
}) ());
```

Listing 3.13: Initialization of the implicit bindings introduced by an AranLang function

3.3.3 Passing the explicit `this` argument

In this section, we clarify how to provide an explicit `this` argument in JavaScript.

Fortunately, the semantics for `ApplyExpression` in AranLang are already implemented by an existing intrinsic in JavaScript, `%Reflect.apply%`. Therefore, as illustrated in Listing 3.14, the retropilation of an `ApplyExpression` simply requires invoking this function from the intrinsic registry. If the `that` argument is omitted or, equivalently, is `%undefined%`, the retropiled code can perform a syntactic call and maintain its original structure.

```
f(that t, x1, x2);
g(y1, y2);
```

⇓ ⇓ ⇓

```
__intrinsic["Reflect.apply"](f, t, [x1, x2]);
g(y1, y2);
```

Listing 3.14: Retropilation of `ApplyExpression` with and without the `that` argument

3.3.4 Evaluating closure-generating expressions

> *In this section, we clarify how to implement the four types of closures in AranLang, with methods and generators being the most challenging.*

Retropiling closure generating expressions is straightforward for both `FunctionExpression` and `ArrowExpression`. `MethodExpression` requires wrapping in an object literal, as demonstrated in Listing 3.15.

```
(method () { return 123; });
```

⇓ ⇓ ⇓

```
({dummy () { return 123; }}.dummy);
```

Listing 3.15: Retropilation of method closure

`GeneratorExpression` is more complex to retropile. The body of generator closures is `GeneratorBlock`, which differs from `RoutineBlock` only in that it includes a list of effects between the declaration and the main body. This separation is necessary because these effects must be executed before the initial suspension of the generator, which occurs in JavaScript immediately after the formal parameters. Consequently, head effects should be retropiled within the parameters of the generator. Additionally, declarations must precede the head effects in the retropiled code, necessitating their presence in the parameters as well. Listing 3.16 illustrates how this can be achieved in a convoluted manner using object destructuring assignments. Note that `function.arguments` must be derived from `arguments`, whereas it is a rest parameter for other closures.

```
generator {
  let x; // declaration
  [x = %Reflect.get%(function.arguments, 0)]; // head
  yield x; yield x; return null; // body
}
```

⇓ ⇓ ⇓

```
(function* (
  ...{
    __declaration: {$function$arguments, x} = {
      $function$arguments: __instrinsic["Array.from"](arguments),
      x: __intrinsic.undefined,
    },
    __head: {} = (
      x = __intrinsic["Reflect.get"]($function$arguments, 0),
      {}
    ),
  }
) /* body */ { yield x; yield x; return null; });
```

Listing 3.16: Retropilation of generator closure

3.3.5 Managing simplified module imports and exports

In this section, we clarify how to implement AranLang's module system in JavaScript.

Retropiling modules requires defining a retropilation variable for each export and import declaration. As illustrated in Listing 3.17, import expressions are retropiled into a read from a hidden import variable, whereas export effects are retropiled into a write to a hidden export variable.

```
"module";
import 'pi' from './math';
export 'double-p';
{ export 'double-pi' = 2 * import 'pi' from './math'; }
```

⇓ ⇓ ⇓

```
import { "pi" as __pi } from "./math";
let __double_pi;
export { __double_pi as "double-pi" };
{ __double_pi = 2 * __pi; }
```

Listing 3.17: Retropilation of modules

3.3.6 Optimizing intrinsic call patterns

In this section, we outline an important optimization that involves recognizing patterns in AranLang to generate more efficient JavaScript code.

Direct calls to intrinsic functions can sometimes be optimized and may not actually require invoking the intrinsic function. This is made possible by intrinsic literals, which have a static value. Listing 3.18 demonstrates such optimization for binary operations and an array literal.

```
%aran.performBinaryOperation%("+", x, y);
%Array.of%(x1, x2, x3);
```

⇓ ⇓ ⇓

```
__intrinsic["aran.performBinaryOperation"]("+", x, y)
__intrinsic["Array.of"](x1, x2, x3)
```

Equivalent but slower than:

```
x + y;
[x1, x2, x3]
```

Listing 3.18: Optimization of calls to intrinsic functions

3.3.7 Resolving behavioral divergences of `eval`

As stated in Section 3.2.13, AranLang features a syntactic `eval` expression that differs from the direct call to `%eval%` in JavaScript in two ways: first, the semantics are clearer as they guarantee dynamic code evaluation; second, the argument is expected to be an AranLang Abstract Syntax Tree (AST) rather than a string. Listing 3.19 illustrates how these two divergences are resolved:

- First, at Lines 1 and 2, the value of `eval` is tested to ensure it corresponds to `%eval%`. If this is not the case, a syntactic error is thrown. Note that JavaScript code retropiled from AranLang always executes in strict mode, which prohibits assignments to the `eval` variable; thus, it is impossible to restore its value to `%eval%`.
- Second, the custom intrinsic function `aran.retropileEvalCode` is called to perform the retropilation. Similar to `aran.transpileEvalCode`, the default behavior of this function is to throw a syntactic error. To support `eval` calls, the analysis implementer must override this intrinsic in the registry and manually invoke our implementation. The objective of this design is to support both online and offline deployment of our implementation, which are both discussed in Section 4.2.2.

```
eval %aran.createObject%(
  null,
  "type", "Program",
  "kind", "eval",
  "situ", "local.deep",
  "head", %Array.of%(),
  "body", ..., // etc.
);
```

⇓ ⇓ ⇓

```
if (eval !== __intrinsic["eval"])
  throw new __intrinsic["SyntaxError"]("eval has been tampered with");
eval(__intrinsic["aran.retropileAranCode"]({
  type: "Program",
  kind: "eval",
  situ: "eval.deep",
  head: [],
  body: ... // etc.
}));
```

Listing 3.19: Retropilation of an AranLang `eval` expression

3.4 Transpilation from JavaScript to AranLang

In this section, we describe the process of transpiling JavaScript to AranLang in a semantically preserving manner, ensuring the transparency of the overall approach. This section does not aim to provide a comprehensive account of the transpilation process but rather to highlight the main challenges and solutions. For further reference, the complete transpilation process is available online in our prototype implementation, as presented in Section 4.2. This section is organized as follows:

- *Section 3.4.1 provides a brief overview of the transpilation process.*
- *Sections 3.4.2 to 3.4.12 illustrate the transpilation of various JavaScript constructs to AranLang.*

3.4.1 Overview of the transpilation process

In this section, we provide a brief overview of the transpilation process from JavaScript to AranLang. The goal is to help the reader understand how the transpilation cases presented in subsequent sections can be concretely implemented.

The transpilation of JavaScript programs to AranLang programs can be accomplished through Abstract Syntax Tree (AST) traversal. The input for the traversal includes the following information:

- The main input is the current ESTree node.
- General context: the type of the program (script, module, eval), the type of the closest parent closure (arrow function, plain function, etc.), the current JavaScript mode (strict or sloppy), class information (private fields), and more.
- A list-like structure to represent the frames of the current scope. Both static lexical frames and dynamic frames should be represented. Scope management is the most technically challenging aspect of the traversal.
- Information to generate unique transpilation variables. The goal is to avoid variable name clashes during the traversal of two sibling nodes. For instance, the current path of the node can be used to generate unique names, such as `root_body_1_tmp1` for a variable named `tmp1` at `root.body[1]`.

The output of the traversal includes the following information:

- The main output is the AranLang node, which encapsulates most of the semantics of the input ESTree node.

- Some syntactic elements escape the normal construction flow and must be hoisted to the beginning of blocks or even programs. For instance, the introduction of transpilation variables must be signaled so that they can be added to the declaration of the current block.
- If the main input is a statement node, it may contain variable declarations, and the visitor will have to return an updated version of the scope to indicate which variables have been initialized and are no longer in the temporal dead zone.

3.4.2 Transpilation of JavaScript's two modes

In this section, we describe how the two modes of JavaScript can be transpiled into AranLang.

JavaScript can be executed in two modes: *strict* mode and *sloppy* mode. In strict mode, certain failures throw exceptions, while in sloppy mode, these failures are ignored. For example, failing to assign a property of an object triggers an exception in strict mode but not in sloppy mode. Additionally, features such as the `with` statement are disabled in strict mode, as they are often considered legacy and can prevent JavaScript runtimes from performing important optimizations.

As many language constructs have distinct semantics in each mode, requiring the analysis to accommodate both modes would significantly complicate its implementation and reduce the expressiveness of our approach. Instead, this burden is entirely alleviated, as AranLang does not have modes; it is the transpilation from JavaScript to AranLang that accounts for mode-specific semantics. For instance, Listing 3.20 illustrates the transpilation of property assignments in each mode: in strict mode, the transpilation includes a conditional that throws a `TypeError` if the assignment fails, whereas in sloppy mode, the result is simply discarded.

```
// AranLang (sloppy mode) //
%Reflect.set%(obj, key, val);
// AranLang (strict mode) //
%Reflect.set%(obj, key, val)
  ? true
  : %aran.throwException%(new %TypeError%("cannot set property"))
```

Listing 3.20: Transpilation of `obj[key]=val` in both modes

3.4.3 Transpilation of dynamic frames

In this section, we outline how transpilation can account for the presence of dynamic frames in the environment.

Accurately representing the intricacies of JavaScript scope in AranLang is the most complex aspect of the transpilation process. This challenge primarily arises from the necessity to represent JavaScript's dynamic frames in AranLang, which is entirely lexically scoped. In JavaScript, there are four types of dynamic frames:

- The Global Object: It serves as the outermost frame in any JavaScript scope, containing intrinsic values and variables globally declared using the `var` or `function` keywords. It is represented as an object value that can be accessed through various means, including the `globalThis` global variable.

  ```
  var xyz = 123;
  globalThis.xyz; // 123
  globalThis.xyz = 456;
  xyz; // 456
  ```

- The Global Declarative Record (GDR): It always precedes the global object in any JavaScript scope and contains variables that are globally declared in script programs via the `let`, `const`, or `class` keywords. Unlike the global object, it is not reified and cannot be introspected.

```
let xyz = 123;
globalThis.xyz = 456;
xyz; // 123
```

- With Frames: Dynamic frames can be explicitly introduced using the `with` statement, which accepts an expression that serves as the last frame in the current scope.

```
{ let xyz = 123;
  with ({ xyz: 456 })
    xyz; // 456
}
```

- Eval Frames: In sloppy mode, a direct call to `%eval%` can dynamically declare variables in the calling scope using top-level `var` and `function` declarations. This feature effectively transforms the frame of the nearest closure into a dynamic frame.

```
(((xyz) => {
  eval("var xyz = 456"); // Direct eval call
  xyz; // 456
}) (123));
```

Transpilation of these four types of dynamic frames is performed similarly. As the JavaScript AST is traversed, the current scope is represented by a linked list of frames. Each time a variable is accessed, this structure is queried, and run-time checks are inserted for each encountered dynamic frame. For instance, Listing 3.21 illustrates the transpilation of the `with` statement. The value passed to `with` is stored in a transpilation variable, which is queried to determine whether variables accessed within the `with` statement should be intercepted by the dynamic frame. Note that, for simplicity, the transpilation is not exact, as it omits handling the case where the matched property is marked as unscopable[13].

```
let xyz = 123;
with ({ __proto__: null, xyz : 456 }) {
  xyz;
}
```

⇓ ⇓ ⇓

```
"script";
{
  let xyz;
  let __frame;
  xyz = 123;
  __frame = %aran.createObject%(null, "xyz", 456);
  {
    %Reflect.has%(__frame, "foo")
      ? %Reflect.get%(__frame, "foo")
      : xyz;
  }
}
```

Listing 3.21: Simplified transpilation of a `with` statement

3.4.4 Transpilation of the global declarative record

> *In this section, we explain how AranLang supports access to the native global declarative record and how this functionality can also be emulated.*

In Section 3.2.4, we discussed how global variables are accessed through custom intrinsic functions such as `%aran.readGlobalVariable%`. While this method is effective, it provides an opaque view of the global scope for the analysis. To enhance analysis of the global scope, one would ideally reify both the global object and the global declarative record (GDR). Unfortunately, reifying the latter is not possible. However, it is possible to transpile JavaScript programs into AranLang programs that emulate the GDR with a plain object. This approach forms the foundation of an alternative transpilation that we refer to as *mock GDR*; in contrast, the *real GDR* approach is the one we have been discussing so far and is the default in our implementation.

[13] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/unscopables`

Listing 3.22 illustrates the two transpilation approaches for accessing global variables. The real GDR approach is straightforward and utilizes `%aran.readGlobalVariable%`. The mock GDR approach first queries `%aran.global_declarative_record%`, which emulates the global declarative record, and falls back to `%aran.global_object%` which should refer to the global object. Although this second approach requires more transpilation expansion, it provides a more detailed view of the global scope for analysis.

```
%aran.readGlobalVariable%("xyz");
```

```
%Reflect.has%(%aran.global_declarative_record%, "xyz")
  ? %Reflect.get%(%aran.global_declarative_record%, "xyz")
  : %Reflect.has%(%aran.global_object%, "xyz")
    ? %Reflect.get%(%aran.global_object%, "xyz")
    : %aran.throwException%(new %ReferenceError%("Missing variable: xyz"));
```

Listing 3.22: Two transpilations of the global read `xyz;` for both real (top) and mock GDR (bottom)

Listing 3.23 illustrates the two transpilation approaches for the global declaration `let xyz=123`. The real GDR approach employs a combination of `ScriptHeader` and custom intrinsic functions. In contrast, the mock GDR approach only relies on standard intrinsic functions. It first checks if the variable is already declared in the GDR; if so, it throws an early syntax error. Otherwise, it sets the variable to `%aran.deadzone_symbol%` before reassigning it to the initialization value.

```
"script";
let xyz;
{
  %aran.writeGlobal%(xyz, 123);
}
```

⇓ ⇓ ⇓

```
"script";
{
  %Reflect.has%(%aran.global_declarative_record%, "xyz")
    ? %aran.throwException%(new %SyntaxError%("Duplicate variable: xyz"))
    : %Reflect.set%(%aran.global_declarative_record%, "xyz", %aran.deadzone_symbol%);
  %Reflect.set%(%aran.global_declarative_record%, "xyz");
}
```

Listing 3.23: Two transpilations of the global declaration `let xyz=123;` for both real (top) and mock GDR (bottom)

We now discuss the transparency implications of both approaches. Most importantly, the mock GDR introduces significant semantic discrepancies when faced with partial instrumentation. Indeed, this approach completely circumvents the real GDR, which ceases to serve as a communication channel between transpiled and subsequently retropiled JavaScript code and untouched JavaScript code. Note that comprehensive instrumentation is highly demanding, as there are numerous entry points for evaluating JavaScript code: direct and indirect calls to `eval`, dynamic function creation with `Function`, script tags in the browser, and calls to runtime-specific methods such as `runInContext` from `node:vm`, among others.

The real GDR approach introduces two manageable semantic discrepancies that can only be triggered by peer programs—i.e., code evaluated by the runtime originating from another source. First, the temporal dead zone of `let` and `const` global variables cannot be enforced on peer programs—an issue further discussed in Section 3.4.5. Second, globally declared `const` variables are transformed into globally declared `let` variables. While transpilation can enforce the immutability of these variables within the script, it cannot prevent peer programs from reassigning them. Addressing these two discrepancies would necessitate substantial architectural changes that could undermine the cohesion of our approach.

Table 3.8 summarizes the key distinctions between real GDR transpilation and mock GDR transpilation. The real GDR approach offers limited reasoning capacity regarding the global scope but is compatible with partial instrumentation; it introduces two manageable semantic discrepancies. In contrast, the mock GDR approach provides comprehensive and streamlined reasoning about the global scope but is incompatible with partial instrumentation. It is clear that these approaches are complementary, which is why Aran supports both.

	Real GDR Trans.	Mock GDR Trans.
Support for Analyzing the Global Scope	Limited	Comprehensive
Declaration (`var` & `function`)	`ScriptHeader`	`%Reflect.*%`
Declaration (`let` & `const`)	`ScriptHeader` or `%aran.declareGlobalVariable%`	`%Reflect.*%`
Access	`%aran.*GlobalVariable%`	`%Reflect.*%`
Transparency	- Mutable global `const` - No external TDZ	Compromised by partial instrumentation

Table 3.8: Key comparison points between real and mock GDR transpilation

3.4.5 Transpilation of the temporal dead zone

In this section, we describe how JavaScript's temporal dead zone can be transpiled into AranLang to preserve transparency.

In Section 3.2.5, we noted that variable declarations must occur at the beginning of blocks, which effectively eliminates the temporal dead zone (TDZ) from AranLang and ensures the safety of its lexical scope. We will now focus on representing JavaScript's TDZ in AranLang.

In many cases, it is possible to determine statically whether a variable reference occurs in the TDZ. For instance, Listing 3.24 features two accesses of a variable named `foo`: one occurs before its initialization and is in the TDZ, while the other occurs after its initialization and is safe. In this situation, the TDZ can be detected statically, and any forbidden access to `foo` should be replaced by code that throws a `ReferenceError`.

```
{
  xyz = 123; // 🆗 ReferenceError (Static TDZ)
  let xyz = 456;
  xyz = 789; // 🆗 (Initialized)
}
```

⇓ ⇓ ⇓

```
{
  let xyz = %aran.deadzone_symbol%;
  // 🆗 ReferenceError (Static TDZ)
  ( 123, // Initializer could have side effects
    %aran.throwException%(new %ReferenceError%("xyz in TDZ")));
  xyz = 456;
  xyz = 789; // 🆗 (Initialized)
}
```

Listing 3.24: Transpilation of static TDZ

However, variable references wrapped within a closure complicate the static detection of the TDZ. For instance, Listing 3.25 defines two arrow functions that both access `foo`: one is defined before the variable's initialization, and the other is defined afterward. While the second arrow function is safe because it cannot be called before `foo` is initialized, the first arrow function can be invoked either before or after the initialization of `foo`. This creates uncertainty that necessitates a run-time check for resolution. In AranLang, this check is implemented by assigning `%aran.deadzone_symbol%` as the initial value of variables associated with dynamic TDZ.

The only other instance of dynamic TDZ occurs within `switch` statements, where `case` clauses can declare `let` and `const` variables for the entire body of the `switch` statement. Since `case` clauses are not guaranteed to execute, this introduces uncertainties regarding variable initialization.

```
{
  const readUnsafe = () => xyz; // ⚠ ReferenceError (Dynamic TDZ)
  let xyz = 123;
  const readSafe = () => xyz;   // OK (Initialized)
}
```

⇓ ⇓ ⇓

```
{
  let xyz = %aran.deadzone_symbol%; let readUnsafe; let readSafe;
  readUnsafe = arrow {
    // ⚠ ReferenceError (Dynamic TDZ)
    return %aran.performBinaryOperation%("===", xyz, %aran.deadzone_symbol%)
      ? %aran.throwException%(new %ReferenceError%("xyz in TDZ"))
      : xyz;
  };
  xyz = 123;
  readSafe = arrow {
    return xyz; // OK (Initialized)
  };
}
```

Listing 3.25: Transpilation of dynamic TDZ

So far, we have only discussed the transpilation of TDZ for internal variables. The only external variables that can be declared with a TDZ are global variables of type `let` and `const`. In mock GDR transpilation, the TDZ can be globally enforced by setting the initial value of these variables to `%aran.deadzone_symbol%`. However, in real GDR transpilation, it is not possible to enforce the TDZ of global variables in programs originating from another source. For example, in Listing 3.26, real GDR transpilation cannot signal the violation of the TDZ for the variable `xyz` during dynamic code evaluation.

```
globalThis.eval("xyz;"); // OK ReferenceError (TDZ)
let xyz = 123;
```

Listing 3.26: Global TDZ across different program sources

3.4.6 Transpilation of object literals

> *In this section, we outline some of the technical challenges associated with transpiling features related to JavaScript object literals in AranLang.*

Over time, JavaScript has extended object literals with features such as custom prototypes, accessor properties, method definitions, and spread properties. All these features can be expressed by creating an empty object with `%Object.create%` and then chaining calls to `%Reflect.defineProperty%`. However, this approach introduces significant transpilation expansion, as each property must be represented as a descriptor[14], which is itself an object.

To more concisely transpile simpler cases, such as the one in Listing 3.27, we introduced the custom intrinsic function `%aran.createObject%`. Unlike alternative object creation methods, such as `%Object.fromEntries%`, it does not require wrapping the arguments in another object. Instead, it takes a prototype followed by a flattened list of key-value pairs. This approach prioritizes source-adjacency over cohesion, as the analysis implementer must account for an additional custom intrinsic function. However, given that object literals are pervasive in both the original JavaScript and the transpilation expansion, we believe this tradeoff is justified.

Listing 3.27 illustrates the transpilation of an object literal that utilizes the `super` keyword to syntactically access its prototype.

[14] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty`

```
const object = {
  __proto__: makePrototype(),
  key: 123,
  getSuperKey () {
    return super.key;
  }
};
```

⇓ ⇓ ⇓

```
{
  let object; let __prototype;
  __prototype = makePrototype();
  object = %aran.createObject%(
    __prototype,
    "key", 123,
    "getSuperKey", method {
      return %aran.getValueProperty%(__prototype, "key");
    },
  );
}
```

Listing 3.27: Transpilation of a simple object literal

3.4.7 Transpilation of classes

In this section, we outline how JavaScript classes can be transpiled into AranLang, with a focus on private fields.

Classes are an important feature added in ECMAScript 2015. They provide a more ergonomic way to define constructor functions, properties, and methods. However, the class syntax is merely syntactic sugar over the existing prototype-based inheritance. This is precisely the type of feature that AranLang was designed to lower in order to enhance the expressiveness of Aran.

Listing 3.28 illustrates the transpilation of a simple class definition named `Person`, which includes an instance field `name`, a class field `species`, and an instance method `greet`. The key aspect of the transpilation is to represent the class as its constructor function. In this case, the default constructor is explicitly defined from Lines 3–9. The `name` instance field is defined as a property of the newly created instance at Line 6. The `species` class field is defined as a property of the constructor at Line 10. The `greet` instance method is defined from Lines 11–20 as a property of the `prototype` property of the constructor.

A potentially confusing aspect of the transpilation in Listing 3.28 is that the prototype of the newly created instance is derived from `new.target` rather than being statically obtained from the constructor function. While `new.target` refers to the constructor when it is invoked normally, using `Reflect.construct` allows for an alternative value to be provided to `new.target`, which necessitates the dynamic retrieval of the prototype.

Several class-related features complicate transpilation: computed keys, derived classes, static blocks, and private fields. Among these, the most challenging feature to transpile transparently is private fields, which are declared by prefixing a key with the `#` character. To maintain privacy, these fields must be stored separately from instances and classes. Weak collections, introduced in ECMAScript 2015, are particularly suitable for this purpose as they help prevent memory leaks. Listing 3.29 demonstrates how a class named `Person`, defined with a private field `#password`, can be transpiled by introducing a transpilation variable `__password_registry` that associates each instance of `Person` with its private `#password` value.

```
{
  class Person {
    name = "Alice";
    static species = "Human";
    greet () { return `Hello, my name is ${this.name}`; }
  }
}
```

⇓ ⇓ ⇓

```
{
  let C; let __constructor;
  __constructor = function {
    this = %aran.createObject%(
      %aran.getValueProperty%(new.target, "prototype"),
      "name", "Alice",
    );
    return this;
  }
  %Reflect.set%(__constructor, "species", "Human");
  %Reflect.set%(
    %aran.getValueProperty%(__constructor, "prototype"),
    "greet",
    method {
      return %aran.performBinaryOperation%(
        "Hello, my name is ",
        %aran.getValueProperty%(this, "name"),
      );
    }
  );
  C = __constructor;
}
```

Listing 3.28: Simplified transpilation of a basic class definition

```
class Person {
  #password = "****";
  login (attempt) {
    return this.#password === attempt;
  }
}
```

⇓ ⇓ ⇓

```
{
  let C; let __constructor; let __password_registry;
  __password_registry = new %WeakMap%();
  __constructor = function () {
    this = %aran.createObject%(%aran.getValueProperty%(new.target, "prototype"));
    %WeakMap.prototype.set%(that __password_registry, this, "****");
    return this;
  };
  %Reflect.set%(
    %Reflect.get%(__constructor, "prototype"),
    "login",
    method {
      return %aran.performBinaryOperation%(
        "===",
        %WeakMap.prototype.has%(that __password_registry, this)
          ? %WeakMap.prototype.get%(that __password_registry, this)
          : %aran.throwException%(new %TypeError%("Cannot read #password")),
        %aran.getValueProperty%(function.arguments, 0),
    },
  );
  C = __constructor;
}
```

Listing 3.29: Transpilation of a private instance field

3.4.8 Transpilation of the iterator protocol

In this section, we outline how the JavaScript iterator protocol can be implemented in AranLang.

In JavaScript, the iterator protocol defines a standard way to access the elements of a collection sequentially[15]. To be *iterable*, an object must have a method at `%Symbol.iterator%` that returns an iterator. An *iterator* is an object with a method `next`, which returns an object containing two properties: `value`, representing the current value, and `done`, a boolean indicating whether there are still elements to iterate over. Three important language constructs rely on the iterator protocol: `for ... of` loops, array destructuring assignments, and the spread syntax. Through their lowering, these features provide a compelling opportunity to enhance the cohesion of our approach with minimal impact on source-adjacency.

Listing 3.30 illustrates how to transpile array destructuring assignments in AranLang by explicitly implementing the underlying iterator protocol. The transpilation involves introducing five variables: `__iterable` for the iterable, `__iterator` for the iterator, `__next` for the next method, `__step` for the last result of the next method, and `__done` to indicate the end of the iteration. The transpilation utilizes the custom intrinsic function `%aran.listIteratorRest%`, which collects the remaining elements of the iterator into an array. While this could have been achieved inline with a loop, as discussed in Section 3.2.3, we believe that custom intrinsic functions are justified when the transpilation expansion involves loops in order to preserve source-adjacency.

```
let [head, ...tail] = array;
```

⇓ ⇓ ⇓

```
{
  let head, tail, __iterable, __iterator, __next, __step, __done;
  __iterable = array;
  __iterator = %Reflect.get%(__iterable, %Symbol.iterator%)(that __iterable);
  __next = %Reflect.get%(__iterator, "next");
  __step = __next(that __iterator);
  __done = %Reflect.get%(__step, "done");
  head = __done ? %undefined% : %Reflect.get%(__step, "value");
  tail = __done ? [] : %aran.listIteratorRest%(__iterator, __next);
}
```

Listing 3.30: Simplified transpilation of array destructuring assignments

Note that the iterator protocol mandates that the `return` method should always be called on the iterator when the iteration is complete; this allows the iterator to release any resources it might have acquired. In Listing 3.30, the `return` method is invoked within `%aran.listIteratorRest%`; in the absence of a rest element, it is called inline. However, in both cases, it will not be executed if the iterator terminates abruptly due to an exception. We currently do not address this semantic discrepancy because intercepting exceptions in an expression context is challenging.

3.4.9 Transpilation of the `arguments` object

In this section, we outline the challenges associated with constructing the JavaScript `arguments` object in AranLang.

In JavaScript, the `arguments` object is an array-like object accessible within non-arrow functions. It provides access to the arguments passed to the function, regardless of the formal parameters. Each index of the `arguments` object corresponds to an argument, and its `length` property reflects the number of arguments passed.

In sloppy mode, the `arguments` object is two-way bound to the function's parameters. This means that modifying a parameter's value will also change the corresponding property of the `arguments` object, and vice versa. However, as illustrated in Listing 3.31, this behavior is not preserved in AranLang after

[15] `https://developer.mozilla.org/docs/Web/JavaScript/Iteration_protocols`

transpilation, as it relies solely on `%aran.toArgumentList%` to create the `arguments` object. This custom intrinsic function defines all relevant properties: indexes, `length`, `Symbol.iterator`, and `callee`, but it cannot replicate the two-way binding with parameters.

```
function f (x0, x1) {
  arguments[0] = 123;
  log(x0);           // 123
  x1 = 456;
  log(arguments[1]); // 456
}
f("arg0", "arg1");
```

⇓ ⇓ ⇓

```
f = function {
  let arguments; x0, x1;
  __arguments = %aran.toArgumentList%(function.arguments, "sloppy");
  x0 = %Reflect.get%(function.arguments, 0);
  x1 = %Reflect.get%(function.arguments, 1);
  %Reflect.set%(__arguments, 0, 123);
  log(x0);                              // ✗ "arg0" (should be 123)
  x1 = 456;
  log(%Reflect.get%(__arguments, 1)); // ✗ "arg1" (should be 456);
  return %undefined%;
}
f("arg0", "arg1");
```

Listing 3.31: Transpilation of the `arguments` object which is not fully transparent

In a previous iteration of Aran, we addressed this semantic discrepancy by wrapping the `arguments` object in a proxy to update parameters upon modification. The reverse linkage was established by updating the `arguments` object whenever the parameter changed. However, the transpilation expansion was complex and involved a loop to populate the indices of `arguments`. While we prioritize transparency, we believe that the trade-off for source-adjacency in this case was justified, as two-way binding of the `arguments` object is considered legacy.

3.4.10 Transpilation of closure properties

In this section, we outline the challenges of constructing closures in AranLang that possess the same properties as those in JavaScript.

In JavaScript, closures possess several properties, which are summarized in Table 3.9. In AranLang, only the `prototype` property is retained in `FunctionExpression` and `GeneratorExpression` because it is a non-configurable property that cannot be removed. To ensure transparency, other properties must be added during transpilation. Listing 3.32 demonstrates the transpilation of the `name` and `length` properties. For simplicity, `%Reflect.set%` is used, but full transparency requires using `%Reflect.defineProperty%` instead.

Property	Description	Applicable
`prototype`	the prototype it should use when called as a constructor	- plain functions - constructors - generators
`name`	its name, based on its declaration	all
`length`	its arity, based on its parameters	all
`arguments`	the last `arguments` object it created	plain functions in sloppy mode
`caller`	the last function that called it	plain functions in sloppy mode

Table 3.9: Properties of function instances in JavaScript

While minimizing closure properties requires some transpilation expansion and affects source-adjacency, it enhances cohesion by enabling the analysis implementer to reason about the provenance of these properties in a manner similar to primitive literals. This approach is particularly justified for the `length` and `name` properties, as they provide a lightweight introspection mechanism that is likely to be disrupted

```
let square = (x) => x * x;
```

⇓ ⇓ ⇓

```
{
  let square;
  square = arrow () {
    let x;
    x = %Reflect.get%(function.arguments, 0);
    return %aran.performBinaryOperation%("*", x, x);
  }
  %Reflect.set%(square, "length", 1);
  %Reflect.set%(square, "name", "square");
}

%Reflect.defineProperty%(__function, "length", %aran.createObject%(
  null,
  "value", 1,
  "writable", false,
  "enumerable", false,
  "configurable", true,
));
%Reflect.defineProperty%(__function, "name", %aran.createObject%(
  null,
  "value", "square",
  "writable", false,
  "enumerable", false,
  "configurable", true,
));
```

Listing 3.32: Simplified transpilation of `name` and `length` properties

by instrumentation and would require updates to restore transparency. For instance, when a closure is directly assigned to a variable, its name corresponds to that variable. However, in the presence of instrumentation and identifier mangling, the name matches the mangled variable name instead.

JavaScript provides a more heavyweight introspection mechanism for closures: the intrinsic method `%Function.prototype.toString%`, which returns the source code of the closure as a string. Achieving transparency would require returning the original source code rather than the instrumented version. This could be accomplished by intercepting calls to the intrinsic method and replacing the result with the original source code. However, the ECMAScript specification does not actually define the format of the source code. It only stipulates that if two closures share the same source code, `%Function.prototype.toString%` should return the same string value. Consequently, this introspection mechanism is not guaranteed to be stable across different JavaScript engines and is best suited for debugging purposes. For this reason, Aran does not attempt to restore the transparency of `%Function.prototype.toString%`.

3.4.11 Transpilation of completion values and return statements

In this section, we outline how return statements in closures and completion values in programs can be transpiled into AranLang.

In JavaScript, the result of evaluating a program is determined by the value of its last value-generating statement, such as an expression statement. While this rule is conceptually straightforward, it can lead to intricate cases, particularly when the completion value arises within a loop or a `try` statement. AranLang eliminates this complexity by requiring programs to explicitly define their result using a return statement at the end of their body. Listing 3.33 illustrates how the transpilation of a program's completion value works in a simple case.

AranLang mandates that function bodies conclude with a return statement, which may appear only at this location. Listing 3.34 illustrates how multiple return statements can be transpiled by introducing a transpilation label, enabling the control flow to jump to the end of the function.

```
{ 123; debugger; }
```

⇓ ⇓ ⇓

```
"script"; {
  let __completion;
  __completion = 123;
  debugger;
  return __completion;
}
```

Listing 3.33: Transpilation of the completion value of programs

```
const compare = (x, y) => {
  if (x < y) return -1;
  if (x > y) return 1;
  return 0;
};
```

⇓ ⇓ ⇓

```
{
  let compare;
  compare = arrow {
    let x; let y; let __result;
    x = %Reflect.get%(function.arguments, 0);
    y = %Reflect.get%(function.arguments, 1);
    __return: {
      if (%aran.performBinaryOperation%("<", x, y)) {
        __result = -1;
        break __return;
      } else {}
      if (%aran.performBinaryOperation%(">", x, y)) {
        __result =  1;
        break __return;
      } else {}
      __result = 0;
      break __return;
    }
    return __result;
  };
}
```

Listing 3.34: Transpilation of a function featuring multiple return statements

3.4.12 Transpilation of direct `eval` calls

In this section, we outline how direct calls to `%eval%` can be transpiled into AranLang.

When `%eval%` is invoked in a direct call, the code executes within the caller's context. To support this feature, AranLang provides a dedicated syntactic construct, which was already presented in Section 3.2.13. Unlike direct calls to `%eval%`, which expect a code string, this construct expects an object representing an AranLang AST. To perform this conversion, the transpilation process calls the custom intrinsic `%aran.transpileEvalCode%`, which takes the code string as its first argument and a second argument called "situ" containing contextual information, such as the current scope and JavaScript mode.

Listing 3.35 illustrates the transpilation of a direct call to `%eval%` where the argument describes a `with` statement, which is prohibited in strict mode. Because the current mode is passed via the "situ" argument to `%aran.transpileEvalCode%`, the intrinsic has sufficient context to generate the appropriate AranLang AST, which in this case should simply throw a syntactic error.

```
"use strict";
eval("with () {}"); // 🆗 SyntaxError (with prohibited in strict mode)
```

⇓ ⇓ ⇓

```
"script";
{
  // 🆗 SyntaxError (during transpilation)
  return eval %aran.transpileEvalCode%(
    "with () {}", // Original JavaScript code
    "{mode: \"strict\", ...}", // Situ (incomplete)
  );
}
```

Listing 3.35: Direct calls to `%eval%` require transitive transpilation

As stated in Section 3.2.13, the default behavior of `%aran.transpileEval-Code%` is to throw an exception. To actually support direct `%eval%` calls, the analysis implementer is responsible for overriding this function and transpiling JavaScript code into AranLang AST by calling functions exported from our implementation. This architecture enables support for both online and offline deployment of the implementation, which are discussed in Section 4.2.2.

3.5 Conclusion

In this chapter, we presented AranLang, a core variant of JavaScript. First, we formally defined its syntax. Next, we qualitatively described its semantics by explaining how it should be retropiled into JavaScript. Finally, we demonstrated how AranLang can express some of the most challenging syntactic constructs of JavaScript. To conclude, we summarize the results and limitations, outline the main contributions, review related work, and highlight future research directions.

Assessment of design objectives

Below, we qualitatively assess how AranLang fulfills the design objectives outlined in Section 3.1.2:

- Minimalism: In Section 3.2, we introduced both an abstract and a concrete grammar for AranLang. Table 3.10 illustrates the rough equivalence between nodes from JavaScript and those from AranLang. The only AST node introduced by AranLang is `IntrinsicExpression`, while 43 out of 75 ESTree nodes have been removed, resulting in a reduction of 57%. Furthermore, AranLang nodes are generally simpler than their ESTree counterparts—e.g., closures do not have parameters, and write operations do not return a value. The two exceptions to this rule are programs, which are differentiated based on their intended execution context, and blocks, which can feature labels, head declarations, and a tail return statement.

- JavaScript Coverage: In Section 3.4, we discussed the process of transpiling JavaScript to AranLang and highlighted several cases where preserving JavaScript semantics is difficult. Most of these cases, such as code reification and two-way bound `arguments` objects, can be resolved, albeit at the cost of fairly convoluted transpilation expansion. The only JavaScript semantics that cannot be represented in AranLang concern externally declared variables: it is impossible to enforce their temporal dead zone and immutability inside programs originating from another source. This is the sole instance where we prioritized minimalism over JavaScript coverage.
- JavaScript Proximity: Overall, we prioritized minimalism over proximity to JavaScript, as we believe that the expressiveness of our approach is more important than its source-adjacency. The only exceptions are a few non-essential intrinsic functions, such as `aran.getValueProperty`, due to their pervasive usage, and the retention of coroutines, which requires significant transpilation expansion.

The design decisions of AranLang can be summarized as follows. We consistently prioritize JavaScript coverage, as the transparency of our approach is paramount. We favor minimalism over JavaScript proximity because we consider the expressiveness of our approach to be more important than its source-adjacency. Finally, we align with JavaScript whenever possible, without compromising the other criteria, to enhance the source-adjacency of our approach. We believe this strategy positions AranLang as a strong contender for simplifying the instrumentation of JavaScript programs.

Main contributions

The main contribution of this chapter is technical, providing a comprehensive repertoire of transpilation techniques to lower complex ECMAScript 2025 constructs into simpler constructs defined in a carefully designed core variant, AranLang. This technical contribution is validated in the next chapter through two experiments: semantic transparency is evaluated by testing against Test262, and performance transparency is evaluated by benchmarking against Octane.

Related work

We have identified three main reasons for transpiling a programming language into a simpler core variant: to enhance compatibility, to simplify implementation, and to simplify program analysis. Our motivation falls into the third category. Next, we briefly review transpilation in the context of all three motivations.

Compatibility-oriented transpilation Compatibility transpilers aim to enable modern language features to run on legacy language runtimes. For JavaScript, Babel[16] has emerged as the de facto standard. Although compatibility transpilers are technically similar to simplification transpilers, their primary goal is compatibility rather than facilitating program analysis or code transformation. This results in significantly different design decisions. For example, Babel transpiles away coroutines but not increment operations, whereas our approach does the opposite.

Implementation-oriented transpilation Language implementers have also recently shown interest in simplifying JavaScript. The rapid introduction of new features has made it challenging for them to provide ECMAScript-compliant runtimes that remain secure and performant. This issue was raised in the TC39 meeting in October 2024[17], where Shu-yu Guo, an engineer from Google, proposed essentially dividing JavaScript into two parts[18]: JS0, which would consist of a core subset of stable JavaScript features, and JSSugar, which would include fast-evolving and non-essential features that can be expressed in terms of JS0.

[16] `https://babeljs.io`
[17] `https://github.com/tc39/agendas/blob/main/2024/10.md`
[18] `https://docs.google.com/presentation/d/1ylROTu3N6MyHzNzWJXQAc7Bo1O0FHO3lNKfQMfPOA4o/edit#slide=id.p`

ESTree	AranTree
Program	
Program	ModuleProgram ScriptProgram GlobalEvalProgram DeepLocalEvalProgram RootLocalEvalProgram
Block	
BlockStatement	BlockStatement SegmentBlock RoutineBlock GeneratorBlock
Statement	
VariableDeclaration VariableDeclarator	Declaration ExternalDeclaration
EmptyStatement	
ExpressionStatement	EffectStatement ExpressionEffect
DebuggerStatement	DebuggerStatement
ReturnStatement	
ThrowStatement	
BreakStatement	BreakStatement
ContinueStatement	
LabeledStatement	
TryStatement CatchClause	TryStatement
IfStatement	IfStatement
SwitchStatement SwitchCase	
WhileStatement	WhileStatement
DoWhileStatement	
ForStatement	
ForInStatement	
ForOfStatement	
WithStatement	
Pattern & Class	
ObjectPattern AssignmentProperty ArrayPattern AssignmentPattern RestElement	
ClassExpression ClassDeclaration ClassBody MethodDefinition PropertyDefinition StaticBlock PrivateIdentifier	

ESTree	AranTree
Expression	
	IntrinsicExpression
Identifier	ReadExpression
Literal BigIntLiteral	PrimitiveExpression
RegExpLiteral	
TemplateLiteral TemplateElement	
ThisExpression	
Super	
MetaProperty	
ArrayExpression	
ObjectExpression Property	
FunctionExpression FunctionDeclaration	FunctionExpression GeneratorExpression
ArrowFunctionExpression	ArrowExpression
UnaryExpression	
UpdateExpression	
BinaryExpression	
AssignmentExpression	WriteEffect
LogicalExpression	
MemberExpression	
ConditionalExpression	ConditionalExpression ConditionalEffect
CallExpression	ApplyExpression EvalExpression
NewExpression	ConstructExpression
TaggedTemplateExpression	
SequenceExpression	SequenceExpression
YieldExpression	YieldExpression
AwaitExpression	AwaitExpression
SpreadElement	
ChainExpression ChainElement	
ImportExpression	
Module	
ImportDeclaration ImportSpecifier ImportDefaultSpecifier ImportNamespaceSpecifier	ImportHeader ImportExpression
ExportNamedDeclaration ExportSpecifier ExportDefaultDeclaration ExportAllDeclaration	ExportHeader AggregateHeader ExportEffect

Table 3.10: Rough equivalence between ESTree and AranTree

Analysis-oriented transpilation The research community has explicitly pursued the simplification of programming languages for program analysis and code transformation. A seminal work in this context is the CIL [81] framework, which compiles arbitrary C programs into a significantly reduced subset of C. This approach has also been explored for managed languages, with frameworks such as λ_{JS}[46] and λ_{π}[93], which respectively target JavaScript and Python. However, λ_{JS} targets pre-Harmony JavaScript (i.e., ECMAScript 5.1), whereas our work provides full support for ECMAScript 2025. To the best of our knowledge, our work is the first simplification framework to support post-Harmony JavaScript. This distinction is critical: the language has evolved drastically since the Harmony era, introducing high-complexity features such as block-scoped variables, modules, coroutines, and classes.

Directions for future research

In addition to exploring transpilation to core variants for other managed languages, we believe it is worthwhile to investigate how this process can simplify static analysis. While this dissertation focuses on dynamic program analysis, it is important to note that syntactic sugar also complicates static program analysis. In the future, it will be interesting to investigate how much easier it is to formally reason about AranLang programs compared to JavaScript programs.

Discussion: Applicability to other managed languages

While the JavaScript specification has adopted new features at an impressive rate, other managed languages have also demonstrated significant growth. Due to backward compatibility requirements and the engineering complexity of maintaining a runtime compliant with specifications, these new features often serve as syntactic sugar, making them suitable candidates for removal from the core variant. Below, we briefly speculate how transpilation into a core variant could potentially eliminate the incidental syntactic complexity present in managed languages other than JavaScript:

- Python: Python has an execution model that shares many similarities with JavaScript, and many of the transpilation techniques described in this chapter can be adapted for Python. This is effectively what has been explored in the λ_{π} [93] semantic framework. For instance, similar to JavaScript, Python implements objects as inheritance-based dictionaries, where methods are closures with a hidden parameter for passing the receiver. This allows for the replacement of syntactic class definitions with a call to the `type` metaclass, which dynamically creates a new class based on its name, its parent, and a dictionary of values that define methods and class attributes.

- Smalltalk: Smalltalk is a pure object-oriented language in which every value, including classes and control structures, is an object. Although Smalltalk is renowned for its minimal syntax, it still includes some syntactic sugar, such as literals for creating collections and the cascade operator for chaining method invocations. Method and block definitions present a more intriguing case for desugaring. In principle, these syntactic definitions can be replaced by a call to `compile:`, which accepts the source code as a string argument. However, this low-level method breaks lexical scoping and accesses the call stack to capture free variables in the scope of the caller. Hence, it is unclear whether such a transformation would actually be beneficial for the analysis. To summarize, Smalltalk has minimal syntactic sugar, which makes it a less suitable candidate for investigating transpilation into a core variant.

- Ruby: Ruby is largely inspired by Smalltalk and inherits its dynamic and reflective nature. However, to make it more appealing to developers, Ruby offers significantly more syntactic sugar. This includes attribute accessors, which serve as shorthand for manually defined setters and getters; syntactic class definitions, which can be emulated by a call to `Class.new`; and operators, which are method invocations. This situation is somewhat similar to how JavaScript was inspired by Scheme. In the same way that AranLang aims to restore the core Scheme-inspired simplicity of JavaScript, it would be interesting to investigate how a core variant of Ruby could restore the core Smalltalk-inspired simplicity of Ruby.

- Java: Because Java is statically compiled, its potential for reflection and dynamic behavior is significantly lower than that of JavaScript and the other managed languages mentioned above.

In particular, most object-oriented features, such as class names, subclassing, and method and attribute definitions, cannot be expressed in terms of calls to reflective functions. Nonetheless, Java includes syntactic sugar, such as string concatenation with `+`, which utilizes a string builder under the hood, and implicit primitive boxing. However, these constructs are straightforward to desugar, and we believe they should be regarded as mere annoyances rather than significant cognitive load for the analysis implementer. Moreover, Java is compiled to JVM bytecode, which already provides a cohesive interface for instrumentation, albeit at the cost of source-adjacency since it operates at the level of a stack machine. For these reasons, we believe that transpilation of Java into a core variant would not be as successful as in JavaScript for eliminating incidental syntactic complexities.

- C#: We believe C# presents a case similar to that of Java regarding transpilation into core variants. As a statically typed language compiled into bytecode with native support for classes, its object-oriented syntax is unlikely to be easily desugared. Most syntactic sugar, such as automatic properties, collection initializers, and LINQ query expressions, can be easily expressed in terms of method invocations and loops. Consequently, like Java, we believe that C# does not provide substantial opportunities for eliminating incidental syntactic complexities.

Chapter 4

Aran: instrumentation of JavaScript programs

In Chapter 3, we introduced AranLang, a core variant of JavaScript designed to render the analysis of JavaScript programs more cohesive. In this chapter, we first describe an aspect-oriented interface for defining analyses. We then detail Aran, which is the prototype implementation of our approach and serves to evaluate it both qualitatively and quantitatively. This chapter is organized as follows:

- *Section 4.1 describes a join point model for AranLang specifically designed for dynamic program analysis.*
- *Section 4.2 outlines our prototype, Aran.*
- *Section 4.3 presents a first experiment involving the deployment of Aran on Test262, the official conformance test suite for ECMAScript.*
- *Section 4.4 discusses a second experiment involving the deployment of Aran on Octane, a benchmark suite developed by Google.*
- *Section 4.5 concludes the chapter by summarizing the results and discussing related work.*

4.1 Weaving analysis logic into AranLang

While AranLang is suitable for directly conducting instrumentation through recursive traversal of the AST, greater reusability can be achieved by providing an API for inserting analysis logic into AranLang programs. Aspect-oriented programming [37, 60] offers a well-established methodology for this endeavor. In this section, we introduce an aspect-oriented API for integrating analysis logic into AranLang programs. This section is organized as follows:

- *Section 4.1.1 outlines the standard join point model for AranLang.*
- *Section 4.1.2 briefly presents each join point.*
- *Section 4.1.3 demonstrates how our join point model effectively supports shadow execution.*

4.1.1 Overview of the standard join point model

In this section, we outline our standard join point model for AranLang, which is designed to provide comprehensive introspection and intercession capabilities.

In aspect-oriented programming [60, 37], behaviors collectively referred to as *advice* are inserted at execution points known as *join points*, based on a specification called a *pointcut*. This process is termed *weaving*. The pairing of an advice and a pointcut is referred to as an *aspect*.

The standard join point model of AranLang is designed to provide comprehensive introspection and intercession capabilities. The naming convention for join points follows the usual aspect-oriented terminology: the first part of the name describes the targeted AST node, while the second part indicates the location where the join point is inserted within the AST node. For instance, `read@after` represents the join point that is inserted immediately after a `ReadExpression`.

The first argument of each advice is a state that can be localized to the current block via the `block@setup` advice, which receives the state of the parent block and returns the state for the current block. This argument aids in state management and is particularly useful for recovering a consistent state after hiatuses in control flow, such as those caused by `yield` and `await` expressions.

Listing 4.1 demonstrates how local state management in Aran enables scope analysis with minimal logic. The initial root state is set to `null` by default. At Line 5, upon entering a block, a new frame is created by prototype extension of the parent scope. At Lines 6–7, right after `block@setup`, the newly created frame is populated with the variables of the current block, initializing them with two counters. At Lines 8–9, the `read` and `write` counters are updated when advising a read expression or a write effect, respectively. Finally, at Line 10, the counters for each variable of the current block are logged upon exiting the block.

```
1  const init = (variable) => [variable, ({ read: 0, write: 0 })];
2  const report = ([variable, { read, write }]) =>
3    console.log(`${variable} was read ${read} times and written ${write} times`);
4  export const advice = {
5    "block@setup": (scope, _kind, _location) => ({ __proto__: scope }),
6    "block@declaration": (scope, _kind, frame, _location) =>
7      Object.assign(scope, Object.fromEntries(Object.keys(frame).map(init))),
8    "write@before": (scope, variable, value, _location) => (scope[variable].write++, value),
9    "read@after": (scope, variable, value, _location) => (scope[variable].read++, value),
10   "block@teardown": (scope, _kind, _location) => Object.entries(scope).forEach(report),
11 };
```

Listing 4.1: A simple scope analysis expressed as an Aran standard advice for Aran

The last argument of each advice (named `location` in Listing 4.1) is a hash digest of the JavaScript AST node from which the current join point originates. This hash digest must be unique across the entire ESTree program and is computed using three pieces of information: the ESTree node itself, the JSON path of the ESTree node, and the file path. The hash argument allows for the transmission of static information to the analysis, which can be customized by defining a digest function in the configuration. By default, the hash is set to the JSON path of the node, which is guaranteed to be unique. Alternatively, the combination of the node and its location also guarantees uniqueness. Hashes can also be numbers, which may provide performance benefits.

```
1  // String Hash //
2  export const digestDefault = (_node, json_path, _file_path) => json_path
3  export const digestLocation = (node, _json_path, _file_path) =>
4    `${node.type}@${node.loc.start.line}:${node.loc.start.column}`;
5  // Number Hash //
6  let nodes = [];
7  export const digestIndex = (node, _json_path, _file_path) => {
8    nodes.push(node);
9    return nodes.length - 1;
10 };
```

Listing 4.2: Three valid examples of digest functions

The standard join point model supports multiple formats for the pointcut:

- Boolean: Matches either all join points or none.
- Iterator: Enumerates all join point names to be matched.
- Object: Maps join point names to either a constant boolean value or a predicate function that receives the static information of the current join point.
- Closure: Determines matching based on the current reified join point.

While the standard join point model is straightforward and intuitive, it lacks flexibility. For example, each join point can be advised only once, and only the hash parameter can be customized; the other parameters are hard-coded within the model. These limitations have been addressed in a more flexible join point model which remains experimental and is available online[1].

4.1.2 Description of the standard join point interface

> *In this section, we provide a comprehensive description of our standard join point model for AranLang. The goal is to serve as a reference for understanding the analyses presented in the remainder of this dissertation.*

The complete standard join point model is detailed in Listings 4.3 and 4.4 as a TypeScript type declaration, comprising 31 join points. Each advice function receives the current local state as the first parameter (which is parameterized by the type variable `X`) and the hash digest of the current AST node as the last parameter (which is parameterized by the type variable `H`).

```ts
// Import //
import type { DeepLocalSitu } from "./situ.ts";
import type { Header } from "./syntax.ts"
// Kind //
type TestKind = "if" | "while" | "conditional";
type ProgramKind = "module" |"script"
  | "global-eval" | "root-local-eval" | "deep-local-eval";
type ArrowKind = "arrow" | "async-arrow";
type FunctionKind = "function" | "async-function";
type GeneratorKind = "generator" | "async-generator";
type MethodKind = "method" | "async-method";
type SegmentKind = "bare" | "while"
  | "try" | "catch" | "finally"
  | "then" | "else";
type ClosureKind = ArrowKind | FunctionKind | GeneratorKind | MethodKind;
type RoutineKind = ProgramKind | ClosureKind;
type ControlKind = RoutineKind | SegmentKind;
// Synonym //
type Intrinsic = string;
type Source = string;
type Specifier = string;
type Label = string;
type Variable = string;
// Value //
type ScopeValue = unknown; // Environment Value
type StackValue = unknown; // Kontinuation/Stack Value
type OtherValue = unknown; // Other Value
// Helper //
type Primitive = null | boolean | number | string | bigint;
type Frame<Value> = Record<Variable, Value>;
```

Listing 4.3: The complete standard join point model for AranLang (1/2)

Note that we decided against providing redundant join points. For example, Listing 4.5 advises a `ReadExpression` with both `read@before` and `read@after`. However, since scope access in AranLang is safe and atomic, the `read@before` join point does not provide any additional information to the analysis. A similar observation applies to literals and `WriteEffect`. The node types that require both `@before` and `@after` join points are those that introduce a hiatus in the control flow: `EvalExpression`, `AwaitExpression`, and `YieldExpression`.

[1] `https://github.com/lachrist/aran/tree/c9f6ca0/lib/weave/flexible`

```
type Advice<H, X> = {
  "block@setup":
    (state:X, kind:ControlKind, hash:H) => X;
  "program-block@before":
    (state:X, kind:ProgramKind, head:Header[], hash:H) => void;
  "closure-block@before":
    (state:X, kind:ClosureKind, hash:H) => void;
  "segment-block@before":
    (state:X, kind:SegmentKind, labels:Label[], hash:H) => void;
  "block@declaration":
    (state:X, kind:ControlKind, frame:Frame<OtherValue>, hash:H) => void;
  "block@declaration-overwrite":
    (state:X, kind:ControlKind, frame:Frame<OtherValue>, hash:H) => Frame<ScopeValue>;
  "generator-block@suspension":
    (state:X, kind:GeneratorKind, hash:H) => void;
  "generator-block@resumption":
    (state:X, kind:GeneratorKind, hash:H) => void;
  "program-block@after":
    (state:X, kind:ProgramKind, result:StackValue, hash:H) => OtherValue;
  "closure-block@after":
    (state:X, kind:ClosureKind, result:StackValue, hash:H) => OtherValue;
  "segment-block@after":
    (state:X, kind:SegmentKind, hash:H) => void;
  "block@throwing":
    (state:X, kind:ControlKind, error:OtherValue, hash:H) => OtherValue;
  "block@teardown":
    (state:X, kind:ControlKind, hash:H) => void;
  "break@before":
    (state:X, label:Label, hash:H) => void;
  "test@before":
    (state:X, kind:TestKind, decision:StackValue, hash:H) => boolean;
  "intrinsic@after":
    (state:X, name:Intrinsic, result:unknown, hash:H) => StackValue;
  "primitive@after":
    (state:X, result:Primitive, hash:H) => StackValue;
  "import@after":
    (state:X, src:Source, spe:(Specifier|null), res:OtherValue, hash:H) => StackValue;
  "closure@after":
    (state:X, kind:ClosureKind, result:Function, hash:H) => StackValue;
  "read@after":
    (state:X, variable:Variable, result:ScopeValue, hash:H) => StackValue;
  "eval@before":
    (state:X, situ:DeepLocalSitu, code:StackValue, hash:H) => OtherValue;
  "eval@after":
    (state:X, result:OtherValue, hash:H) => StackValue;
  "await@before":
    (state:X, promise:StackValue, hash:H) => OtherValue;
  "await@after":
    (state:X, result:OtherValue, hash:H) => StackValue;
  "yield@before":
    (state:X, delegate:boolean, item:StackValue, hash:H) => OtherValue;
  "yield@after":
    (state:X, delegate:boolean, result:OtherValue, hash:H) => StackValue;
  "drop@before":
    (state:X, discard:StackValue, hash:H) => unknown;
  "export@before":
    (state:X, left:Specifier, right:StackValue, hash:H) => OtherValue;
  "write@before":
    (state:X, left:Variable, right:StackValue, hash:H) => ScopeValue;
  "apply@around":
    (state:X, callee:StackValue, that:StackValue, input:StackValue[], hash:H) => StackValue;
  "construct@around":
    (state:X, callee:StackValue, input:StackValue[], hash:H) => StackValue;
};
```

Listing 4.4: The complete standard join point model for AranLang (2/2)

```
xyz;
```

```
%aran.getValueProperty%(__advice, "read@after")(
  __state,
  "xyz",
  (
    %aran.getValueProperty%(__advice, "read@before")(__state, "xyz", "hash"),
    xyz
  ),
  "hash",
);
```

Listing 4.5: Redundant weaving of a `ReadExpression`

Table 4.1 summarizes the variants of join points, detailing their triggers and effects on the value stack. This information is essential for analyses that mirror the value stack, as it clarifies the impact each variant should have on the shadow stack.

Variant	Trigger	Stack
Expression Context		
`@before`	Before consuming the result of an expression	Pop one
`@after`	After executing an expression	Push one
`@around`	Replaces a function call (application or construction)	Pop many Push one
Statement Context (only `BreakStatement`)		
`@before`	Before executing the statement	None
Block Context		
`@setup`	After entering the block; can extend the local state	None
`@before`	After `@setup`; receives program headers or control labels	None
`@declaration`	After `@before`; receives the current frame as a mapping of variables to their initial values	None
`@declaration-overwrite`	After `@declaration`; receives the current frame and can overwrite their initial values	None
`@suspension`	After the head of a generator and before the generator returns its iterator	None
`@resumption`	After the first call to the `next` method of the generator's iterator	None
`@throwing`	After an exception is thrown within the block; can overwrite the error	None
`@after`	After normal block completion; receives and can overwrite the result value (not `SegmentBlock`)	(Pop one)
`@teardown`	After leaving the block, regardless of whether it completed abruptly or normally	None

Table 4.1: Description of all join point variants

The most complex join points are related to blocks, which in AranLang can be classified as either `RoutineBlock`, `SegmentBlock`, or `GeneratorBlock`. To address the various control flow scenarios, the join points associated with blocks include multiple variants, as illustrated in Figure 4.1.

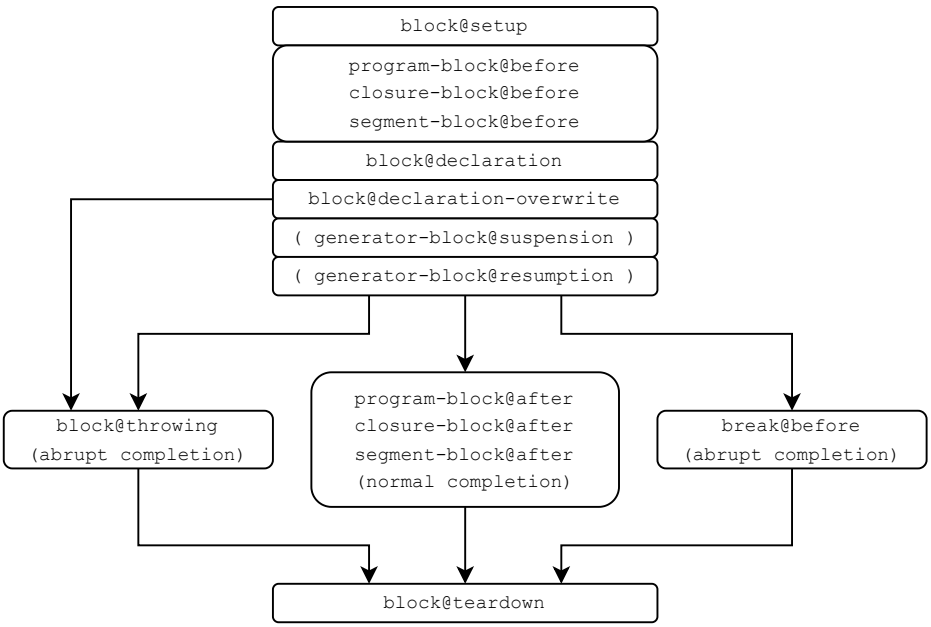


Figure 4.1: Triggering flow of block-related join points in the standard model

4.1.3 Shadow execution of AranLang via standard weaving

> *In this section, we demonstrate how the standard join point model provides a suitable API for conducting shadow execution of AranLang programs.*

Shadow execution [87, 72, 83, 17] is a common technique for maintaining analysis-related data during program execution without altering its behavior. It involves mirroring part of the state with a "shadow" to enrich it with information relevant to the analysis, such as sensitivity levels for taint analysis or relationships to program input for symbolic execution.

Shadow execution is one of the most comprehensive techniques available for conducting dynamic program analysis, requiring complete control over the execution of the target program to maintain synchronization between the shadow state and the actual state. Consequently, it serves as an excellent showcase for the capabilities of a join point model designed to support dynamic program analysis. The remainder of this section presents how shadow execution can be implemented within the standard join point model, serving as a demonstration of the expressiveness of our approach.

Listing 4.6 presents the types involved in our analysis as TypeScript type definitions:

- `Value`: An unknown type that is branded to aid the type system in uncovering mistakes.
- `Frame`: A mapping from strings representing variables to their corresponding values.
- `State`: A singly linked list where each element contains a scope frame and a value stack.

```
1 export type Value = {__brand: "Value"};
2 export type Frame = Record<string, Value>;
3 export type State = {parent: State | null, frame: Frame, stack: Value[]}
```

Listing 4.6: TypeScript type definitions for our shadow execution analysis

Listing 4.7 defines accessor procedures for managing the state:

- `pop`: Removes a value from the current value stack and verifies via `Object.is` (the strictest comparison method in JavaScript) that it matches the provided argument; the argument value is returned for convenience.

- `push`: Adds a value to the current value stack and returns the new value for convenience.
- `read`: Retrieves a variable from the current scope by traversing the state list structure.
- `write`: Updates a variable in the current scope by traversing the state list structure.

```
/** @typedef {import("./types").Value} Value */
/** @typedef {import("./types").State} State */

/** @type {(state: State, value: Value) => Value} */
export const pop = ({ stack }, item) => {
  if (stack.length === 0)
    throw new Error("Empty stack");
  if (!Object.is(stack.at(-1), item))
    throw new Error("Stack value mismatch");
  stack.length--;
  return item;
};

/** @type {(state: State, value: Value) => Value} */
export const push = ({ stack }, item) => {
  stack.push(item);
  return item;
};

/** @type {(state: State | null, variable: string) => Value} */
export const read = (state, variable) => {
  while (state) {
    if (variable in state.frame)
      return state.frame[variable];
    state = state.parent;
  }
  throw new Error("Missing variable");
}

/** @type {(state: State | null, variable: string, value: Value) => Value} */
export const write = (state, variable, value) => {
  while (state) {
    if (variable in state.frame)
      return state.frame[variable] = value;
    state = state.parent;
  }
  throw new Error("missing variable");
};
```

Listing 4.7: Accessor procedures for manipulating the current state

Finally, Listing 4.8 defines a function that compiles an advice capable of mirroring both the scope and the stack of AranLang programs. While the analysis has no observable effects, it verifies, whenever applicable, that the shadow scope and the shadow stack accurately depict their actual counterparts. The advice is separated into six logical parts:

- Block (Lines 18–28): The `block@setup` advice extends the current state with an empty frame and an empty stack. The initial state is expected to be `null` to indicate the end of the linked list. The next trigger is `block@declaration`, which populates the current frame. Finally, the `@after` advices consume the returned value from the current stack.
- Around (Lines 29–42): Both `@around` advices share a similar structure: they first consume the arguments in reverse order, then the `this` argument if applicable, and finally the callee. Afterward, they forward the call and produce the resulting value on top of the stack. Note that the calls are forwarded to members of `%Reflect%`, which does not account for cases where the target program executes in a different realm from the analysis.
- Scope (Lines 43–52): The scope, represented as a singly linked list, can be traversed using a standard `while` loop over frames. When a value is read from the scope, it is verified that the actual value matches the corresponding shadow value. When a value is written to the scope, the shadow scope is updated accordingly.
- Producer (Lines 53–57): Literal AranLang nodes only require pushing a value on top of the stack.
- Consumer (Lines 58–61): Simple value consumption occurs when the value is discarded, exported, or used to branch the control flow.

- Hiatus (Lines 62–68): There are three AranLang expressions that introduce a hiatus in the control flow: `EvalExpression`, `AwaitExpression`, and `YieldExpression`. A value should be consumed from the current stack before the hiatus and produced on top of the stack afterward. Support for direct evaluation calls requires the user to deploy Aran at run time and link the export `weaveStandard` to the `weaveEval` parameter.

We believe that Listing 4.8 demonstrates the expressiveness of our approach, as it implements shadow execution of the scope and value stack in a clear and concise manner. It is important to recognize that our standard advice can conduct shadow execution of arbitrary JavaScript programs and, through transpilation, can handle constructs such as classes that are absent from the standard join point model. As noted in Section 3.1.2, directly applying this logic to JavaScript would necessitate hundreds of join points.

```js
import { pop, push, read, write } from "./state.mjs";

/**
 * @type {(
 *   config: {
 *     weaveEval: (root: import("aran").Program) => import("aran").Program,
 *   },
 * ) => import("aran").StandardAdvice<{
 *   State: import("./types").State,
 *   StackValue: import("./types").Value,
 *   ScopeValue: import("./types").Value,
 *   OtherValue: import("./types").Value,
 * }>}
 */
export const compileShadowAdvice = ({ weaveEval }) => ({
  // Block //
  "block@setup": (state, _kind, _hash) => ({
    parent: state,
    frame: /** @type {any} */ ({ __proto__: null }),
    stack: [],
  }),
  "block@declaration": (state, _kind, frame, _hash) => {
    Object.assign(state.frame, frame);
  },
  "program-block@after": (state, _kind, result, _hash) => pop(state, result),
  "closure-block@after": (state, _kind, result, _hash) => pop(state, result),
  // Around //
  "apply@around": (state, callee, that, args, _hash) => {
    for (let index = args.length - 1; index >= 0; index -= 1)
      pop(state, args[index]);
    pop(state, that);
    pop(state, callee);
    return push(state, Reflect.apply(callee, that, args));
  },
  "construct@around": (state, callee, args, _hash) => {
    for (let index = args.length - 1; index >= 0; index -= 1)
      pop(state, args[index]);
    pop(state, callee);
    return push(state, Reflect.construct(callee, args));
  },
  // Scope //
  "read@after": (state, variable, result, _hash) => {
    if (!Object.is(result, read(state, variable)))
      throw new Error("Scope value mismatch");
    return result;
  },
  "write@before": (state, variable, right, _hash) => {
    write(state, variable, right);
    return pop(state, right);
  },
  // Producer //
  "primitive@after": (state, result, _hash) => push(state, result),
  "intrinsic@after": (state, _name, result, _hash) => push(state, result),
  "import@after": (state, _src, _specifier, res, _hash) => push(state, res),
  "closure@after": (state, _kind, result, _hash) => push(state, result),
  // Consumer //
  "test@before": (state, _kind, decision, _hash) => pop(state, decision),
  "export@before": (state, _specifier, right, _hash) => pop(state, right),
  "drop@before": (state, discard, _hash) => pop(state, discard),
  // Hiatus //
  "eval@before": (state, root, _hash) => weaveEval(pop(state, root)),
  "eval@after": (state, result, _hash) => push(state, result),
  "await@before": (state, promise, _hash) => pop(state, promise),
  "await@after": (state, result, _hash) => push(state, result),
  "yield@before": (state, _delegate, item, _hash) => pop(state, item),
  "yield@after": (state, _delegate, result, _hash) => push(state, result),
});
```

Listing 4.8: Standard advice for conducting shadow execution

4.2 Aran: implementation of our instrumentation approach

In this section, we present Aran, a tool that implements the following: the transpilation from JavaScript to AranLang, as outlined in Section 3.4; the transpilation of AranLang back to JavaScript, as described in Section 3.3; and the weaving of the standard join point model presented in Section 4.1. This section is organized as follows:

- *Section 4.2.1 describes the general usage of Aran.*
- *Section 4.2.2 explains how to deploy Aran both sequentially and concurrently with the program under analysis.*
- *Section 4.2.3 highlights important internal design decisions.*

4.2.1 General usage of Aran

In this section, we present and demonstrate Aran's interface.

JavaScript programs can be deployed in a variety of ways including complex build systems and CI/CD pipelines. Attempting to integrate deployment in the Aran infrastructure would necessitate making assumptions about the execution context, which would limit the applicability of Aran. Instead, Aran focuses on JavaScript instrumentation which shifts the responsibility of deployment to the user. This design decision requires additional work from the user, potentially diminishing reusability, but it guarantees that Aran remains maximally applicable. Nonetheless, we discuss the two main deployment architectures of Aran in Section 4.2.2.

Aran has been realized as a tool that is available on GitHub[2] and published as an NPM package[3]. Our tool is implemented in vanilla JavaScript but relies on TypeScript type annotations for static type checking. It is a substantial project, comprising over 90,000 lines of code.

Figure 4.2 illustrates the multi-stage instrumentation process conducted by Aran. The pipeline consists of five passes: parse, transpile, weave, retropile, and generate. Each pass is responsible for a specific transformation of the input code. Aran implements the transpile, weave, and retropile passes, while the parsing and generation passes rely on external tools. Below, we describe each pass:

- Parse: Convert JavaScript code into an ESTree Program node. This pass is intentionally omitted from Aran to avoid dependencies and enhance interoperability with other tools. It can be realized with any ESTree-compatible parser, such as Acorn[4] or Esprima[5].
- Transpile: Convert ESTree nodes into AranTree nodes. This pass, detailed in Section 3.4, is the most complex, comprising 52,000 lines of code.
- Weave: Insert the analysis logic into AranTree nodes. As a result, analysis implementers interact with AranTree nodes rather than ESTree nodes. This design is intentional, as AranLang was developed to shield analysis implementers from the complexities of JavaScript. This pass is simpler than the Retropile pass, comprising 11,000 lines of code and supporting two APIs: the Standard API and the Flexible API, both discussed in Section 4.1.1.
- Retropile: Convert AranTree nodes back into ESTree nodes. This pass, detailed in Section 3.3, is relatively straightforward, with 8,000 lines of code, which is expected since AranLang is intended to be a close subset of JavaScript.

[2] `https://github.com/lachrist/aran`
[3] `https://www.npmjs.com/package/aran`
[4] `https://github.com/acornjs/acorn`
[5] `https://esprima.org`

- Generate: Convert the ESTree Program back into JavaScript code. This pass is also intentionally omitted from Aran to avoid dependencies and improve interoperability. It can be realized with any ESTree-compatible code generator, such as Astring[6] or ESCodeGen[7].

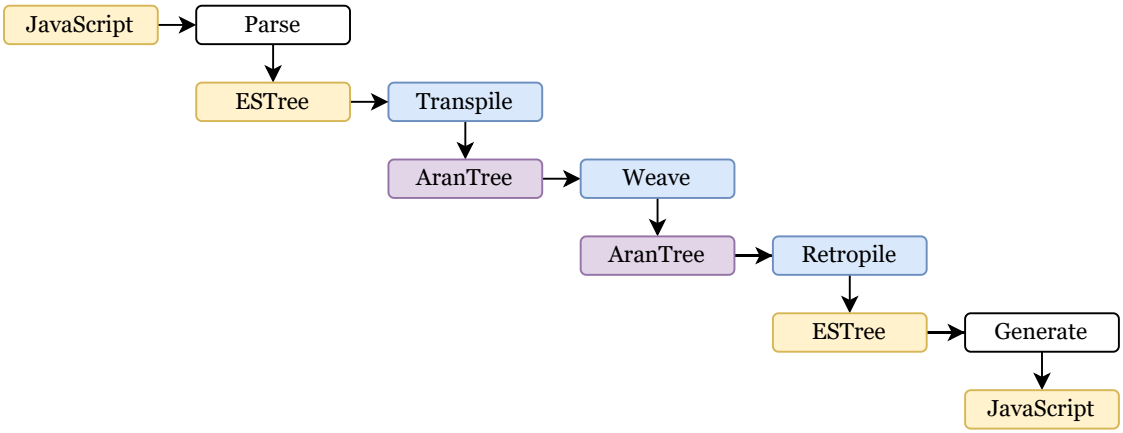


Figure 4.2: The overall pipeline of Aran instrumentation

In Listing 4.9, we demonstrate the usage of Aran by integrating it with Acorn and Astring to analyze a JavaScript program that computes the factorial of 6. The code reads as follows:

- Line 5 defines the initial state of the analysis, which is global and consists solely of the call stack depth.
- Line 6 declares the global variable that will hold the advice at run-time.
- Line 7 specifies the pointcut of the analysis, capturing every apply operation.
- Lines 8–20 outline the analysis logic as advice for apply operations. If the function being called is the factorial function, the call stack depth is incremented, the call is logged then forwarded, and the call stack depth is decremented.
- Line 22 assigns the advice to its global variable.
- Line 23 evaluates the setup code, which must always be executed before any instrumented code.
- Lines 34–38 instrument and evaluate the code. Information provided to Aran includes the kind of code being instrumented, its path, its root node, the advice variable, the pointcut, and the initial state.

[6] `https://github.com/davidbonnet/astring`
[7] `https://github.com/estools/escodegen`

```
import { instrument, compileIntrinsicRecord } from "aran";
import { parse } from "acorn";
import { generate } from "astring";

const conf = {
  advice_global_variable: "__ADVICE__",
  intrinsic_global_variable: "__INTRINSIC__",
  pointcut: ["apply@around"],
  initial_state: { depth: 0 },
}

global[conf.intrinsic_global_variable] = compileIntrinsicRecord(globalThis);
global[conf.advice_global_variable] = {
  "apply@around": (level, callee, that, args, hash) => {
    if (callee.name !== "fac")
      return Reflect.apply(callee, that, args);
    level.depth += 1;
    const prefix = ".".repeat(level.depth);
    console.log(`${prefix} >> fac(${args[0]})`.padEnd(20) + `at ${hash}`);
    const result = Reflect.apply(callee, that, args);
    console.log(`${prefix} << ${result}`.padEnd(20) + `at ${hash}`);
    level.depth -= 1;
    return result;
  },
};

const code = `
  const fac = (n) => n === 0 ? 1 : n * fac(n - 1);
  fac(6);
`;
const root = parse(code, { sourceType: "script", ecmaVersion: 2024 });
const file = { kind: "eval", situ: { type: "global" }, path: "main", root };
global.eval(generate(instrument(file, conf)));
```

↓↓↓

```
. >> fac(6)          at main#$.body.1.expression
.. >> fac(5)         at main#$.body.0.declarations.0.init.body.alternate.right
... >> fac(4)        at main#$.body.0.declarations.0.init.body.alternate.right
.... >> fac(3)       at main#$.body.0.declarations.0.init.body.alternate.right
..... >> fac(2)      at main#$.body.0.declarations.0.init.body.alternate.right
...... >> fac(1)     at main#$.body.0.declarations.0.init.body.alternate.right
....... >> fac(0)    at main#$.body.0.declarations.0.init.body.alternate.right
....... << 1         at main#$.body.0.declarations.0.init.body.alternate.right
...... << 1          at main#$.body.0.declarations.0.init.body.alternate.right
..... << 2           at main#$.body.0.declarations.0.init.body.alternate.right
.... << 6            at main#$.body.0.declarations.0.init.body.alternate.right
... << 24            at main#$.body.0.declarations.0.init.body.alternate.right
.. << 120            at main#$.body.0.declarations.0.init.body.alternate.right
. << 720             at main#$.body.1.expression
```

Listing 4.9: A simple analysis for recording the call stack

4.2.2 Deployment of Aran

In this section, we outline the process of deploying Aran, either sequentially or concurrently, alongside the program under analysis.

Aran focuses on exporting functions for manipulating the Abstract Syntax Tree (AST), while deployment logic has been intentionally excluded from its implementation. We made this design choice to accommodate the diverse deployment workflows of modern JavaScript applications. Next, we outline three complementary approaches for deploying Aran, each with its own trade-offs.

Offline deployment

Figure 4.3 illustrates the *offline* deployment of Aran. This architecture performs instrumentation and analysis sequentially. In the first stage, the target program is instrumented according to the pointcut of the analysis, and the setup code is generated. These two components are then bundled with the analysis advice, creating a standalone JavaScript program. In the second stage, the analysis is conducted by executing the bundled program normally. The offline architecture is advantageous for reducing the memory footprint and performance overhead of the analysis, as instrumentation is performed in advance.

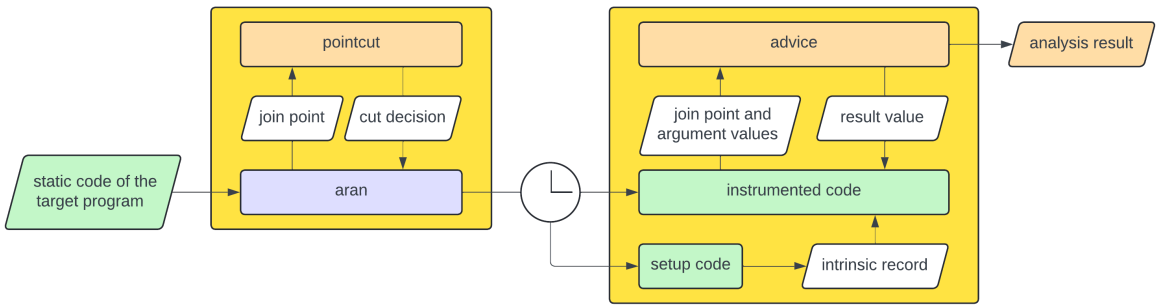


Figure 4.3: Deployment of Aran offline

Contemporary JavaScript applications are often built before deployment. This process typically involves bundling the source code and minifying it, producing smaller files that improve loading speed. Instrumentation can occur before, after, or even during this stage. When instrumentation takes place before the build stage, it must locate all sources, instrument them, and instruct the build stage to load them from their new locations. If instrumentation is applied after the build stage, it may hinder source-adjacency, since the analysis will no longer be conducted on the files as written by the developer. Ideally, the build stage should be configured to apply instrumentation as sources are loaded. Fortunately, popular JavaScript build systems such as Rollup, Gulp, and Grunt are highly configurable, enabling integration of Aran as a custom plugin.

Direct online deployment

Figure 4.4 illustrates the *online* deployment of Aran. In this architecture, instrumentation and analysis run concurrently within the JavaScript virtual machine, which may be less transparent than offline deployment. However, it enables the instrumentation of dynamically generated code. While global code does not strictly require instrumentation, code evaluated through a direct `eval` call must be imperatively instrumented. If not, the base layer of the target program could access the meta layer of the analysis. Consequently, direct `eval` calls are exclusively supported in this architecture.

When Aran is deployed online, the JavaScript virtual machine must be modified to intercept and instrument JavaScript code at load time. While patching the virtual machine is often feasible, doing so would tie the approach to a specific version of a particular runtime, undermining the primary motivation

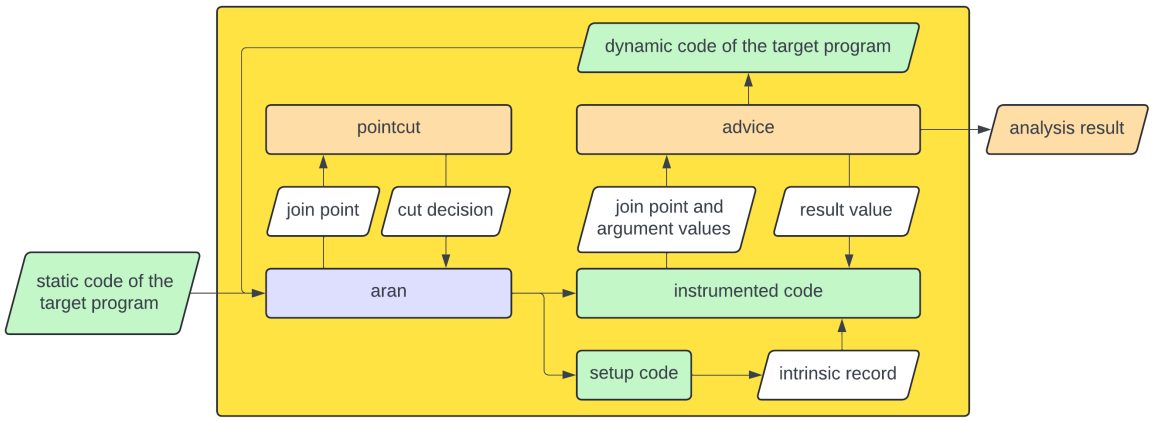


Figure 4.4: Deployment of Aran online

for conducting dynamic analysis through code instrumentation. Instead, whenever possible, the runtime should be configured to perform instrumentation at load time, which is much more portable.

Fortunately, load-time transpilation is common, making this feature essential for JavaScript virtual machines. For example, TypeScript offers two transpilation modes: ahead-of-time for production and at load time for development. Listing 4.10 illustrates the deployment of Aran in Node using its hook API[8]. In the diagram, files highlighted in red relate to the analysis, while the target program is represented in blue. Below, we describe each file in the listing:

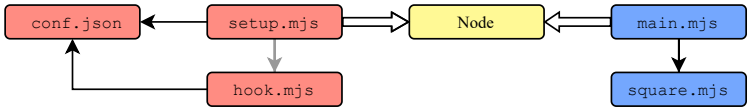
- `setup.mjs`: This file should be preloaded in Node using the `--import` flag. It registers `hook.mjs` as a global hook, assigns the advice to a hidden global variable, and evaluates the setup code.
- `hook.mjs`: Node executes this file in a separate thread and calls the `load` export when loading a source file. If the source is a module, it transforms it using `instrumentModule` from the analysis file.
- `analysis.mjs`: This file employs Aran, Acorn, and Astring to implement the instrumentation function and includes the advice `apply@around`; it is similar to Listing 4.9.
- `main.mjs` & `square.mjs`: These files represent the target program.

Indirect online deployment

If the JavaScript virtual machine lacks configuration options to support transpilation at load time, it may be possible to configure other parts of the runtime to conduct instrumentation at load time. When sources are fetched from the local file system, they can be transformed by a virtual file system such as FUSE. When sources are retrieved from the network, they can be transformed by a proxy server. This technique can operate in a single pass, where instrumentation occurs outside the virtual machine, either within the virtual file system or the proxy server. However, this requires setting up an additional communication channel to instrument dynamically evaluated code. This difficulty can be alleviated by implementing a two pass approach. Initially, Aran is injected into the JavaScript virtual machine, after which all code is stringified and passed as an argument to Aran's instrumentation functions.

This deployment technique becomes significantly more challenging when sources are fetched securely over SSL. A common solution is to conduct a man-in-the-middle attack, as illustrated in Figure 4.5, when the JavaScript virtual machine is embedded in a browser. Initially, the browser must be configured to redirect all requests to a forward proxy. Normally, in this setup, secure communication is achieved by establishing an HTTPS tunnel through the proxy server. However, in this case, the proxy impersonates the requested remote server using a self-signed root certificate that the browser trusts. We developed

[8] `https://nodejs.org/docs/latest-v22.x/api/module.html#customization-hooks`



conf.json

```
{ "advice_global_variable": "__ADVICE__",
  "intrinsic_global_variable": "__INTRINSIC__",
  "pointcut": ["apply@around"] }
```

setup.mjs

```
import { register } from "node:module";
import { compileIntrinsicRecord } from "aran/runtime";
import conf from "./conf.json" with { type: "json" };
register(new URL("hook.mjs", import.meta.url));
globalThis[conf.intrinsic_global_variable] = compileIntrinsicRecord(globalThis);
globalThis[conf.advice_global_variable] = {
  "apply@around": (_state, callee, that, args, hash) => {
    console.log(callee.name, hash);
    return Reflect.apply(callee, that, args);
  },
};
```

hook.mjs

```
import { parse } from "acorn";
import { generate } from "astring";
import { instrument } from "aran";
import conf from "./conf.json" with { type: "json" };
const instrumentModule = (url, content) => {
  const root = parse(content, { sourceType: "module", ecmaVersion: 2024 });
  const file = { kind: "module", path: url.split("/").pop(), root };
  return generate(instrument(file, conf));
};
export const load = async (url, context, loadNext) => {
  const result = await loadNext(url, context);
  if (result.format === "module")
    result.source = instrumentModule(url, result.source);
  return result;
};
```

square.mjs

```
export const square = (x) => x * x;
```

main.mjs

```
import { square } from './square.mjs';
console.log(square(2));
```

↓↓↓

```
> node --import ./setup.mjs main.mjs
square main.mjs#$.body.1.expression.arguments.0
aran.binary square.mjs#$.body.0.declaration.declarations.0.init.body
log main.mjs#$.body.1.expression
4
```

Listing 4.10: Online deployment of Aran via Node's hook API

our own tool, called Otiluke[9], to deploy Aran during a man-in-the-middle attack, but other tools such as mitmproxy[10] or Fiddler[11] can also be utilized.

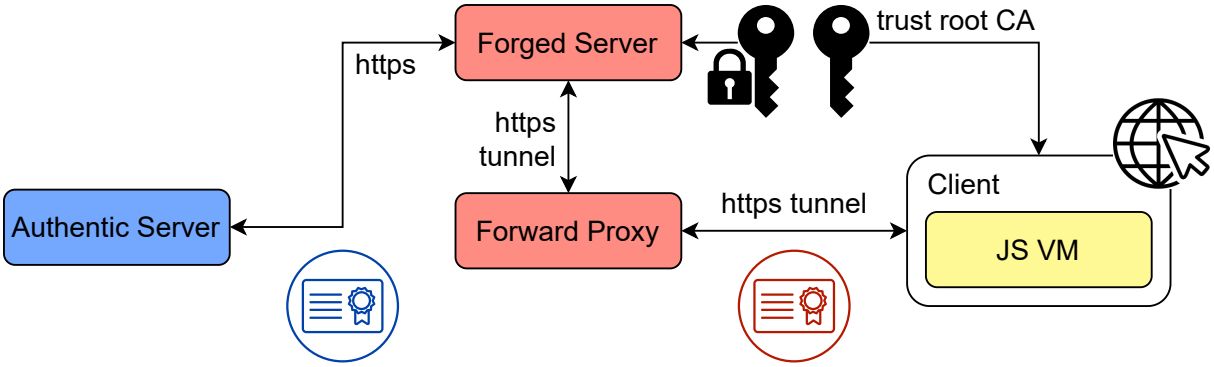


Figure 4.5: Man-in-the-middle attack for transforming JavaScript code in the browser

4.2.3 Internal design decisions

In this section, we list several implementation decisions regarding the inner workings of Aran.

The transpilation from JavaScript to AranLang described in Sections 3.4, 3.3, and 4.1 is quite complex, explaining why Aran is a substantial project with over 90,000 lines of code. This tool is the result of significant engineering efforts spanning 10 years, including periods of inactivity. In this section, we provide a historical perspective on several important internal design decisions related to the implementation of Aran that do not affect its external behavior.

Multi-stage architecture

The overall pipeline architecture depicted in Figure 4.2 was absent in the original version of Aran, which operated in a single stage. While this approach sufficed in the early days, it became unmanageable as we added support for more JavaScript features. By dividing the instrumentation into multiple stages, we separated concerns, making the complexity more manageable. In fact, it was the division into stages that drove the development of AranLang, rather than the other way around.

Validation and verification

In the past, Aran was implemented in pure JavaScript, and validation was ensured by achieving high coverage with unit tests. This approach led to significant friction in making the tool more compatible with the ECMAScript specification. Not only was it initially demanding (since writing unit tests is time-consuming) but it was also brittle and required ongoing maintenance. A typical test case involved comparing the actual transpilation output with the expected output; however, the number of correct transpilation variants is infinite. A trivial example of variation is changing the naming of transpilation variables, and every time we altered the naming convention, we had to adjust the unit tests. Other, more complex variations, such as changing the structure of the transpiled code, also caused issues.

Initially, we attempted to consolidate our unit tests with additional logic. However, the complexity of these tests became comparable to that of the implementation itself, which was unacceptable. Consequently, we decided to eliminate unit testing altogether and replace it with end-to-end testing by directly

[9] https://github.com/lachrist/otiluke
[10] https://mitmproxy.org/
[11] https://www.telerik.com/fiddler

evaluating the instrumented code. While they provide less diagnostic power than unit tests, these tests are resilient to variations in transpilation, as they focus on the external behaviors of transpiled programs. Instead of developing our own conformance test suite, we leveraged an existing suite named Test262, which is discussed in Section 4.3.1.

While using Test262 drastically reduced friction, it also increased the duration of development cycles and delayed feedback, as running Test262 takes several minutes and requires a fully built tool. To catch bugs earlier in the development process, we introduced TypeScript type annotations in comments to perform static type checks. We believe that without combining functional tests and TypeScript type annotations, we would not have succeeded in making Aran nearly fully compliant with the ECMAScript specification.

Functional programming style

Over the years, we migrated from an imperative style to a functional programming style. First, by removing side effects, the code became easier to reason about; for instance, the impact of changes was easier to evaluate, and fewer regressions were introduced. Second, type checking is more effective in a functional style, as it eliminates bugs related to the ordering of operations that cannot be caught by TypeScript.

To enforce a functional programming style, we utilize ESLint[12], a customizable static code analysis tool for JavaScript. We implemented several custom rules that, when applied together, make it difficult to write impure code. For instance, both assignment expressions and expression statements are prohibited—the former due to its explicit side effect, and the latter because it is meaningless in pure functional programming. Most of the logic required to implement Aran could be naturally expressed in a functional programming style without side effects, with two exceptions described below.

The first exception concerns the scope, which is mostly immutable, as the values of variables are not represented. When entering a block, the scope is extended with a new frame while the existing frames remain intact. To enhance performance and eliminate run-time checks, it is necessary to study the temporal dead zone of static variables, which requires updating their initialization status. Consequently, the scope must be updated whenever a `let` or `const` declaration is encountered. We implemented this update without side effects by ensuring that the relevant visitors return an updated version of the scope. Although we considered using the reader monad, we deemed it unnecessary for the few scope updates present in the codebase.

The second exception involves transpilation variables, which require state management to avoid clashes. In the code, the state is termed `meta` and can be manipulated using two functions: `nextMeta`, which returns the next state in the current namespace, and `forkMeta`, which returns a new namespace. Namespaces ensure that visitors can create an arbitrary number of transpilation variables without the risk of clashing with sibling nodes. Because this namespace manipulation was pervasive, we decided to make an exception in our ESLint rule to allow assignment to the `meta` variable.

4.3 Evaluating Aran's semantic transparency by applying it to Test262

Throughout Chapter 3, we discussed the transparency of Aran qualitatively. In this section, we address the transparency of Aran quantitatively by presenting an experiment that involves deploying basic analyses onto the official ECMAScript conformance test suite, Test262. This section is organized as follows:

- *Section 4.3.1 details the setup of the experiment.*
- *Section 4.3.2 discusses the semantic overhead observed during the experiment.*
- *Section 4.3.3 discusses the performance overhead observed during the experiment.*

[12] `https://eslint.org`

4.3.1 Setup of the Test262 experiment

In this section, we discuss the setup of the Test262 experiment. First, we introduce Test262, the codebase to be analyzed. Next, we outline the analyses to be applied to this codebase, some of which do not utilize Aran. Finally, we provide the hardware and software setup necessary for replicating the experiment.

Presentation of Test262

This first experiment tests our tool against the official ECMAScript conformance test suite, known as Test262. This suite includes tests that must pass for any compliant ECMAScript implementation and contains approximately 50,000 test cases that cover the entire ECMAScript specification, including upcoming proposals. Often, each test case must be executed twice (once in sloppy mode and once in strict mode) resulting in approximately 100,000 individual executions.

Listing 4.11[13] presents an example of a Test262 test case that evaluates the value returned by the `typeof` operator for boolean values. It performs two assertions using a non-standard `assert` function. This global function is defined in an auxiliary file, `harness/assert.js`, that must be executed prior to running any Test262 cases. The `harness` directory contains files that provide functionalities through the global scope to facilitate testing and reduce code duplication.

```
1 assert.sameValue(typeof true,  "boolean", 'typeof true  === "boolean"');
2 assert.sameValue(typeof false, "boolean", 'typeof false === "boolean"');
```

Listing 4.11: Example of Test262 test case

The transparency of our code transformation can be assessed by executing the instrumented version of Test262 cases and verifying that the outcomes are preserved. For example, running the instrumented version of Listing 4.11 should complete without throwing any exceptions. It is important to note that this method only evaluates the *external* transparency (as defined Section 2.2.1) of Aran's instrumentation, as it solely checks the outcomes of the instrumented test cases. However, the widespread use of assertions in Test262 ensures that this method serves as a reliable indicator of the *internal* transparency of the instrumentation.

To deploy our tool on Test262, we developed a custom test runner that adheres to the interpretation rules outlined in the repository[14]. Our test runner complies with most rules, except those concerning the implementation of *agents*, which serve as an abstraction for executing JavaScript concurrently. Agents complicate the test runner, and since inter-agent communication is managed through function calls, we consider them irrelevant for assessing the transparency of our code transformation. Consequently, we decided not to implement agents, which will result in the failure of agent-related test cases.

Our test runner is parameterized by an analysis that follows a custom format, which includes both a transformation function for instrumenting JavaScript code and a setup function for initializing advice if necessary. This analysis is deployed online, as described in Section 4.2.2, which is crucial for supporting the instrumentation of dynamically generated code. Our test runner performs the following steps, with steps 3 to 6 executed once or twice depending on whether it should run in both JavaScript modes for each Test262 case.

1. Prepare the main realm to conduct the analysis. This involves loading the test runner, Aran, and the analysis for the current pass.
2. Eagerly instrument the test harness files and cache the results. This improves performance, as many Test262 cases require executing some test harness prior to their main file.

[13] `https://github.com/tc39/test262/tree/18ebac81/test/language/expressions/typeof/boolean.js`
[14] `https://github.com/tc39/test262/blob/18ebac81/INTERPRETING.md`

3. Create a dedicated realm to execute the current Test262 case (in a specific mode). Realms provide a mechanism to execute JavaScript code in isolation, which is significantly more lightweight than restarting an entirely new virtual machine instance.
4. Execute the requested harness files from the instrumentation cache.
5. Load and instrument the main file of the current Test262 case; this step also involves prepending a `use-strict` directive if necessary.
6. Execute the instrumented main file of the current Test262 case; this may include loading and instrumenting dependency module files.

Participating analyses

The analyses used in the Test262 experiment are carefully structured to increase in complexity in a controlled manner. This incremental design helps isolate the root causes of semantic discrepancies encountered during development. It also supports the triage of remaining discrepancies across different components of the setup for blame assignment. Below, we briefly introduce the seven analyses involved in this experiment.

`none` [15] This no-op analysis leaves the test case intact and is designed to reveal the limitations of the underlying Node.js runtime or those of our custom test runner. Additionally, it serves as a reference point for calculating the slowdown factors introduced by subsequent analyses.

`parse` [16] This analysis parses and directly regenerates the files involved in the test case without manipulating their AST nodes. The goal of this analysis is to identify the limitations of the Acorn and Astring library.

`bare-main` [17] This analysis transpiles the main file of the test case into AranLang, transforms `EvalExpression` AST nodes to instrument dynamic local code, and finally retropiles the minimally woven AranLang code back into JavaScript. The goal of this analysis is to evaluate the transparency and performance impact of Aran's normalization alone.

`bare-comp` [18] This analysis also implements Aran normalization with minimal weaving; however, it extends the transformation to all source files, not just the main test file. This requires transforming not only `EvalExpression` AST nodes but also `ApplyExpression` and `ConstructExpression` AST nodes to intercept calls to `%$262.evalScript%`, `%eval%`, and `%Function%`, all of which evaluate code globally.

`stnd-void` [19] This analysis is similar to `bare-main`, but it utilizes standard Aran weaving to transform `EvalExpression` at the `eval@before` join point.

`stnd-full` [20] This analysis is similar to `stnd-void`, but it advises every standard join point using minimal forward logic. Like `bare-main` and `stnd-void`, only the main file and its locally evaluated code are instrumented. Listing 4.12 outlines the advice of this analysis for three representative join points. The advice for `eval@before` ensures support for direct calls to `%eval%` by normalizing the local code. The advice for `primitive@after` simply returns the value of the primitive. The advice for `apply@around` forwards the apply operation to `%Reflect.apply%`.

[15] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/identity.mjs`
[16] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/parsing-comp.mjs`
[17] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/bare-main.mjs`
[18] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/bare-comp.mjs`
[19] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/stnd-void.mjs`
[20] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/stnd-full.mjs`

```
advice["eval@before"] = (_state, code, situ, _hash) =>
  weaveLocalEvalCode(code, situ);
advice["primitive@after"] = (_state, primitive, _hash) => primitive;
advice["apply@around"] = (_state, callee, that, args, _hash) =>
  Reflect.apply(callee, that, args);
```

Listing 4.12: Parts of the advice of `stnd-full`

`track` [21] This analysis records the slice of the value flow graph related to each individual run-time value. The analysis employs shadow execution and is similar to the analysis described in Section 4.1.3, with two main differences. First, shadow values are value flow graphs whose root refers to the shadowed value. Second, an additional mechanism was implemented to shadow both the argument values and the result values during calls to instrumented functions. The analysis itself consists of approximately 300 lines of code and is applied only to the main file of each test case. Listing 4.13 presents the type declaration for the analysis alongside the value flow graph that it produces for simply doubling the value of `pi`. As demonstrated, even small computations generate large amounts of data; thus, this analysis is expected to have a significant impact on performance.

Replicability

For replicability, Table 4.2 summarizes our experimental software setup, all of which is fully accessible on GitHub. It includes the version, the name of the remote repository, the Git commit SHA, and the date. The experiment was conducted on an Apple Air M2 (2022) in 2025.

Name	Version	GitHub	SHA	Date
Node	`v22.13.0`	`nodejs/node`	`48726ac`	2025-01-07
Aran	`v5.2.1`	`lachrist/aran`	`161d57e`	2025-04-18
Acorn	`v8.14.0`	`acornjs/acorn`	`3c6a5a9`	2024-10-27
Astring	`v1.9.0`	`davidbonnet/astring`	`96dfb2b`	2024-08-25
Test262		`tc39/test262`	`18ebac81`	2024-07-24

Table 4.2: Overview of the software versions used in our experiment

[21] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/track-origin.mjs`

types.d.ts

```ts
type Variable = string;  type Parameter = string;
type Intrinsic = string; type Path = string;
type Primitive = number | string | boolean | null | { bigint: string };
type Leaf =
  | { type: "primitive"; primitive: Primitive; path: Path; }
  | { type: "closure", kind: string, path: Path }
  | { type: "intrinsic", name: Intrinsic, path: Path }
  | { type: "initial", variable: Variable | Parameter, path: Path }
  | { type: "import", source: string, specifier: string | null, path: Path }
  | { type: "resume", path: Path };
type Branch =
  | { type: "apply", function: Node, this: Node; arguments: Node[]; path: Path }
  | { type: "construct", function: Node, this: Node[], arguments: Path }
  | { type: "arguments", members: Node[], path: Path; }
type Node = Leaf | Branch;
type ShadowState = {
  parent: ShadowState | null;
  frame: { [key in Variable | Parameter]?: Node };
  stack: Node[];
};
```

target.mjs

```mjs
import { log } from "node:console";
const double = (x) => 2 * x;
const pi = 3.14;
log(double(pi));
```

↓↓↓

```json
{
  "type": "apply",
  "function": {
    "type": "intrinsic",
    "name": "aran.performBinaryOperation",
    "path": "$.body.1.declarations.0.init.body" },
  "this": {
    "type": "intrinsic",
    "name": "undefined",
    "path": "$.body.1.declarations.0.init.body" },
  "arguments": [
    {
      "type": "primitive",
      "value": "*",
      "path": "$.body.1.declarations.0.init.body" },
    {
      "type": "primitive",
      "value": 2,
      "path": "$.body.1.declarations.0.init.body.left" },
    {
      "type": "apply",
      "function": {
        "type": "intrinsic",
        "name": "aran.getPropertyValue",
        "path": "$.body.1.declarations.0.init" },
      "this": {
        "type": "intrinsic",
        "name": "undefined",
        "path": "$.body.1.declarations.0.init" },
      "arguments": [
        {
          "type": "arguments",
          "members": [
            {
              "type": "primitive",
              "value": 3.14,
              "path": "$.body.2.declarations.0.init" }],
          "path": "$.body.1.declarations.0.init" },
        {
          "type": "primitive",
          "value": 0,
          "path": "$.body.1.declarations.0.init" }]}],
  "path": "$.body.1.declarations.0.init.body" }
```

Listing 4.13: Value flow graph generated by the `track` analysis

4.3.2 Semantic overhead observed during the Test262 experiment

In this section, we quantitatively analyze Aran's transparency by examining the reasons for the failures of certain Test262 cases across the seven participating analyses.

Figure 4.6 provides an overview of the discrepancies observed during our seven analyses. The entire experiment was conducted as a pipeline, where test cases that failed during upstream analyses were removed from the corpus, as they would not succeed during more complex downstream analyses. Notably, analyses based on Aran demonstrate a very low failure rate, which is a significant achievement considering the complexity of the code transformations performed by Aran.

Next, we detail the observed semantic discrepancies for all seven analyses. Our discussion is based on a system for tagging Test262 cases available on GitHub and summarized in Table 4.3[22][23].

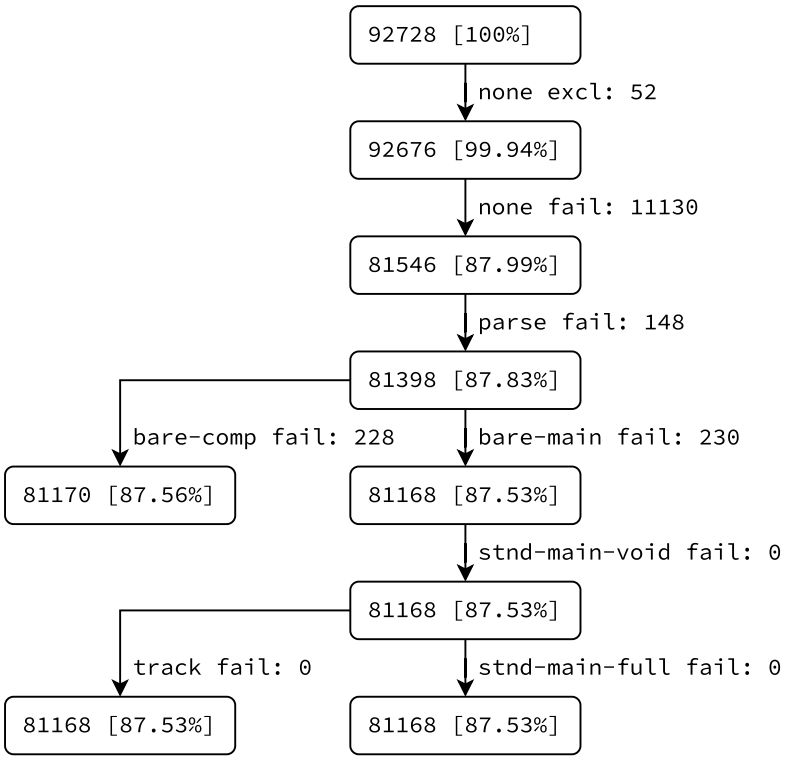


Figure 4.6: Evolution of the corpus of Test262 cases during the experiment

Semantic overhead observed during the `none` analysis

The `none` analysis provides a baseline execution of our Test262 experiment and identifies the limitations of our test runner and the underlying Node.js runtime. Right off the bat, we had to exclude 52 Test262 tests due to a Node-related issue that caused our test runner to hang on some asynchronous cases, leaving us with an initial corpus of 92,676 test cases. In total, we observed 11,130 failures, corresponding to a failure rate of 12%. Note that a single failure may have multiple reasons, even if it is rare. We categorize failures as follows:

- Feature 9,945 [89%]: These discrepancies arise from Node not yet implementing certain ECMAScript features. Some of these features may still be in the proposal stage and not yet promoted to the specification. The new `Temporal` API, proposed to replace the `Date` API, is responsible for most of the failures.

[22] `https://github.com/lachrist/aran/tree/161d57e/test/262/tagging/data`
[23] `https://github.com/lachrist/aran/tree/161d57e/doc/src/_issues`

Tag	Count	Cause
none		
Temporal	8430 [75.74%]	(Feature)
$262.detachArrayBuffer	582 [5.23%]	**Runner**
$262.agent	224 [2.01%]	**Runner**
regexp-unicode-property-escapes	220 [1.98%]	(Feature)
Intl.DurationFormat	194 [1.74%]	(Feature)
source-phase-imports	186 [1.67%]	(Feature)
$262.IsHTMLDDA	140 [1.26%]	**Runner**
ShadowRealm	124 [1.11%]	(Feature)
regexp-modifiers	122 [1.10%]	(Feature)
uint8array-base64	116 [1.04%]	(Feature)
eval-arguments-declaration	88 [0.79%]	(Runtime)
Float16Array	84 [0.75%]	(Feature)
Intl.Locale-info	84 [0.75%]	(Feature)
compound-assignment	77 [0.69%]	(Runtime)
legacy-regexp	50 [0.45%]	(Feature)
unknown	47 [0.42%]	(Runtime)
RegExp.escape	40 [0.36%]	(Feature)
regexp-duplicate-named-groups	38 [0.34%]	(Feature)
decorators	38 [0.34%]	(Feature)
resizable-arraybuffer	36 [0.32%]	(Feature)
FinalizationRegistry.prototype.cleanupSome	34 [0.31%]	(Feature)
tail-call-optimization	34 [0.31%]	(Feature)
import-assertions	33 [0.30%]	(Feature)
async-iterator-bypass-finally	28 [0.25%]	(Runtime)
update-inside-with	24 [0.22%]	(Runtime)
intl402	22 [0.20%]	(Runtime)
Math.sumPrecise	20 [0.18%]	(Feature)
promise-try	20 [0.18%]	(Feature)
date-coercion-order	16 [0.14%]	(Runtime)
prevent-extension-vm-context	16 [0.14%]	(Runtime)
explicit-resource-management	16 [0.14%]	(Feature)
Atomics.pause	12 [0.11%]	(Feature)
wrong-realm-for-dynamic-import	11 [0.10%]	(Runtime)
AbstractModuleSource	8 [0.07%]	(Feature)
flaky-symbol-replace	8 [0.07%]	(Runtime)
flaky-symbol-match	6 [0.05%]	(Runtime)
import-attributes	6 [0.05%]	(Feature)
atomic-wait-work	4 [0.04%]	(Runtime)
annex-b	2 [0.02%]	(Runtime)
Total	11130	
parse		
function-string-representation	100 [67.57%]	Acorn
import-assertion	32 [21.62%]	Acorn
specifier-literal	14 [9.46%]	Astring
cover-parenthesis	2 [1.35%]	Acorn
Total	148	
bare-main		
missing-iterable-return-in-pattern	84 [36.52%]	Section 3.4.8
function-string-representation	68 [29.57%]	Section 3.4.10
arguments-two-way-binding	32 [13.91%]	Section 3.4.9
early-declaration	20 [8.70%]	Section 3.4.5
async-iterator-async-value	14 [6.09%]	
wrong-realm-for-default-prototype	6 [2.61%]	
function-dynamic-property	2 [0.87%]	Section 3.4.10
duplicate-constant-global-function	2 [0.87%]	
duplicate-super-prototype-access	2 [0.87%]	
Total	230	

Table 4.3: Categorization of the failures observed during our Test262 experiment

- Runner 946 [8%]: These discrepancies are due to limitations in our custom test runner. For instance, accessing `$262.agent` throws an exception, causing the current test case to fail. While unfortunate, the test cases affected by these discrepancies primarily concern built-in functions, whereas the main goal of the Test262 experiment is to verify that Aran preserves the semantics of syntactic constructs.
- Runtime 349 [3%]: Node itself does not fully comply with the ECMAScript specification. Sometimes this is intentional, while at other times it indicates actual bugs in the Node.js runtime, for which we have submitted several reports[24][25][26].

Semantic overhead observed during the `parse` analysis

Like `none`, the `parse` analysis aims to identify failures that should not be attributed to Aran. The failures identified in this analysis should be attributed to either Acorn or Astring. The corpus of this analysis consists of 81,546 test cases, obtained by including only the test cases that passed the `none` analysis. We observed 148 failures which represent a failure rate of 0.18%. We briefly describe them below:

- `function-string-representation`: After parsing and regeneration, the source code of closures is no longer the same, which interferes with code reification provided via `%Function.prototype.toString%`.
- `import-assertion`: Import assertions are a recent addition to ECMAScript that Acorn does not yet fully support.
- `specifier-literal`: Module exports can be arbitrary string literals, which Astring does not yet support. We submitted a bug report for this[27].
- `cover-parenthesis`: When a closure is surrounded by parentheses, it should not be named according to the surrounding context, whether it is inside a property definition or a variable assignment. As the ESTree specification does not provide a node format for parentheses, this information is lost after parsing, resulting in a discrepancy.

Semantic overhead observed during the Aran-based analyses

We can now assess the transparency of Aran. The remaining corpus of Test262 cases that pass the `parse` analysis consists of 81,328 cases. We observed 230 failures, resulting in a failure rate of 0.28%. Table 4.3 categorizes these discrepancies and provides references in Chapter 3 where this issue was previously discussed, if applicable. Below, we describe the observed discrepancies.

- `missing-iterable-return-in-pattern`: Array destructuring assignments utilize the iterable protocol, which prescribes calling the `return` method of the iterator upon exiting the destructuring assignment, regardless of its outcome. However, after Aran instrumentation, this method is not invoked if an exception is thrown during iteration. Restoring this behavior is challenging because destructuring assignments occur in an expression context and cannot be directly enclosed in a `try` statement.
- `function-string-representation`: In addition to the `parse` analysis, Aran further alters the source code of closures, causing more calls to `%Function.prototype.toString%` to return a discrepant value. These discrepancies could be resolved by intercepting calls this built-in and the source code of the closure prior to instrumentation. However, we decided against implementing this solution because it would require significant engineering effort and is unlikely to be beneficial in real-world applications, as JavaScript code is often bundled and/or minified, which also does not preserve code reification. Note that the `%Error%` constructor offers another mechanism for code reification, but since the exact format of the stack trace is not part of the ECMAScript specification, it is not tested by Test262 and did not produce any additional discrepancies.

[24] `https://github.com/nodejs/node/issues/52720`
[25] `https://github.com/nodejs/node/issues/53575`
[26] `https://github.com/nodejs/node/issues/52737`
[27] `https://github.com/davidbonnet/astring/issues/713`

- `arguments-two-way-binding`: In sloppy mode, functions with a simple list of parameters have them two-way bound to the `arguments` object. This means that changes to the `arguments` object are reflected in the values of the parameters and vice versa. In a previous version of our tool, we preserved this behavior by leveraging the `Proxy` API [123]. However, we decided to remove this feature due to performance overhead and because dynamic argument binding is considered legacy.

- `early-declaration`: To enhance cohesion, Aran hoists export and variable declarations to the beginning of the source files. While the temporal dead zone is enforced within a source, other sources can bypass it. This issue arises for modules only in the case of circular dependencies. For scripts, it occurs when a script synchronously executes another script that uses its own declared global variables. Additionally, due to hoisting, Aran must convert globally declared `const` variables within scripts to `let` variables. While immutability is enforced in the current script, other programs can circumvent this mechanism and reassign the supposedly constant variable. We believe these discrepancies are the most likely to cause practical issues; however, addressing them would require a specific representation of the root scope of scripts in AranLang. In this alternative design, global variables declared in scripts are no longer hoisted and require dedicated join points. Additionally, transpilation variables are replaced by a registry to avoid polluting the global declarative frame.

- `async-iterator-async-value`: In an async `for...of` loop, if the `value` property of an iterator result is a promise-like object, it will be awaited before the iteration. Currently, Aran simply always wraps this result within an `await` expression. This transformation is mostly transparent but can be observed as it introduces an additional tick in the event loop. Fixing this issue would require querying the iterator result to evaluate whether it is a promise-like object, but as this operation can also be observed, it is unclear whether this will introduce more discrepancies than it resolves.

- `wrong-realm-for-default-prototype`: During object creation, the `new.target` argument can sometimes be queried to determine the origin realm of the default prototype. However, this is unlikely to pose a significant issue in practice, as it necessitates access to values from different realms. Such access cannot be achieved through standard ECMAScript; instead, it requires engine-specific mechanisms, such as the native `node:vm` modules in Node.

- `function-dynamic-property`: Functions created in sloppy mode contain two properties that change dynamically as the function is being called: `arguments` and `caller`. We decided not to preserve this feature because it is technically challenging and because these properties have been deprecated, although they are still part of the ECMAScript 2024 specification.

- `duplicate-constant-global-function`: Aran converts global declarations of functions into global variable declarations. This transformation is transparent most of the time but not if the function is already present as a constant property of the global object. For instance, `function NaN (){}` throws an early instance of `%SyntaxError%`, while `var NaN` executes without throwing. Similar to `early-declaration`, resolving these discrepancies would require rendering AranLang more complex which would compromise the expressiveness of Aran.

- `duplicate-super-prototype-access`: Upon creating a derived class, Aran accesses the `prototype` property of the parent class twice, which simplifies the transpilation process. As this operation can be observed with accessors or proxies, it can lead to discrepancies. However, this is unlikely to cause any issues in practice.

The failures of the `bare-comp` analysis are similar to those of the `bare-main` analysis, except that they do not introduce the two `duplicate-constant-global-function` failures. The remaining three Aran-based analyses did not introduce any additional failures.

The outcome of the Test262 experiment is extremely positive, demonstrating that Aran achieves a 99.8% success rate against a corpus of 81,546 Test262 cases. This indicates extensive compliance with the ECMAScript specification and highlights the practical applicability of Aran. While the discrepancies introduced by Aran may only appear in convoluted code, they should be taken seriously, as they can frustrate users who expect dynamic analysis to be highly accurate. To address this issue, we implemented a warning system that informs users when the instrumented program may contain discrepancies. Currently, this system is heuristic, and the absence of warnings does not guarantee transparency.

4.3.3 Performance overhead observed during the Test262 experiment

In this section, we address the overhead observed during our Test262 experiment.

In practice, most discrepancies are performance-related and can be attributed to the performance overhead introduced by the analysis. As the performance overhead of the analysis increases, the likelihood of impacting time-sensitive applications also rises. For instance, the analysis may observe event interleaving that would not occur otherwise. The reason performance overhead did not cause issues during our Test262 experiment is that this test suite is designed to be independent of performance to prevent flakiness.

However, neither Test262 nor our test run was designed to benchmark performance. Nonetheless, we recorded the time required for each analysis to execute each individual Test262 case separately within the corpus of 81,168 cases that passed all seven analyses. The recorded operations correspond to steps 3 to 6 of our test runner, as described in Section 4.3.1, which unfortunately combines both instrumentation and execution time. Next, in Section 4.4, we present an experiment specifically designed to benchmark performance, in which instrumentation has been excluded from the time recording. Table 4.4 provides a statistical summary of the observed slowdown factors, while Figure 4.7 visualizes the data as a boxplot.

Analysis	Mean	SD	Q1	Med.	Q3
`parse`	21	58	6	13	25
`bare-main`	26	65	7	17	32
`bare-comp`	45	89	11	28	57
`stnd-main`	25	59	7	17	31
`stnd-full`	36	75	9	24	46
`track`	37	69	10	25	46

Table 4.4: Statistical summary of the slowdown factor for each Test262 case across participating analyses (excluding the `none` analysis)

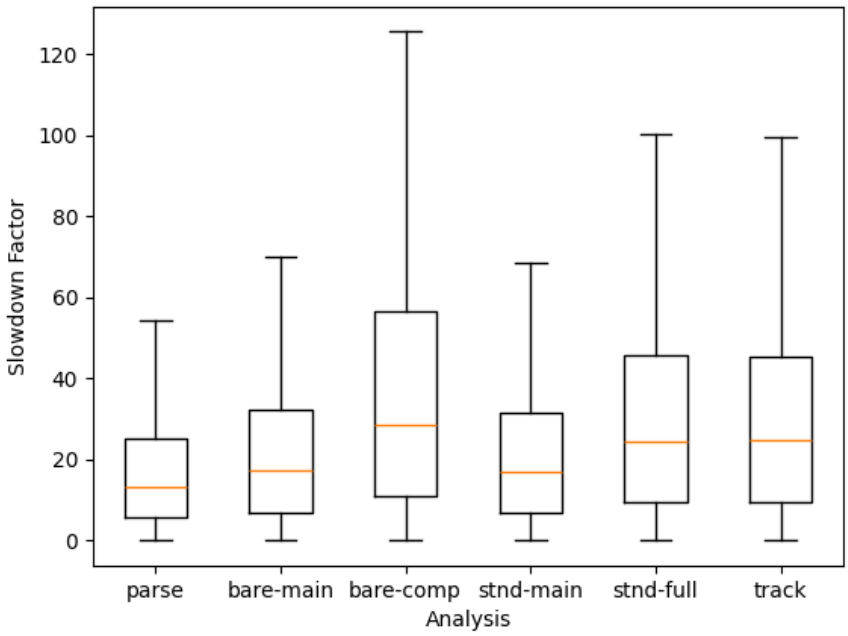


Figure 4.7: Boxplot visualization of the slowdown factors summarized in Table 4.4

It appears that the `parse` analysis already incurs a performance overhead of about 20×, indicating that instrumentation itself is an important contributing factor to the overall slowdown. Another finding is that

the analyses `bare-main` and `stnd-void`, which only perform minimal weaving, incur a marginally higher slowdown factor of about 25×, suggesting that Aran normalization is efficient compared to code parsing and generation. The analysis with the highest recorded slowdown was `bare-comp`, which comprehensively instrumented code from all sources with about a 45× slowdown. The performance impact of the advice of the remaining two analyses, `stnd-full` and `track`, was marginal, with a slowdown factor of about 35×.

These findings establish that, for fast-executing programs, the overhead introduced by instrumenting code significantly outweighs the overhead from executing the instrumented code. This experiment provides valuable insights into real-world scenarios, such as analyzing the execution of unit tests, which can potentially involve a large code base but run for a limited duration. Our findings suggest that to optimize the analysis of fast-executing programs, analysis builders should focus on improving the instrumentation rather than the execution; for instance, by caching instrumented code.

4.4 Evaluating Aran's performance overhead by applying it to Octane

Our Test262 experiment was designed to evaluate the semantic transparency of Aran. In this section, we present a second experiment on the Octane benchmark, specifically designed to quantify the performance overhead of Aran. This section is organized as follows:

- *Section 4.4.1 outlines the setup of the Octane experiment.*
- *Section 4.4.2 discusses the performance overhead observed during the experiment.*

4.4.1 Setup of the Octane experiment

In this section, we outline the setup of our Octane experiment, beginning with a description of the Octane benchmark, followed by an overview of the participating analyses.

The Octane (version 2)[28] suite contains 15 individual benchmarks that exercise various aspects of JavaScript; it incorporates tests from popular libraries and applications such as PDF.js, Mandreel, and Box2D. Each benchmark defines one or sometimes two entry functions that are executed multiple times by the runner specified in `base.js`. This repetition ensures consistent performance measurements by minimizing warm-up effects and reducing noise from transient run-time behaviors. Although Octane was retired in 2017[29], it remains widely used in the literature to assess the behavior of JavaScript engines under different research conditions—e.g., [108, 86, 96]. Table 4.5 outlines the 15 individual cases of Octane. We could not execute `zlib` (highlighted in red) due to missing global functions, leaving our Octane corpus with the 14 remaining cases.

Unlike the test runner for the Test262 experiment, the test runner for the Octane experiment instruments code offline. This architecture has the benefit of clearly separating the performance overhead of the instrumentation from the execution of the instrumented code. However, it does not support the instrumentation of dynamically evaluated code, which can lead to two types of semantic discrepancies:

- If the code is evaluated locally (i.e. within a direct call to `%eval%`), discrepancies may occur because the instrumentation interferes with the interface between the local code and its surrounding context. To identify this issue, we configured our Aran-based analyses to throw an exception when they encounter a direct `eval` call at run-time. However, while we observed that direct `eval` calls were statically present in four Octane cases (`earley-boyer`, `mandreel`, `pdfjs`, and `typescript`), they do not result in any local code evaluation at run-time.
- If the code is evaluated globally (e.g., within a call to `%Function%`) discrepancies may arise in analyses that require comprehensive instrumentation, such as those emulating the global declarative record.

[28] `https://github.com/chromium/octane`
[29] `https://v8.dev/blog/retiring-octane`

Name	Description	(KB)	(LoC)
box2d	Executes a 2D physics engine to test bitwise and mathematical operations.	223	566
code-load	Measures the time taken to load and parse JavaScript code.	113	1552
crypto	Performs encryption and decryption operations to evaluate bitwise and mathematical computations.	47	1698
deltablue	Implements a constraint-solving algorithm to test core language capabilities.	25	883
earley-boyer	Classic benchmark for memory allocation and garbage collection performance	191	4684
gbemu	Emulates a GameBoy system to test virtual machine capabilities.	504	11131
mandreel	Runs a JavaScript port of the Bullet physics engine to test virtual machine performance.	4883	277403
navier-stokes	Simulates fluid dynamics to test string and array processing.	13	415
pdfjs	Renders PDF files using Mozilla's PDF.js library to evaluate string and array handling.	1432	33053
raytrace	Executes 3D ray tracing algorithms to test core language performance.	27	904
regexp	Tests the efficiency of regular expression processing.	124	1805
richards	Simulates operating system kernel tasks to assess core language features.	15	540
splay	Manipulates splay trees to assess memory management and garbage collection.	11	423
typescript	Compilation of TypeScript code to JavaScript.	2449	25747
zlib	Performs data compression using asm.js to evaluate low-level JavaScript performance.	191	2426

Table 4.5: Overview of the Octane benchmarks

However, we have never observed such discrepancies. We attribute this to three factors: 1. Octane cases lack assertions, and discrepancies might have slipped unnoticed, 2. Octane was developed prior to ECMAScript 2015, which introduced block-scoped variables and the global declarative record, 3. Code evaluated globally by Octane executes mostly independently of static code.

To summarize the above, although our Octane runner does not support instrumentation of dynamically evaluated code, it has not resulted in any observable semantic discrepancies. On a related note, the `code-load` case contains minimal static logic; most of its logic is dynamically evaluated, which is precisely the focus of the case. Since this code does not undergo any transformation, the performance overhead for this particular case should be considered with caution. Other occurrences of dynamically evaluated code exist, but they contain minimal logic, and their analysis is unlikely to significantly impact the recorded timings.

The analyses participating in the Octane experiment are similar to those from the Test262 experiments:

- `none`: Like the `none` analysis from the Test262 experiment, this analysis does not modify the target program and provides a baseline execution.
- `bare`: Like the `bare-main` analysis from the Test262 experiment, this analysis only applies Aran normalization. The difference is that it does not perform any weaving at all; as a result, direct calls to `%eval%` will cause an exception to be thrown at run-time.
- `full`: Like the `stnd-full` analysis from the Test262 experiment, this analysis comprehensively weaves the target program with a simple forward advice.
- `track`: This analysis is the same as the `track` analysis from the Test262 experiment and offers a representative example of a moderately complex analysis based on shadow execution.

4.4.2 Performance overhead observed during the Octane experiment

In this section, we discuss the performance overhead observed during our Octane experiment, which separates instrumentation time from analysis execution time, unlike our Test262 experiment.

Table 4.6 and Table 4.7 summarize the results of our Octane experiment, including code size, average execution time, and average score. Octane scoring is directly correlated with execution time but provides a more user-friendly way to compare performance. In what follows, we focus on execution time, as it is more directly interpretable.

Table 4.8 presents a statistical summary of the code bloat factor, slowdown factor, and score decrease factor, all relative to the baseline execution obtained from the `none` analysis. The summary includes the first quartile (Q1), median, third quartile (Q3), mean, standard deviation (SD), and coefficient of variation (CV). Figures 4.8 and 4.9 provide visualizations in the form of boxplots for the code bloat factor and the slowdown factor, respectively.

First, we examine the increase in code size, which may have technical implications for large monolithic programs, such as bundles. The `none` analysis serves as an appropriate baseline for this measure, as it formats the code using the Astring generator, which is also employed in the other analyses. It appears that the Octane benchmark, after formatting, consists of medium-sized programs, with sizes ranging from 13 KB for `splay` to 4883 KB for `mandreel`. Our results indicate that Aran normalization alone is expected to increase code size by a factor of 10×, while a comprehensive weaving of all the standard join points further multiplies the code size by an additional factor of 10×.

Such code bloat resulted in a `RangeError` exception during the instrumentation of `mandreel` for the `full` analysis, as the string representation of the code contained too many characters. This technical issue can likely be resolved by modifying the code generation process and adopting a stream-like architecture. However, a more sustainable approach would be to avoid instrumenting large bundles all at once; instead, we recommend instrumenting modularized source files.

Analysis	Size[KB]	Time[ms]	Score
box2d			
none	304	3.58	151423
bare	5045	480	1131
full	82105	3637	149
track	74661	9464	57.4
code-load			
none	116	0.148 2.43	74926
bare	299	0.16 3.48	60229
full	2674	0.191 5.42	44126
track	2297	0.25 7.04	33919
crypto			
none	47	0.0933 1.61	69321
bare	823	7.04 156	803
full	10555	68.5 1276	90
track	9090	490 9038	13
deltablue			
none	22	0.0502	131562
bare	418	94.4	70
full	5506	284	23
track	4727	794	8
earley-boyer			
none	149	0.136 2.846	106888
bare	1872	55.5 849	307
full	25715	154 2548	106
track	21517	472 8686	33
gbemu			
none	450	18.3	143297
bare	5401	1997	1316
full	78617	13068	201
track	70979	43060	61

Table 4.6: Results of the Octane experiment (1/2)

Analysis	Size[KB]	Time[ms]	Score
mandreel			
none	5974	20.6	81498
bare	32062	8312	203
full	Code Bloat		
track	483045	225469	7
navier-stokes			
none	17	3.42	43290
bare	310	38.7	3828
full	4397	1499	99
track	3651	8799	17
pdfjs			
none	1445	10.7	94701
bare	6906	529	1912
full	134392	3326	304
track	118012	13940	73
raytrace			
none	28	0.715	103524
bare	409	265	279
full	6082	741	100
track	5337	1800	41
regexp			
none	130	7.14	12754
bare	3727	105	868
full	7139	512	178
track	6407	1908	47
richards			
none	16	0.0724	48720
bare	290	35.5	99
full	3758	113	31
track	3242	306	11
splay			
none	13	0.352	23096
bare	215	8.33	978
full	2841	27.4	298
track	2440	69.4	117
typescript			
none	2266	155	164100
bare	12309	11839	2154
full	194363	53363	478
track	165937	117478	217

Table 4.7: Results of the Octane experiment (2/2)

Analysis	Q1	Median	Q3	Mean	SD	CV
Size						
bare	5	17	18	14	7	0.5
full	93	217	235	175	80	0.46
track	81	188	203	147	70	0.47
Time						
bare	24	97	371	261	436	1.67
full	311	734	1036	924	1301	1.41
track	758	2573	4227	3588	4039	1.13
Score						
bare	24	109	371	286	471	1.65
full	312	713	1016	1006	1432	1.42
track	756	2546	4429	3834	4524	1.18

Table 4.8: Statistical summary of the results from the Octane experiment

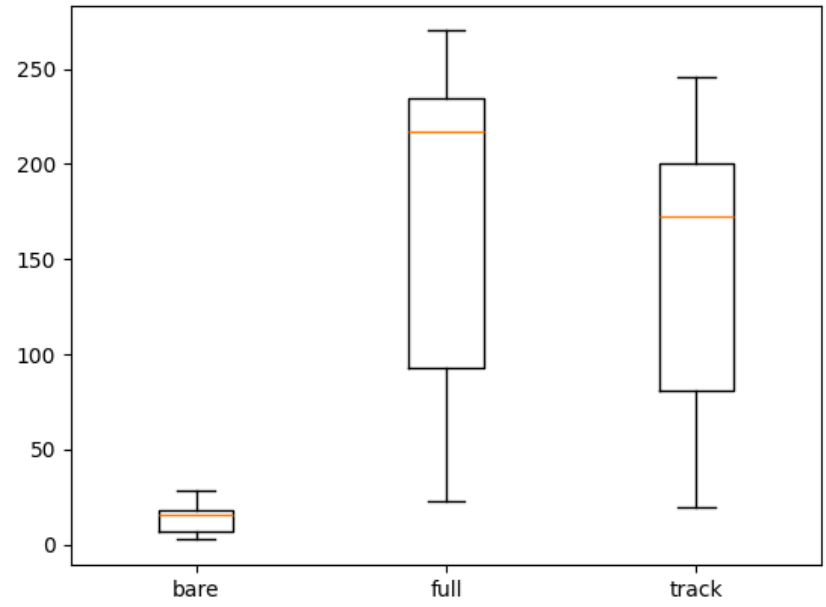


Figure 4.8: Boxplot of the code size increase factor observed in our Octane experiment

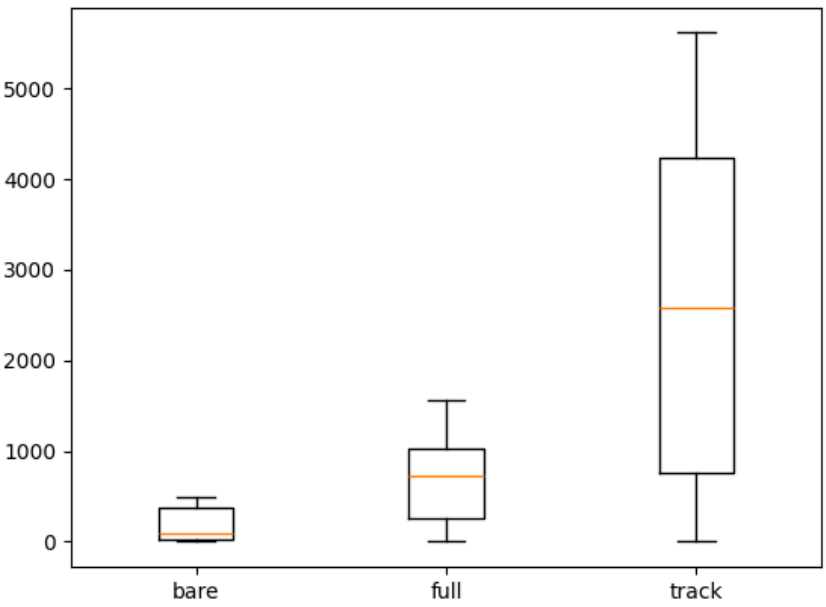


Figure 4.9: Boxplot of the slowdown factor observed in our Octane experiment

We now analyze the performance overhead introduced by Aran, measured by both time and score metrics. Since these two values are highly correlated, we focus on the time slowdown factor, which provides a more straightforward interpretation than the score decrease factor. Normalization alone introduces a performance overhead of approximately 100×, while comprehensive weaving with simple forward logic further increases the overhead by a multiplicative factor of 5×. The logic of our `track` analysis adds yet another multiplicative factor of 5× to the performance overhead. The only exception to this trend is `code-load`, which exhibits nearly constant execution time across analyses. This behavior is expected, as the dynamic evaluation of global code remains invisible to our analyses.

To summarize, we observed a slowdown of 100× for normalization, while additional logic adds another 25× slowdown factor, resulting in a total slowdown of 2500×. Although the 25× analysis slowdown seems justified, the initial 100× slowdown factor from normalization alone is surprising. We believe this negative result can be attributed to the following two contributing factors:

- Octane was retired in part because JavaScript engines were optimized to achieve high scores. Consequently, variations of the benchmark program without semantic changes may worsen the outcome due to falling out of the finely tuned optimization conditions.
- Octane uses many global variables, whose access is highly regulated after Aran normalization, incurring performance overhead. This contributing factor may not impact modern codebases as significantly, as the use of global variables is discouraged in contemporary programming practices.

It may be tempting to attribute the observed 100× slowdown of the `bare` analysis to the numerous unary and binary operations present in the Octane benchmark. However, as explained in Section 3.3.7, although these operations are initially transpiled into calls to reflective functions (which is quite slow) they are subsequently retropiled into standard operations when they fall outside the pointcut specification of the analysis. Since the pointcut of `bare` is empty, retropilation to unary and binary operations is assured.

4.5 Conclusion

In this chapter, we demonstrated how AranLang can be utilized to provide an aspect-oriented API for the instrumentation of JavaScript programs which is expressive enough to conduct shadow execution. To conclude, we summarize results and limitations, outline the main contributions, review related work, and highlight future research directions.

Assessment and opportunities for improvement

In Section 4.1, we presented a join point model for AranLang. Our experience indicates that while developing analyses within this model is feasible, it necessitates careful reasoning. In particular, despite providing local states to the advice, managing the state of the analysis remains challenging and prone to errors. In the future, it may be beneficial to explore ways to simplify the implementation process for new analyses. For example, employing a domain-specific language could render the specification of analyses more declarative.

In Section 4.3.2, we evaluated the transparency of the instrumentation performed by AranLang. We encountered few semantic discrepancies and achieved a high success rate of 99.8%. In the future, it will be advantageous to enhance our system for warning about potential discrepancies. For instance, we could experimentally verify that the absence of warning ensures the absence of semantic discrepancies.

In Section 4.4.2, we assessed the performance overhead introduced by Aran on the Octane benchmark. Normalization alone appears to introduce a slowdown of 100×, while representative analysis logic incurs an additional multiplicative factor of 25×, resulting in a total slowdown factor of 2500×. Further research is needed to address the following questions:

- What is the slowdown factor in real-world scenarios?

- How does it impact the transparency of the analysis for time-sensitive applications?
- Can Aran's normalization be fine-tuned to reduce its prohibitive 100× slowdown factor?

Main contributions

The primary contribution of this chapter is to validate the proposition from Chapter 3: that focusing on a core variant of ECMAScript 2025 is an effective approach to dynamic analysis of JavaScript which is simpler than addressing the full complexity of the target language directly. Our demonstration is threefold:

- Section 4.1.3 demonstrates that this approach allows for the creation of a join point model that maintains manageable complexity while being sufficiently expressive to shadow the value stack and the environment of JavaScript programs.
- Section 4.3.2 establishes that the semantics of the target language can largely be preserved using this approach.
- Section 4.4.2 demonstrates that this approach can handle computation-intensive programs, albeit at the cost of performance overhead.

Related work

Code instrumentation is a well-established technique for dynamic program analysis. Next, we discuss generic code instrumentation frameworks capable of expressing a wide range of dynamic program analyses.

Aspect-oriented programming Historically, aspect-oriented programming (AOP) was introduced as a means to separate the implementation of cross-cutting concerns and was intended as a technique for production use [60]. Dynamic analyses provide a clear example of a cross-cutting concern, and it is natural to express them as aspects. Whether or not they explicitly mention it, generic instrumentation frameworks can often be viewed as exposing an aspect-oriented API. Examples of instrumentation frameworks for dynamic analysis that explicitly describe themselves as aspect-oriented include Racer [10], which is based on AspectJ, and DiSL [75], which conducts binary code instrumentation.

Binary/Bytecode instrumentation frameworks There is a substantial body of literature on conducting dynamic program analysis through the instrumentation of binary code or bytecode. In this context, the most successful tools are dynamic binary translation frameworks such as Pin [72] and Valgrind [83], which instrument binary code at run-time. More closely related to our work is Wasabi [68], which targets WebAssembly. These generic frameworks provide a view of program execution that is no longer source-adjacent. This low-level perspective benefits analyses focused on profiling performance but is less suitable for analyses aimed at program comprehension.

Source instrumentation frameworks Source code instrumentation has been proposed as early as the 1970s, with FORDAP [64] for Fortran programs, PASDAP [110] for Pascal programs, and COBOLDAP for COBOL programs. Later, CIL [81] was proposed to instrument C programs; we discussed it in the previous chapter because it features transpilation to a core subset. As managed languages became more popular, source code instrumentation was naturally extended to them. In this context, one of the most closely related works is DynaPyt [32]: it targets Python, a language with an execution model similar to JavaScript and comprehensive enough to support taint analysis.

Jalangi The work most closely related to our approach is Jalangi [105], a pre-Harmony JavaScript instrumentation framework with support for shadow values. Initially, Jalangi provided record-and-replay capabilities for offline analysis, though this feature has been abandoned in more recent implementations.

Like our approach, Jalangi relies on the analysis implementer to advise run-time operations. The main differentiating factor between Jalangi and Aran is that Jalangi instruments the target language directly, whereas Aran instruments a core variant. This direct approach is reasonable for pre-Harmony JavaScript, but we contend that it would expose analysis implementers to excessive complexity when extended to ECMAScript 2025.

NodeProf Jalangi shares the same tradeoff as our approach: it is generic while requiring only a manageable amount of manual effort to build advanced dynamic program analyses. However, this comes at the cost of high performance overhead. This limitation motivated the development of NodeProf [117], which exposes an interface similar to Jalangi's but is implemented via runtime instrumentation (Graal.js in this case) rather than source code instrumentation. On the Octane benchmark, NodeProf was able to achieve a reduction in slowdown of one to two orders of magnitude, decreasing from the thousands to the tens.

Directions for future research

The main weakness of our approach is the manual effort required to leverage our framework to build user-facing tools. In particular, advice must often be written in an imperative style to manage state, which is error-prone. To alleviate this issue while remaining generic and applicable to a wide range of analyses, an interesting approach would be to develop a more declarative way to express analyses, for instance, by providing a domain-specific language. This could enhance cross-analysis reusability to the extent that analyses are developed on a per-application basis, transforming our approach into a ready-to-use infrastructure for dynamic program analysis.

Discussion: Applicability to other managed languages

In Section 3.5, we briefly speculate on how transpilation into a core variant would perform in other mainstream managed languages, with Python and Ruby emerging as the strongest candidates. Because these two languages also support closures, we believe it would be possible to propose an aspect-oriented interface for the core variant in these languages, similar to that of Aran. Regarding deployment, Python provides a mechanism to customize imports[30], which is superior to JavaScript, where import hooks lack standardization. And, true to its highly dynamic nature, Ruby allows monkey-patching of the `require` method from the `Kernel` module, which is responsible for loading modules.

[30] `https://peps.python.org/pep-0302/`

Chapter 5

Transparent value promotion for tracking provenance

In this chapter, we discuss how to track the provenance of run-time values in a sound manner while maintaining transparency. The goal is to establish an approach that is independent of consuming analyses, thereby achieving a separation of concerns between provenance tracking and analysis-specific logic. This chapter is organized as follows:

- *Section 5.1 introduces how promoting values enables sound provenance tracking.*
- *Section 5.2 details how to preserve transparency when allowing promoted values to reside in both the stack and the environment.*
- *Section 5.3 details how to preserve transparency when allowing promoted values to reside in the store.*
- *Section 5.4 presents applications of provenancial equality.*
- *Section 5.5 presents a metric to quantitatively evaluate the precision of provenance tracking.*
- *Section 5.6 concludes the chapter by summarizing contributions and discussing related work.*

5.1 Introduction to value promotion

In this section, we outline how promoting run-time values to references can serve as a basis for tagging them with metadata. Although conceptually simple, this approach has transparency implications. This section is organized as follows:

- *Section 5.1.1 describes how provenancial equality can be approximated via value promotion.*
- *Section 5.1.2 discusses the transparency implications of promoting values specifically in managed languages.*
- *Section 5.1.3 discusses the memory implications of promoting values and emphasizes the importance of focusing on managed languages.*
- *Section 5.1.4 explains how promoting values of unknown origin leads to imprecision.*

5.1.1 Approximating provenancial equality through value promotion

In this section, we outline how provenancial equality can be approximated by promoting run-time values to references.

The end goal of our approach is to facilitate provenance-aware analyses, such as taint analysis and symbolic execution, by providing a framework for attaching analysis-related data to run-time values—a process commonly referred to as *shadow values* [105, 20]. However, we deem this term to be a misnomer because the traditional technique for implementing shadow values is shadow execution [72, 83, 138, 14, 105] (discussed in Section 4.1.3) which is location-centric rather than value-centric. This technique involves maintaining a structure that contains metadata mirroring the run-time state of the program under analysis. Consequently, metadata is attached to locations in the run-time state, such as addresses in the store, variables in the scope, or indices in the value stack.

In this dissertation, we propose an alternative approach based on the realization (discussed in Section 2.3.4) that provenance can be effectively tracked in an allocation system where all newly created values are stored in fresh locations. To implement such an allocation system, two critical performance optimizations must be reverse-engineered: value inlining (cf. Section 2.3.5) and value interning (cf. Section 2.3.6). In mainstream managed languages such as Python and JavaScript, the value types targeted by these optimization are commonly referred to as primitive or atomic values. In contrast, value types that ensure fresh allocation are typically known as reference or object values. Hence, a straightforward approach to extend the naive allocation system to all value types involves *promoting* at least primitive value types to *handle* references.

Consider Listing 5.1, which illustrates how structural and provenancial equality should behave for numeric value types. As specified in Line 3, if two compared numeric values share the same content, they should be considered structurally equal. Note that in JavaScript, structural equality is made available for primitive value types via the `%Object.is%` intrinsic function. Additionally, as specified in Line 4, provenancial equality should be able to distinguish between two numeric values based on their provenance, even when they share the same content. This behavior aligns with our definition of provenancial equality from Section 2.3.7.

compare.mjs

```
1 import { provEq } from "./provenance.mjs";
2 const structEq = Object.is;
3 const num1 = 123, num2 = 123;
4 console.log(structEq(num1, num2)); // should be true  (same content)
5 console.log(provEq(num1 , num2));  // should be false (diff provenance)
```

Listing 5.1: Illustration of the desired behavior of structural and provenancial equality

Listing 5.2 defines a runtime for tracking the provenance of values. It consists of a *frontier module* that exports promotion and demotion operations, as well as a *provenance provider* that establishes provenancial equality by referentially comparing handles.

frontier.mjs

```
1 export const promote = (inner) => ({ inner });
2 export const demote = (handle) => handle.inner;
```

provenance.mjs

```
1 export const provEq = (handle1, handle2) => handle1 === handle2;
```

Listing 5.2: Simple runtime for tracking provenance

Leveraging the modules from Listing 5.2 to implement the desired behavior of Listing 5.1 requires three code transformations, as illustrated in Listing 5.3. Line 2 connects the provenancial equality comparison to an actual implementation. Line 4 inserts promotion operations around the two primitive literals. Line 5 preserves the behavior of structural equality by first demoting the operands.

To summarize, our approach to implementing provenancial equality involves promoting values, particularly primitive values, into handle references. Our approach prescribes two run-time components: a

compare.instr.mjs

```
import { promote, demote } from "./frontier.mjs";
import { provEq } from "./provenance.mjs";
const structEq = Object.is;
const num1 = promote(123), num2 = promote(123);
console.log(structEq(demote(num1), demote(num2))); // ✅ true  (same content)
console.log(provEq(num1, num2));                   // ✅ false (different provenance)
```

Listing 5.3: Instrumentation of Listing 5.1 for tracking provenance

frontier module that exports promotion and demotion operations, and a provenance provider that exports provenancial equality. Additionally, a source code transformer is required to insert promotion and demotion operations at the appropriate locations in the code.

5.1.2 Transparency implications of value promotion

> *In the previous section, we demonstrated that demoting the operand of structural equality was necessary to preserve its behavior. In this section, we discuss, more generally, the transparency implications of promoting values.*

Promoting primitive values to handles is a straightforward and effective solution for tracking provenance. However, it raises transparency concerns since references often do not behave like primitive values. In some cases, the built-in functions and operators of the language include mechanisms to convert unexpected references into primitive values. Unfortunately, these mechanisms do not suffice to ensure total transparency. We propose to illustrate the limitations of these mechanisms in Listing 5.4, which features basic manipulation and inspection of numeric values: addition at Line 2, type inspection at Line 3, and JSON serialization at Line 4.

```
1 const num = 123;
2 console.log(num + 1);             // should be 124
3 console.log(typeof num);          // should be "number"
4 console.log(JSON.stringify(num)); // should be "123"
```

Listing 5.4: Manipulation and inspection of numeric value in JavaScript

First, we discuss the plain wrapping strategy from Section 5.1.1. As shown in Listing 5.5, because the JavaScript runtime has no indication of how to convert plain wrappers into primitive values, all three manipulations require demotion to uphold transparency. At Line 3, the built-in addition serializes the handle into a string, resulting in string concatenation instead of arithmetic addition. At Line 4, the introspected string type of the handle is `"object"` instead of `"number"`. Finally, at Line 5, JSON serialization outputs the handle in object format rather than in number format.

```
1 const promotePlain = (inner) => ({ inner });
2 const num = promotePlain(123);
3 console.log(num + 1);             // ❌ "[object Object]1" (should be 124)
4 console.log(typeof num)           // ❌ "object"           (should be "number")
5 console.log(JSON.stringify(num)); // ❌ {"inner":123}      (should be 123)
```

Listing 5.5: Transparency implications of promoting values to plain objects

To indicate to the JavaScript runtime that a reference value is convertible to a primitive value, one can define a custom conversion method at the property key `%Symbol.toPrimitive%`. Listing 5.6 illustrates the transparency implications of promoting primitive values to convertible objects. At Line 4, the JavaScript runtime correctly demotes the handle back into its original primitive value within the built-in addition operator. However, the conversion is not applied at Lines 5 and 6 for type introspection and JSON serialization, respectively.

```
1 function returnInner () { return this.inner; }
2 const promoteConvertible = (inner) => ({ inner, [Symbol.toPrimitive]: returnInner });
3 const num = promoteConvertible(123);
4 console.log(num + 1);             // ✅ 124           (automatic convertion)
5 console.log(typeof num)           // ❌ "object"      (should be "number")
6 console.log(JSON.stringify(num)); // ❌ {"inner":123} (should be 123)
```

Listing 5.6: Transparency implications of promoting values into convertible objects

JavaScript runtimes also support an older and less flexible conversion mechanism commonly referred to as *primitive boxing*. This process involves constructing exotic objects that feature a hidden property holding the inner primitive values. Primitive boxing can be achieved via the `%Object%` intrinsic function, which returns a box object of the appropriate class. Listing 5.7 illustrates the transparency implications of promoting primitive values into box objects. At Lines 3 and 5, the JavaScript runtime correctly demotes the handle back into the original primitive value. However, at Line 4, the handle does not resist type introspection and still returns `"object"` instead of `"number"`.

While built-in conversion mechanisms can be beneficial, as the runtime handles some necessary demotion operations, this section illustrates that they do not provide a comprehensive solution. The intricate

```
const promoteBox = Object;
const num = promoteBox(123);
console.log(num + 1);             // ✅ 124      (automatic unboxing)
console.log(typeof num)           // ❌ "object" (should be "number")
console.log(JSON.stringify(num)); // ✅ 123      (automatic unboxing)
```

Listing 5.7: Transparency implications of promoting values into box objects

conversion rules make it unclear where demotion must be applied, potentially leading to bugs and semantic discrepancies. Therefore, we advocate for avoiding built-in conversion mechanisms altogether and instead recommend promoting values into plain objects.

5.1.3 Memory implications of value promotion

> *In this section, we examine how value promotion alters the memory layout of programs—an effect that is observable in languages featuring direct memory access.*

Compared to full-blown shadow execution, value promotion is more lightweight because it does not require maintaining a separate state. For instance, popping a value from the stack and passing it to a promotion-aware function (such as an instrumented closure) can be entirely delegated to the runtime, as the content of the value is never accessed. This raises the question of why value promotion has received less attention than shadow execution for implementing shadow values.

We believe the answer lies in the fact that shadow values have traditionally been implemented in unmanaged languages that do not strictly enforce value abstraction. For example, in C, the type checker offers a certain level of value abstraction, but this can be easily compromised by direct memory access. When value abstraction is violated, value-centric tagging becomes nonsensical; only location-centric tagging remains meaningful.

When facing direct memory management, achieving transparent value promotion becomes problematic. Consider Listing 5.8, which allocates a memory block intended to hold two `int32` numbers. Each cell in this block is `4` bytes in size, and the variable `size` stores this reflective information about the program's state. To promote the numbers in this memory block, we must increase the cell size to accommodate additional memory. For example, to tag these numbers with 8 bytes of metadata, we need to change the value of `size` from `4` to `12`. Achieving such code transformations transparently is challenging because reflective data may be logged or utilized by other parts of the program. As illustrated in Figure 5.1, shadow execution preserves memory layout and does not face the same difficulties.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main () {
  int size = 4;                 // ⚠️ Reflective information
  void *ptr0 = malloc(2 * size); // Arbitrary pointer arithmetic
  void *ptr1 = (ptr0 + size);    // Arbitrary pointer arithmetic
  *(int32_t*)ptr0 = 123;
  *(int32_t*)ptr1 = 456;
  printf("%d\n", *(int32_t*)ptr0); // 123
  printf("%d\n", *(int32_t*)ptr1); // 456
  printf("%d\n", size);            // 4
}
```

Listing 5.8: A C program that allocates a memory segment of 8 bytes for two integers

In this section, we have observed that direct memory access makes it challenging to promote primitive values transparently. By concentrating on managed languages in this dissertation, we avoid this issue entirely.

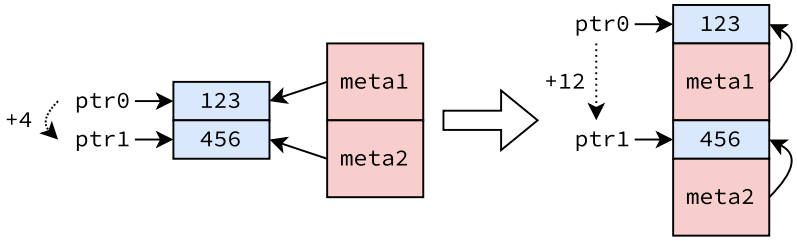


Figure 5.1: Memory layout when performing shadow execution (left) and when promoting values (right)

5.1.4 Promoting values of unknown origin

> *In this section, we explore two strategies for handling the promotion of values with an unknown origin.*

Thus far, we have only examined the promotion of values originating from primitive literals, where their origin is fully known at the time of promotion. However, it is crucial to address scenarios in which the origin of values is uncertain at the time of promotion. For instance, such adverse situations arise when a value is returned from an unknown arbitrary function. In these cases, value promotion can no longer ensure the accuracy of the corresponding provenancial equality, which becomes an approximation.

In the following discussion, we exemplify the behavior of two approximations of provenancial equality: one that is sound and one that is complete. As defined in Section 2.3.7, a sound approximation of provenancial equality returns `true` only if both operands share the same provenance. In contrast, a complete approximation of provenancial equality returns `false` only if both operands do not share the same provenance.

Listing 5.9 depicts a provenance provider that exports two such approximations of provenancial equality. It relies on a frontier module that extends the handle with an origin tag. The sound approximation of provenancial equality is implemented by referentially comparing both handle operands, while the complete approximation of provenancial equality defaults to structural equality when the origin of either operand is unknown.

frontier.mjs

```
export const promote = (inner, origin) => ({ inner, origin });
export const demote = (handle) => handle.inner;
export const hasUnknownOrigin = (handle) => handle.origin === "unknown";
```

provenance.mjs

```
import { hasUnknownOrigin, demote } from "./frontier.mjs";
export const provEqSound = (handle1, handle2) => handle1 === handle2;
export const provEqCompl = (handle1, handle2) =>
  hasUnknownOrigin(handle1) || hasUnknownOrigin(handle2)
    ? demote(handle1) === demote(handle2)
    : handle1 === handle2;
```

Listing 5.9: Two approximations of provenancial equality (one sound, the other complete)

Listing 5.10 illustrates the behavior of the provenance provider from Listing 5.9 when the origins of both operands are fully known. At Lines 5 and 6, since both operands share the same provenance, the sound approximation may return either `true` or `false`, while the complete approximation must return `true`. Conversely, at Lines 9 and 10, since the operands do not share the same provenance, the sound approximation must return `false`, whereas the complete approximation may return either booleans. Since the origin of both operands is known, both approximations are accurate.

Let's now consider Listing 5.11, which depicts a module that will not be instrumented. This may arise because the resource is unavailable at the time of instrumentation, or the analysis implementers may have chosen not to instrument certain files for performance reasons. The module exports two functions: the identity function and a function that always returns the number `123`, both of which log their arguments.

Listing 5.12 depicts the behavior of two approximations of provenancial equality when the origin of one operand is uncertain. Since the result of the identity function shares the same origin as its argument, the

```
1
2  import { provEqSound, provEqCompl } from "./provenance.mjs";
3  // Same Provenance //
4  const num1 = 123;
5  console.log(provEqSound(num1, num1)); // Can be either true or false
6  console.log(provEqCompl(num1, num1)); // Should be true
7  // Diff Provenance //
8  const num2 = 123;
9  console.log(provEqSound(num1, num2)); // Should be false
10 console.log(provEqCompl(num1, num2)); // Can be either true or false
```

⇓ ⇓ ⇓

```
1  import { promote } from "./frontier.mjs";
2  import { provEqSound, provEqCompl } from "./provenance.mjs";
3  // Same Provenance //
4  const num1 = promote(123, "literal");
5  console.log(provEqSound(num1, num1)); // ✅ true
6  console.log(provEqCompl(num1, num1)); // ✅ true
7  // Diff Provenance //
8  const num2 = promote(123, "literal");
9  console.log(provEqSound(num1, num2)); // ✅ false
10 console.log(provEqCompl(num1, num2)); // ✅ false
```

Listing 5.10: The behavior of both approximations of provenancial equality when the origins of both operands are fully known

```
1  export const identity = (x) => (console.log(x), x);
2  export const return123 = (x) => (console.log(x), 123);
```

Listing 5.11: Dependency module representing code that will not be instrumented

values of the variables `num1` and `num2` also share the same origin. However, this particular implementation of the identity function logs its argument, making it crucial to demote the argument at Line 5 to preserve transparency. Importantly, Line 5 also indicates to the frontier module that the origin of the result of the call is unknown. In this case, the complete guess, which relies on structurally comparing handles, is correct, whereas the sound safeguard, which relies on referentially comparing them, is incorrect.

```
1
2  import { identity } from "./external.mjs";
3  const num1 = 123;
4  const num2 = identity(num1);          // Should print 123
5  console.log(provEqSound(num1, num2)); // Can be either true or false
6  console.log(provEqCompl(num1, num2)); // Should be true
```

⇓ ⇓ ⇓

```
1  import { promote, demote } from "./frontier.mjs";
2  import { provEqSound, provEqCompl } from "./provenance.mjs";
3  import { identity } from "./external.mjs";
4  const num1 = promote(123, "literal");
5  const num2 = promote(identity(demote(num1)), "unknown"); // ✅ prints 123 (transparency)
6  console.log(provEqSound(num1, num2));                    // ❌ false (wrong sound safeguard)
7  console.log(provEqCompl(num1, num2));                    // ✅ true  (correct complete guess)
```

Listing 5.12: The behavior of two approximations of provenancial equality when facing a non-instrumented identity function

Similarly to Listing 5.12, Listing 5.13 depicts the behavior of both approximations of provenancial equality when the origin of one operand is uncertain. However, in this case, the two operands do not share the same provenance, despite having the same content. Again, to preserve transparency, the value passed to `return123` must be demoted, as it logs its arguments. In this instance, the complete guess is incorrect, while the sound safeguard is correct.

The remainder of this chapter is largely independent of the strategy used to approximate provenancial equality. Only our metric for measuring the precision of provenance tracking from Section 5.5 necessitates a sound approximation of provenancial equality. In our implementation, we focus on supporting only the sound approximation for two main reasons. First, it is clearly distinct from structural equality, making its advantages readily apparent. Second, it does not require additional metadata beyond a unique identity, which is automatically provided by the address of the handle in the store. In the future, it will be

```

import { return123 } from "./external.mjs";
const num1 = 123;
const num2 = return123(num1);         // Should print 123
console.log(provEqSound(num1, num2)); // Should be false
console.log(provEqCompl(num1, num2)); // Can be either true or false
```

⇓ ⇓ ⇓

```
import { promote, demote } from "./frontier.mjs";
import { provEqSound, provEqCompl } from "./provenance.mjs";
import { return123 } from "./external.mjs";
const num1 = promote(123, "literal");
const num2 = promote(return123(demote(num1)), "unknown"); // ✅ prints 123 (transparency)
console.log(provEqSound(num1, num2));                     // ✅ false (sound safeguard)
console.log(provEqCompl(num1, num2));                     // ❌ true  (incomplete guess)
```

Listing 5.13: The behavior of the two approximations of provenancial equality when facing a non-instrumented function that always returns the number `123`

worthwhile to investigate the impact of other approximation strategies on the precision of consuming analyses, such as taint analysis and concolic execution.

5.2 Syntax-centric frontier designs

The previous section demonstrated how promoting values can serve as a foundation for approximating provenancial equality. We now explore how to systematically enable the presence of promoted values within state locations that directly derive from syntactic constructs, such as in the continuation and environment in a CESK machine. This section is organized as follows:

- *Section 5.2.1 introduces the concept of a frontier by describing a simple analysis capable of tracking the provenance of values intra-procedurally.*
- *Section 5.2.2 formally defines the concept of a frontier.*
- *Section 5.2.3 examines frontier designs to track the provenance of values inter-procedurally.*

5.2.1 Intra-procedural frontier design

In this section, we introduce the concept of a frontier by presenting an analysis that transparently maintains the presence of handles in both the stack and the environment.

In the previous section, we demonstrated that promoting values can serve as a basis for approximating provenancial equality. This process involved inserting promotion operations to enhance precision and inserting demotion operations to maintain transparency. To systematize such code transformations, it is crucial to precisely define which value locations in the run-time state are susceptible to holding handles for enhanced precision, and which should contain only non-handle values to preserve transparency.

We refer to such separation of value locations within the run-time state as a *frontier*. The value locations that may hold handles are collectively referred to as the *internal region*, while those that should not contain handles are called the *external region*. Hence, handles are labeled as internal while other values are labeled as external. As shown in Figure 5.2, promotion and demotion operations serve as conversions between internal and external values. In this section, the internal region corresponds to state locations in the stack (continuation for a CESK machine) and in the environment that originate from the target program prior to instrumentation.

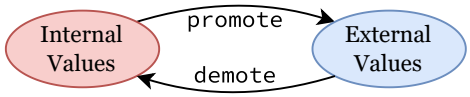


Figure 5.2: Basic frontier design

Figure 5.3 depicts the module dependency graph of the analysis for tracking the provenance of values intra-procedurally. The modules in gray are executed prior to the analysis to instrument the target program. The entry point of the analysis is `main.mjs`, which imports both the advice and the instrumented code. The analysis consists of an advice module, a frontier module, and a provenance provider. These files are in yellow to indicate that they generate state locations that will hold either internal or external values; we refer to these locations as the *bridge region*. Although the original version of the target program is not executed by the analysis, it is highlighted in red to indicate that it generates the state locations of the internal region. Finally, `node` refers to the Node.js runtime, specifically the module `node:console`, and is highlighted in blue to indicate that it generates state locations in the external region.

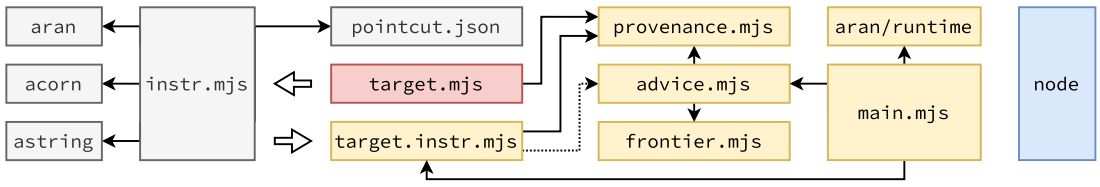


Figure 5.3: Module dependency graph for intra-procedural provenance tracking

For brevity, we omit `instr.mjs`, which contains 12 lines of boilerplate code, and `pointcut.json`, which merely enumerates the names of the advice functions implemented in `advice.mjs`. Listing 5.14 presents the main entry point of the analysis. Line 3 defines the intrinsic registry globally, Line 4 defines the advice globally, and Line 5 imports the instrumented version of the target program.

main.mjs

```
1 import { compileIntrinsicRecord } from "aran/runtime";
2 import { advice } from "./advice.mjs";
3 globalThis.__intrinsics__ = compileIntrinsicRecord(globalThis);
4 globalThis.__advice__ = advice;
5 await import("./target.instr.mjs");
```

Listing 5.14: Entry point of the analysis for intra-procedural provenance tracking

Listings 5.15 and 5.16 present the frontier module and the provenance provider of the analysis, respectively. The frontier module resembles the one from Section 5.1, except that it relies on a weak set to distinguish handles from structurally similar values (i.e., plain objects that also feature an `inner` property but have not been created by the `promote` function). This differentiation facilitates detecting violations of the frontier, such as when handles escape to the external region.

frontier.mjs

```
1  const handles = new WeakSet();
2  export const promote = (inner) => {
3    if (handles.has(inner))
4      throw new TypeError("Value has already been promoted");
5    const handle = { inner };
6    handles.add(handle);
7    return handle;
8  };
9  export const demote = (handle) => {
10   if (!handles.has(handle))
11     throw new TypeError("Value has not been promoted");
12   return handle.inner;
13 };
```

Listing 5.15: Frontier module of the analysis for intra-procedural provenance tracking

provenance.mjs

```
1 export const provEqSound = (handle1, handle2) => handle1 === handle2;
```

Listing 5.16: Provenance provider of the analysis for intra-procedural provenance tracking

Listing 5.17 presents the advice of the analysis which is responsible for enforcing the frontier between the internal and external regions. It demonstrates that the internal region encompasses the state locations in the stack and the environment that originates from the target program prior to instrumentation. The advice reads as follows:

- Lines 5–9 promote the initial values inside the environment frame of the target program. These can never be handles because variables are always initialized with either `%undefined%` or `%aran.deadzone_symbol%`, and the rest of the analysis ensures that implicit parameters are never initially bound to handles.
- Lines 11–16 promote values before they are pushed onto the value stack of the target program. Although `read@after` also pushes a value onto the stack, it is not defined, as values read from the environment should already have been promoted into handles.
- Lines 18–23 demote values that have just been removed from the stack before they escape to the external region, thereby preserving transparency. Although `write@before` also removes a value from the stack, it is not defined, as the environment should contain only handles.
- Lines 25–30 demote arguments before calling a function and promote the result. Crucially, it does not demote the arguments of `provEqSound`, as doing so would not only violate the frontier but also render the approximation of provenancial equality unsound.

Listing 5.18 presents the target program. When executing the instrumented code, the approximation of provenancial equality correctly returned `true` for the first occurrence originating from Line 5. This accurate result required tracking the provenance of values across both the stack and the environment. It also correctly returned `false` for the second occurrence originating from Line 6, which is expected for a

advice.mjs

```
import { promote, demote } from "./frontier.mjs";
import { provEqSound } from "./provenance.mjs";
export const advice = {
  // Initialize //
  "block@declaration-overwrite": (_state, _kind, frame, _loc) => {
    for (const variable in frame)
      frame[variable] = promote(frame[variable]);
    return frame;
  },
  // Produce //
  "primitive@after": (_state, primitive, _loc) => promote(primitive),
  "intrinsic@after": (_state, _name, intrinsic, _loc) => promote(intrinsic),
  "closure@after": (_state, _kind, closure, _loc) => promote(closure),
  "import@after": (_state, _source, _specifier, value, _loc) => promote(value),
  "yield@after": (_state, _delegate, result, _loc) => promote(result),
  "await@after": (_state, result, _loc) => promote(result),
  // Consume //
  "test@before": (_state, _kind, test, _loc) => demote(test),
  "export@before": (_state, _specifier, value, _loc) => demote(value),
  "eval@before": (_state, _code, _loc) => { throw new Error("TODO"); },
  "await@before": (_state, value, _loc) => demote(value),
  "yield@before": (_state, _delegate, value, _loc) => demote(value),
  "closure-block@after": (_state, _kind, result, _loc) => demote(result),
  // Combine //
  "apply@around": (_state, callee, that, input, _loc) =>
    demote(callee) === provEqSound
      ? promote(provEqSound(input[0], input[1]))
      : promote(Reflect.apply(demote(callee), demote(that), input.map(demote))),
  "construct@around": (_state, callee, input, _loc) =>
    promote(Reflect.construct(demote(callee), input.map(demote))),
};
```

Listing 5.17: Advice of the analysis for intra-procedural provenance tracking

sound approximation. However, it incorrectly returned `false` for the occurrence originating from Line 7. This inaccurate result, which renders our approximation of provenancial equality incomplete, is due to the analysis's inability to track the provenance of values across function calls.

target.mjs

```
import { log } from "node:console";
import { provEqSound } from "./provenance.mjs";
const identity = (x) => x;
const num = 123;
log(provEqSound(num, num));           // ✅ true  (same provenance)
log(provEqSound(num, 123));           // ✅ false (diff provenance)
log(provEqSound(num, identity(num))); // ❌ false (same provenance)
```

Listing 5.18: Target program for the analysis for intra-procedural provenance tracking

To summarize, we introduced the concept of a frontier by describing a simple Aran-based analysis capable of intra-procedurally tracking the provenance of values. To ensure transparency, the analysis was responsible for enforcing a frontier between an internal region encompassing state locations in the stack and environment, and an external region encompassing remaining state locations.

5.2.2 Defining frontiers and regions

In this section, we formally define what it means for an analysis to enforce a frontier.

In the previous section, we exemplified how an analysis may enforce a *frontier* between two regions. We now present a formal definition of this concept. The corresponding formalism, illustrated in Figure 5.4, relies on a pre-existing interpreter and a system for locating values within its state. The formalism reads as follows:

- The interpreter is characterized by the functions `load` and `step`. The `load` function establishes its initial state after loading a program, while the `step` function defines its labeled state transition system.

- The state location system is characterized by the `resolve` function, which resolves a (state) location by returning the value it holds within a state. To ensure totality, a placeholder value should be returned if the location is missing from the state.
- The frontier is characterized by the `regionalize` and `infer` functions, which respectively partition state locations across regions and allowlist values across regions. Both functions require a state for contextualization.
- The `reach` predicate inductively defines state reachability.
- The `satisfiesFrontier` predicate defines that a run-time state *satisfies* a frontier when each of its locations is compatible with the value it holds.
- The `respectsFrontier` predicate defines that a program *respects* a frontier when each of its reachable states satisfies the frontier.
- The `enforcesFrontier` predicate defines that an analysis, characterized as a program transformation function, *enforces* a frontier when it always generates programs that respect the frontier.

Sorts: `Program`, `State`, `Label`, `Value`, `Location`, `Region`

Signatures:

$\texttt{load} : \texttt{Program} \to \texttt{State}$

$\texttt{step} : \texttt{State} \times \texttt{Label} \to \texttt{State}$

$\texttt{resolve} : \texttt{State} \times \texttt{Location} \to \texttt{Value}$

$\texttt{regionalize} : \texttt{State} \times \texttt{Location} \to \texttt{Region}$

$\texttt{infer} : \texttt{State} \times \texttt{Value} \to \mathcal{P}(\texttt{Region})$

Definitions:

$\texttt{reach}(s_{from} : \texttt{State},\ s_{to} : \texttt{State}) :=$

$\quad s_{from} = s_{to} \ \lor\ \exists\ s : \texttt{State},\ \exists\ l : \texttt{Label},\ s\ =\ \texttt{step}(s_{from},\ l)\ \land\ \texttt{reach}(s,\ s_{to})$

$\texttt{satisfiesFrontier}(s : \texttt{State}) :=$

$\quad \forall\ l : \texttt{Location},\ \texttt{regionalize}(s,\ l)\ \in\ \texttt{infer}(s,\ \texttt{resolve}(s,\ l))$

$\texttt{respectsFrontier}(p : \texttt{Program}) :=$

$\quad \forall\ s : \texttt{State},\ \texttt{reach}(\texttt{load}(p),\ s)\ \implies\ \texttt{satisfiesFrontier}(s)$

$\texttt{enforcesFrontier}(\alpha : \texttt{Program} \to \texttt{Program}) :=$

$\quad \forall\ p : \texttt{Program},\ \texttt{respectsFrontier}(\alpha(p))$

Figure 5.4: Formal definition of a frontier

Figure 5.5 makes our formalism more concrete by adapting it to describe the frontier outlined in Section 5.2.1. The formalism reads as follows:

- The type `NodePath` identifies nodes in an abstract syntax tree and is formatted as a file path followed by a JSON path. For instance, `registry` identifies the construct call at Line 1 of Listing 5.16, which instantiates the handle registry. And, `promotion` identifies the object literal at Line 5 of Listing 5.16, which instantiates handles.
- The `Value` sort is divided into primitive and reference values, and it includes a special `missing` element to indicate when a location is invalid within a given state.
- The `Region` sort includes: the undefined region for handling invalid locations, the internal region for tracking provenance, the external region for ensuring transparency, and the bridge region for completeness.

- It is assumed that the interpreter is implemented as a CESK machine similar to the one described in Section 2.3.2. Thus, the `Location` sort can be divided into locations in the continuation, locations in environment frames (which should be indirected to support variable assignments), and locations in the data domain (such as the items of an array).
- The `traceLocation` (resp. `traceReference`) function returns the path of the AST node from which a given state location (resp. reference value) originated. These functions can be implemented as simple projections by incorporating the current AST node path along with the following state components created during transitions: new data items in the store, new frames in the environment, and new continuation structures.
- The `sourceMap` function establishes a mapping between node paths in the instrumented version of the target program and those in its original version.
- The `regionalize` function partitions state locations by returning: (a) the undefined region if the location is missing, (b) the internal region if the location is within the registry handle, (c) the external region if the location is elsewhere in the data domain, (d) the internal region if the location originates from the original version of the target program, (e) the bridge region if the location originates from the instrumented version of the target, (f) the bridge region if the location originates from analysis files, (g) the external region for all remaining locations.
- The `infer` function allowlists values by returning: (a) the undefined region if the value is missing, (b) both the bridge and the external regions if the value is a primitive, (c) both the bridge and the internal regions if the reference originates from the provenance provider, (d) only the bridge region if the reference originates from elsewhere in the analysis, (e) both the bridge and the external regions for all other references.

As evidenced by our formalism, assigning regions to a value requires the interpreter state to contextualize that value. For instance, defining `infer` in our formalism necessitated the interpreter state to determine the AST node from which the reference value originated. In what follows, we assume that this state context is implicitly provided. This assumption allows us to create value diagrams and discuss values belonging to a specific region without explicitly mentioning the state. Thus, when we say that a value is *internal* (resp. *external*), we mean that, with respect to an interpreter state, the value is allowed to appear within the `internal` (resp. `external`) region.

As a final note, a strong parallel can be drawn between our description of frontiers and type theory. By constraining the values that may appear at specific state locations, frontiers effectively serve as type constraints. The checks used to detect frontier violations in the frontier module of Listing 5.16 can be interpreted as dynamic type checks. These checks implement a form of nominal typing (as opposed to structural typing) since they rely on the identity of a value and its membership in a global registry rather than on its structure. Our formalism aims to define precisely what it means for a program to enforce a frontier, which we approached in a dynamic manner through a modified interpreter. In the future, it would be worthwhile to investigate how techniques from type theory could statically prove the absence of frontier violations in analyses specified by an aspect.

Sorts:

$\texttt{registry} := \texttt{"provenance.mjs\$.body[0].declarations[0].init"}$

$\texttt{promotion} := \texttt{"provenance.mjs\$.body[1].body.body[2].declarations[0].init"}$

$\texttt{NodePath} := \{\texttt{registry}, \texttt{promotion}, \ldots\}$

$\texttt{Value} := \{\texttt{missing}\} \cup \texttt{PrimitiveValue} \cup \texttt{ReferenceValue}$

$\texttt{Region} := \{\texttt{undefined}, \texttt{internal}, \texttt{external}, \texttt{bridge}\}$

$\texttt{Location} := \texttt{KontLocation} \cup \texttt{EnvLocation} \cup \texttt{DomLocation}$

Signatures:

$\texttt{traceLocation} : \texttt{State} \times \texttt{Location} \to \texttt{NodePath}$

$\texttt{traceReference} : \texttt{State} \times \texttt{ReferenceValue} \to \texttt{NodePath}$

$\texttt{mapSource} : \texttt{NodePath} \to \texttt{NodePath}$

Definitions:

$\texttt{regionalize}(s : \texttt{State}, l : \texttt{Location}) :=$

- $\texttt{undefined}\ \ if\ \ \texttt{resolve}(s, l) = \texttt{missing}$
- $\texttt{internal}\ \ if\ \ l \in \texttt{DomLocation} \wedge \texttt{traceLocation}(s, l) = \texttt{registry}$
- $\texttt{external}\ \ if\ \ l \in \texttt{DomLocation}$
- $\texttt{internal}\ \ if\ \ \texttt{mapSource}(\texttt{traceLocation}(s, l)) \in \texttt{"target.mjs"}\Sigma^*$
- $\texttt{bridge}\ \ if\ \ \texttt{traceLocation}(s, l) \in \texttt{"target.instr.mjs"}\Sigma^*$
- $\texttt{bridge}\ \ if\ \ \texttt{traceLocation}(s, l) \in (\texttt{"frontier.mjs"}\Sigma^* \cup \texttt{"advice.mjs"}\Sigma^*)$
- $\texttt{bridge}\ \ if\ \ \texttt{traceLocation}(s, l) \in \texttt{"provenance.mjs"}\Sigma^*$
- $\texttt{external}\ \ otherwise$

$\texttt{infer}(s : \texttt{State}, v : \texttt{Value}) :=$

- $\{\texttt{undefined}\}\ \ if\ \ v = \texttt{missing}$
- $\{\texttt{bridge}, \texttt{external}\}\ \ if\ \ v \in \texttt{PrimitiveValue}$
- $\{\texttt{bridge}, \texttt{internal}\}\ \ if\ \ \texttt{traceReference}(s, v) = \texttt{promotion}$
- $\{\texttt{bridge}\}\ \ if\ \ \texttt{traceReference}(s, v) \in (\texttt{"frontier.mjs"}\Sigma^* \cup \texttt{"advice.mjs"}\Sigma^*)$
- $\{\texttt{bridge}, \texttt{external}\}\ \ otherwise$

Figure 5.5: Formal definition of the frontier for intra-procedural provenance tracking

5.2.3 Inter-procedural frontier designs

> *In Section 5.2.1, we presented an analysis capable of tracking the provenance of values* intra-procedurally. *In this section, we explore frontier designs for tracking the provenance of values* inter-procedurally.

A primary approach to tracking the provenance of values inter-procedurally involves extending the internal region to include argument and result locations in the state. This method requires tagging functions with information that specifies the regions of their arguments and result. We refer to this function-centric approach as *adaptor*, as it leverages the adaptor pattern to maintain transparency. We also employed a call-centric approach that retains arguments and results in the external region but introduces a new global location in the internal region to temporarily hold handles before and after a call. We designate this approach as *transit*.

Both approaches can be made equivalent; however, they differ in implementation, which may be more or less convenient depending on the use case. For brevity, we omit the transit approach, although it is available online for further reference[1]. We chose to focus on the adaptor approach because it serves as a stepping stone toward our final frontier design described in Section 5.3.

The primary distinction between the adaptor frontier and the frontier discussed in Section 5.2.1 lies in the presence of additional references that may leak handles. Similar to the handle registry, these references, which we refer to as *dirty*, should reside exclusively in the bridge region because their presence in the external region could compromise the transparency of the analysis.

Figure 5.6 illustrates the application domain of our adaptor frontier, with arrows representing conversion functions. As noted in Section 5.2.2, values must always be contextualized by a state to assign them to a region. Consequently, the elements of the set diagram are members of `State` × `Value`, rather than members of `Value` alone. For instance, the `promote` conversion function requires a state in both its domain and codomain, as it inserts a new handle into the store. The three regions of the diagram are as follows:

- Dirty references can be either dirty arrays or dirty functions and may only exist within the bridge region (yellow). Dirty arrays contain handles and can be promoted to specific handles or cleaned up by demoting all their items. Dirty functions expect and return handles, and they can be promoted to specific handles or adapted into clean functions.
- Internal values are handles that may reside in either the internal region (red) or the bridge region. Handles are instantiated by promoting an external value, a dirty function, or a dirty array.
- External values are either primitive or clean references and may exist in either the external region (blue) or the bridge region. In contrast to dirty references, clean references cannot leak handles.

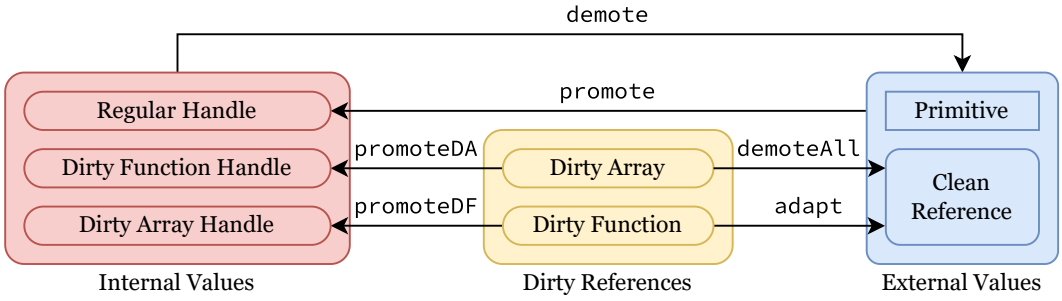


Figure 5.6: Application domain of the adaptor frontier

Listing 5.19 presents a frontier module for an analysis that enforces the adaptor frontier. Similar to the frontier module of the intra-procedural analysis from Section 5.2.1, it implements dynamic checks to detect frontier violations should they occur. The listing reads as follows:

[1] `https://github.com/lachrist/aran/blob/75baf524/test/262/staging/spec/provenancy/track-basic-aspect.mjs`

- `regions` (Line 1): To differentiate between value subsets of Figure 5.6, the handle registry has become a weak map that associates references with region tags.
- Analysis Registration (Lines 2–3): The registry itself is being registered as being part of the bridge region. The same should be done for the functions declared within this file.
- `infer` (Lines 4–7): Returns the region tags where a value is allowed to reside. If the value is a primitive or has not been registered, it may belong to both the bridge and the external region.
- `demote` (Lines 8–12): Converts a handle into an external value by simply unwrapping it.
- `promote` (Lines 13–19): Promote either a primitive value or clean reference into a regular handle. Note that clean references derived from dirty references will be promoted into a regular handle, resulting in precision loss. Preventing such loss involves registering when a dirty reference is converted into a clean reference; however, we did not include this mechanism here as it complicates the frontier.
- `demoteAll` (Line 20): Cleans up a dirty array by demoting all its items.
- `promoteDirtyArray` (Lines 21–26): Promotes a dirty array into a handle that can be used to recover the provenance of its original items.
- `adapt` (Lines 27–33): Cleans up a dirty function by adapting it into a clean function, which promotes its arguments and demotes its result. Note that the frontier does not support dirty coroutines, which require a separate case as they return clean references, whether they be promises or iterators.
- `promoteDirtyFunction` (Lines 34–39): Promotes a dirty function into a handle that can be used to recover it, bypassing the adaptor.
- `isDirtyIndexAccess` (Lines 40–44): Detects whether accessing a property of a handle can be made more precise by utilizing dirty arrays. Note that this detection is unsound, as the staleness check is not provenance-sensitive. Section 5.3.1 further elaborates on this point.
- `apply` (Lines 45–56): Calls a function which may involve accessing a dirty reference to enhance precision.
- `construct` (Lines 57–63): Calls a function as a constructor, which may involve calling a dirty function.

The remainder of the analysis involves leveraging the frontier module from Listing 5.19 to create an Aran standard advice. However, we omit this for brevity, as it closely resembles the advice in Listing 5.17. This results in an analysis capable of tracking the provenance of values inter-procedurally, enforcing a frontier whose internal region encompasses not only stack and environment locations but also argument and result locations.

To summarize, tracking the provenance of values through function calls requires not only introducing dirty functions but also dirty arrays, as arguments are collectively passed as arrays within Aran. However, our handling of dirty collections is not ideal, as the staleness check performed in `isDirtyIndexAccess` is insufficient to guarantee the soundness of the resulting provenancial equality. This indicates that while Aran is effective at tracking the provenance of values through syntactic constructs, it does not facilitate provenance tracking through the store, which will be the focus of the next section.

```
const regions = new WeakMap();
regions.set(regions, ["bridge"]);
// TODO: Register all following functions as being part of the bridge region.
const isPrimitive = (value) =>
  value === null || (typeof value !== "object" && typeof value !== "function");
const infer = (value) =>
  (isPrimitive(value) ? undefined : regions.get(value)) ?? ["external", "bridge"]
export const demote = (handle) => {
  if (!infer(handle).includes("internal"))
    throw new TypeError("Frontier Violation");
  return handle.inner;
};
export const promote = (inner) => {
  if (!infer(inner).includes("external"))
    throw new TypeError("Frontier Violation");
  const handle = { type: "regular-handle", inner };
  regions.set(handle, ["bridge", "external"]);
  return handle;
};
const demoteAll = (dirty_array) => dirty_array.map(demote);
export const promoteDirtyArray = (dirty_array) => {
  regions.set(dirty_array, ["bridge"]);
  const handle = { type: "dirty-array-handle", inner: demoteAll(dirty_array), dirty_array };
  regions.set(handle, ["bridge", "internal"]);
  return handle;
};
const adapt = (dirty_function) => (function (...input) {
  if (new.target) {
    return demote(Reflect.construct(dirty_function, input.map(promote), new.target));
  } else {
    return demote(Reflect.apply(dirty_function, promote(this), input.map(promote)));
  }
});
export const promoteDirtyFunction = (dirty_function) => {
  regions.set(dirty_function, ["bridge"]);
  const handle = { type: "dirty-function-handle", inner: adapt(dirty_function), dirty_function };
  regions.set(handle, ["bridge", "internal"]);
  return handle;
};
const isDirtyIndexAccess = (obj, key) =>
  obj.type === "dirty-array-handle" &&                              // Check dirty array
  typeof demote(key) === "number" &&                                // Check index access
  Object.hasOwn(obj.dirty_array, demote(key)) &&                    // Check bounds
  Object.is(demote(obj.dirty_array[demote(key)]), obj[demote(key)]) // Check staleness
export const apply = (callee, that, input) => {
  if (callee.type === "dirty-function-handle") {
    return Reflect.apply(callee.dirty_function, that, input);
  } else if (demote(callee).name === "getValueProperty" && input.length === 2) {
    const [obj, key] = input;
    return isDirtyIndexAccess(obj, key)
      ? obj.dirty_array[demote(key)]
      : promote(obj.inner[demote(key)]);
  } else {
    return promote(Reflect.apply(demote(callee), demote(that), input.map(demote)));
  }
};
export const construct = (callee, input, new_target) => {
  if (callee.type === "dirty-function-handle") {
    return Reflect.construct(callee.dirty_function, input, demote(new_target));
  } else {
    return promote(Reflect.construct(demote(callee), input.map(demote), demote(new_target)));
  }
};
```

Listing 5.19: Frontier module for the adaptor frontier

5.3 Memory-centric frontier designs

In the previous section, we discussed how promoted values can transparently reside in the stack, the store, and the arguments and results of function calls. We now examine how promoted values can reside in the store without sacrificing transparency. This section is organized as follows:

- *Section 5.3.1 discusses the transparency implications of allowing promoted values to reside in the store.*
- *Section 5.3.2 introduces the concept of virtual values and their implementation in mainstream languages.*
- *Section 5.3.3 showcases how virtual values can implement a transitive frontier design to allow promoted values to reside in the store.*
- *Section 5.3.4 explains why semantic knowledge of built-in functions is necessary to actually achieve any precision gain related to the presence of promoted values in the store.*

5.3.1 Challenges of shadowing values in the store

In this section, we examine the challenges associated with shadowing values in the store, whether through shadow memory or sanitization.

The traditional method for shadowing values is shadow execution, which involves maintaining a copy of the memory, referred to as *shadow memory*, to associate metadata with locations in base memory. The mechanism for tracking provenance through argument arrays, as discussed in Section 5.2.3, can be viewed as a specific instance of shadow memory, where argument arrays containing external values are shadowed by arrays containing internal values.

As evidenced by the presence of the staleness check in Listing 5.19, shadow memory may fall out of sync, which is one of the primary shortcomings of shadow execution. In particular, it is unclear how to synchronize the shadow memory when relinquishing control over arbitrary code. This shortcoming likely explains why shadow execution has primarily been proposed in process-wide [72, 83] or even system-wide [138, 14] contexts, where the transfer of control to uninstrumented code can be avoided altogether.

Synchronization of shadow memory is a well-known challenge in the shadow execution of low-level languages that permit direct memory access [139, 55]. However, synchronization issues also occur in managed languages, even when they strictly enforce value abstraction. For example, the JavaScript program in Listing 5.20 creates an object named `gorilla` and passes it to the function `eat`, which represents arbitrary code. Figure 5.7 illustrates the memory mutations performed by the `eat` function. Not only did it flag the `banana` as being eaten, but it also reassigned a property of the `habitat` to the same value. It remains unclear how to synchronize the provenance identifiers in shadow memory after the arbitrary `eat` function has returned.

```
1  // External Region //
2  const eat = (gorilla) => {
3    gorilla.holding.eaten = true;
4    gorilla.habitat.tree = "oak";
5  }
6  // Internal Region //
7  var tree = "oak";
8  const jungle = globalThis;
9  const banana = { eaten: false };
10 const gorilla = { habitat: jungle, holding: banana };
11 eat(gorilla);
```

Listing 5.20: Illustration of the gorilla and banana problem in OOP

It should be noted that Listing 5.20 reflects poor object-oriented programming design. Instead, good practices dictate passing only the `banana` to the `eat` function, which would limit the scope of possible memory writes. However, in practice, passing too much context to object methods is a widespread flaw colloquially known as the "gorilla and banana" problem [102].

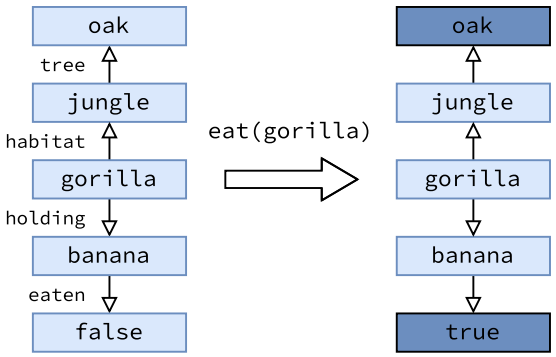


Figure 5.7: Object graph before and after calling the `eat` function from Listing 5.20

> You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

Figure 5.8: Excerpt from an interview with Joe Armstrong [102]

Sanitization is another approach for managing shadow values in the store. Consider Figure 5.9: a first handle (`h1`) points to a dirty reference (`d1`), which contains a second handle (`h2`). This second handle points to another dirty reference (`d2`), which in turn contains a third handle (`h3`) that points to the value (`42`). The first dirty reference (`h1`) can be sanitized into a clean reference (`c1`) by cloning it and recursively sanitizing its property values. Handles for primitive values can be demoted directly.

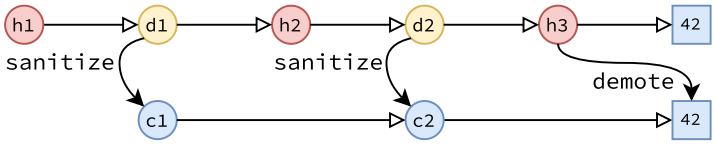


Figure 5.9: Handles for dirty references can be sanitized into clean references

Unfortunately, cloning introduces a two-body problem [34], where the internal and external regions operate on two distinct references instead of a single reference, as is the case in baseline execution. In particular, it remains unclear how mutations in one version should be reflected in the other. Thus, the sanitization process illustrated in Figure 5.9 is only transparent for deeply immutable structures, as would always be the case in pure functional languages.

One might think that the two-body problem can be circumvented by applying demotion in place rather than on a deep clone. However, mutating a dirty reference into a clean reference in place requires notifying all referers of the original dirty reference that it has become clean. Such ripple effects are difficult to reason about and cause provenance loss, as the provenance information initially contained within the dirty reference is irretrievably lost.

To summarize, shadowing values in the store through shadow memory challenges the soundness of the approximation of provenancial equality. And, sanitization-based shadowing of values in the store challenges the transparency of the analysis.

5.3.2 Shallow value virtualization

In this section, we introduce the concept of virtual values, discussing how they can transparently retain promoted values in the store and identifying which mainstream managed languages currently implement them.

The challenge of mediating data exchange between different parts of the program has been extensively studied in the context of software security [27, 98, 78]. Of particular interest are capability-based systems [109, 69], where possession of an unforgeable reference grants access to the underlying resource.

The requirement for unforgeability is essential to ensure that the key cannot be generated by an unauthorized entity. Examples of forgeable references include file paths or pointers in C, which can be created arbitrarily.

Interestingly, in managed languages where value abstraction is consistently enforced, reference values are inherently unforgeable and effectively serve as keys for mediating data exchange. This concept has been investigated to create capability-based systems within a single program [131, 79]. In fact, an entire language called E[2] has been developed largely around this principle [79].

While managed references are essential for enabling robust mediation of data between different code regions, they are insufficient for composing code in a manner that is agnostic to the mediation mechanism. This requirement is essential in our setting because non-instrumented code is, by definition, unaware of the analysis runtime. In addition to managed references, seamless composition requires a reflective mechanism in the target language that supports interposing mediation logic between distinct object graphs. Such reflective capability is categorized as *intercession*, as it enables a program to modify its own behavior.

Fortunately, such an intercession mechanism is available in JavaScript through the standardized `Proxy` API [123, 124], which allows for the redefinition of the semantics of operations within the meta-object protocol (MOP). The act of wrapping a value to control its behavior is sometimes referred to as *virtualization*, with the resulting value termed a virtual value [34, 5].

Listing 5.21 illustrates how proxies can be leveraged within a frontier module. It exports the standard promotion and demotion operations, as well as a virtualization function that converts shallow dirty references (i.e., objects whose property values are all primitive handles) into clean references. As shown, a proxy is instantiated by calling the `%Proxy%` intrinsic as a constructor, with the object it virtualizes referred to as its *target* and an object for intercepting MOP operations referred to as its *handler*.

frontier.mjs

```
export const promote = (inner) => ({ inner }), demote = ({ inner }) => inner;
const get = (obj, key, rec) => demote(Reflect.get(obj, key, rec));
const set = (obj, key, val, rec) => Reflect.set(obj, key, promote(val), rec);
const traps = { get, set };
export const virtualize = (obj) => new Proxy(obj, traps);
```

Listing 5.21: Frontier module that virtualizes shallow dirty references into clean references

Listing 5.22 illustrates how the frontier module from Listing 5.21 can be utilized within analysis-aware code, while Figure 5.10 depicts the resulting object graph.

- The export `$user` is a proxy object that privately points to its target and handler objects. In the figure, these properties start with a `#` character to indicate that they cannot be accessed at the language level, similar to private properties.
- The handler object (`traps`) includes the `get` and `set` properties for overriding the corresponding MOP operations. Note that the virtualization is not complete, as other MOP operations, such as `getOwnPropertyDescriptor`, should also be intercepted.
- The target object (`user`) is a shallow dirty reference that contains the handle `"Bob"@1`. The character `@` is used to denote the handle, distinguishing the inner value (i.e., `"Bob"`) from its provenance (the index `1`).

internal.mjs

```
import { virtualize, promote } from "./frontier.mjs";
const user = { name: promote("Bob") };
export const $user = virtualize(user);
```

Listing 5.22: Utilization of the frontier module from Listing 5.21 in analysis-aware code

Listing 5.23 demonstrates how analysis-unaware code can transparently interact with the virtual user exported from Listing 5.22. Figure 5.11 presents a sequence diagram illustrating this interaction, where colors are used to associate requests and responses (independent of regions).

[2] `http://erights.org/`

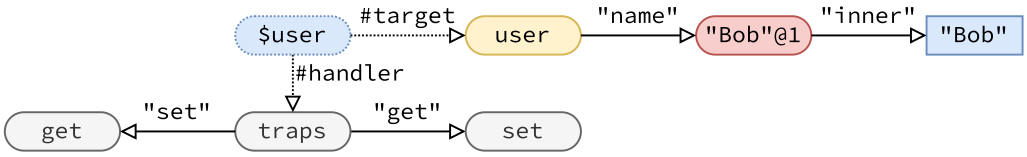


Figure 5.10: Object graph generated from the execution of Listing 5.22

1. Initially, accessing the name property of the virtual user at Line 2 of Listing 5.23 triggers the `get` MOP operation. For consistency, we represent this syntactic operation as a call to the `%Reflect.get%` intrinsic, which we abbreviate as `%get%`.
2. This triggers a nested call to the `get` trap from Listing 5.21, with `user` as the target argument.
3. This results in two sequential lookup operations. The first accesses the name property of the user, yielding the name handle. The second accesses the inner property of the name handle, resulting in a string.
4. That string is used as the result for both nested calls (first `get`, then `%get%`).

external.mjs

```
1 import { $user } from "./internal.mjs";
2 console.log($user.name); // ✅ "Bob"
```

Listing 5.23: Analysis-unaware interaction with the virtual user exported from Listing 5.22

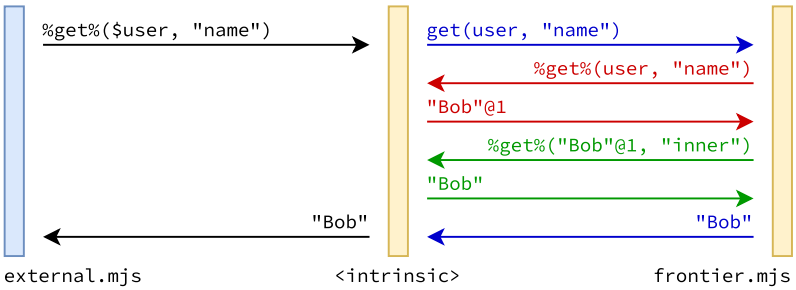


Figure 5.11: Sequence diagram illustrating the execution of Listing 5.23

In contrast to the sanitization conversion discussed in Section 5.3.1, virtual values enable subsequent interactions after being exchanged, which solves the two-body problem and preserves transparency. Crucially, the interaction is mediated by intrinsic functions, which allow external code to remain unaware of the analysis.

As shown in Table 5.1, intercession mechanisms are prevalent in languages that support object-oriented programming. Unfortunately, only the `Proxy` API in JavaScript is comprehensive and highly transparent. For instance, in Python 3, the special methods `__getattr__` and `__setattr__` can be utilized to virtualize objects and insert promotion and demotion operations. However, virtualization is not entirely transparent because these special methods remain accessible and can be bypassed in certain cases[3]. Another example of incomplete virtualization is the `java.lang.reflect.Proxy` API in Java, which can virtualize interfaces but not classes.

[3] `https://docs.python.org/3/reference/datamodel.html#special-lookup`

Language	Intercession Mechanism
JavaScript	`Proxy` with `get`, `set`, `defineProperty`, and `getOwnPropertyDescriptor`
Java	`java.lang.reflect.Proxy` with `InvocationHandler`
C#	`System.Dynamic.DynamicObject` with `TryGetMember` and `TrySetMember`
Smalltalk	`doesNotUnderstand`
Ruby	`method_missing`
Python	`__getattr__` and `__setattr__`
PHP	`__get` and `__set`

Table 5.1: Intercession mechanisms across mainstream managed languages

5.3.3 Transitive frontier design

In this section, we present the blueprint for a transitive frontier that allows handles to reside deeply within data structures while maintaining transparency.

To control access to deeply embedded data, the security community has developed a mechanism in capability-secure languages called membranes [54, 79]. The key idea behind a membrane is to *transitively* virtualize compound values on demand. This process is illustrated in Figure 5.12, which is similar to the sanitization conversion from Figure 5.9 except that clones are replaced with virtual values.

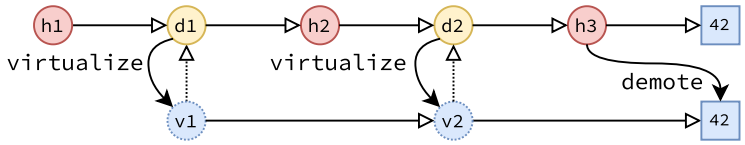


Figure 5.12: Transitive virtualization of compound values

To guide the design of our membrane, we have identified five criteria for categorizing values commonly found in mainstream managed languages. Table 5.2 exemplifies our categorization across several mainstream languages, along with a fictitious representative language at the bottom, which we will use to illustrate our membrane. Our five criteria are:

- Mutability: Refers to the ability to modify data behind a value type in place. For a value to be mutable, its data must be stored indirectly, allowing modifications to be reflected across all appearances of the value within the state. Consequently, mutable value types should be implemented as *references*, while immutable value types can be implemented as *immediates* (cf. inlining in Section 2.3.5).
- Finite Representability: Refers to whether the data behind a value type forms a finite set. A finite set ensures that data items can be encoded as finite sequences in memory. In practice, finite representability is necessary for members of a value type to be implemented as immediates, allowing each value within the state to occupy a fixed length in memory (often 64 bits).
- Freshness: Refers to whether newly created members of a value type are guaranteed to be distinguishable from all other values. Mechanisms for instantiating immediate values cannot ensure unicity. Furthermore, mechanisms for instantiating reference values may not necessarily guarantee unicity due to potential interning optimizations (cf. Section 2.3.6). In mainstream managed languages, mechanisms for instantiating mutable value types guarantee unicity, as failing to do so would introduce global dependencies.

- Non-Compound: Refers to whether the data behind a value type may include other values. Objects, collections, and weak collections are examples of compound value types. In contrast, atomic value types (such as numbers and symbols) contain no other values. Since the data item behind a compound value requires more memory than a single value, compound values are typically implemented as references. These values can be mutable (e.g., `mcons` in Scheme) or immutable (e.g., `cons` in Scheme), whereas atomic values are typically immutable.

- Callability: Refers to whether the members of a value type can be invoked as functions. Although closures require capturing the external scope that contains values, we do not consider this sufficient to classify closures as compound values. To allow mutation of the variables in the captured scope, closures must be allocated uniquely per instantiation. In pure functional programming languages such as Haskell, closures could potentially be instantiated in a non-unique manner; however, Haskell sidesteps this issue by simply disallowing function comparison.

Type	Mutable	Fin. Repr.	Fresh.	Non-Comp.	Callable
JavaScript					
`boolean` / `number`	❌	✅	❌	✅	❌
`bigint` / `string`	❌	❌	❌	✅	❌
`symbol`	❌	❌	✅	✅	❌
`object` (not `null`)	✅	❌	✅	❌	❌
`function`	✅	❌	✅	❌	✅
Scheme					
`cons`	❌	❌	✅	❌	❌
`mcons`	✅	❌	✅	❌	❌
`procedure`	❌	❌	✅	✅	✅
Python					
`int` ∈ `[-5, 256]`	❌	✅	❌	✅	❌
`int` ∉ `[-5, 256]`	❌	❌	✅	✅	❌
`object`	✅	✅	✅	✅	✅
Haskell					
`Int`	❌	✅	❌	✅	❌
`ADT`	❌	❌	❌	❌	❌
`function`	❌	❌	-	✅	✅
Representative Language					
`primitive`	❌	(✅/❌)	❌	✅	❌
`symbol`	❌	(✅/❌)	✅	✅	❌
`record`	✅	❌	✅	❌	❌
`procedure`	❌	❌	✅	✅	✅

Table 5.2: Classification of values based on five criteria related to frontiers

Figure 5.13 illustrates the conversions performed by our membrane for a representative language, featuring the value types listed at the bottom of Table 5.2. The figure reads as follows:

- Primitive: Represents value types that are immutable, not uniquely instantiated, atomic, and not callable. Promotion should involve wrapping them in a fresh record to ensure sound provenance tracking.

- Symbol: Represents value types similar to primitives, except that instantiation mechanisms guarantee uniqueness. To enhance the precision of provenance tracking, promotion should involve a global registry to ensure that the same handle is reused for any given symbol. Since each newly created symbol is unique, this does not compromise soundness.

- Plain Record: Represents value types that are mutable, uniquely instantiated, compound with external values, and not callable. Similar to symbols, promotion should involve a registry to avoid introducing multiple handles for the same record.

- Plain Function: Represents value types that are immutable, uniquely instantiated, atomic, and callable with external values as input and output. Again, uniqueness should be preserved with a registry during promotion.
- Dirty Record: Similar to plain records but may contain handles. They can be transitively virtualized (cf. Section 5.3.2) into records suitable for residing in the external region. This process requires a built-in virtualization mechanism and should involve registration to enable recovering the dirty record from the dirty record handle. For deeply immutable records (as is always the case in pure functional programming languages), virtualization can be replaced with sanitization (cf. Section 5.3.1).
- Dirty Function: Similar to plain functions but callable with handles as input and output. They can be adapted into functions suitable for residing in the external region (cf. Section 5.2.3). This process should involve registration to enable recovering the dirty function from the dirty function handle.

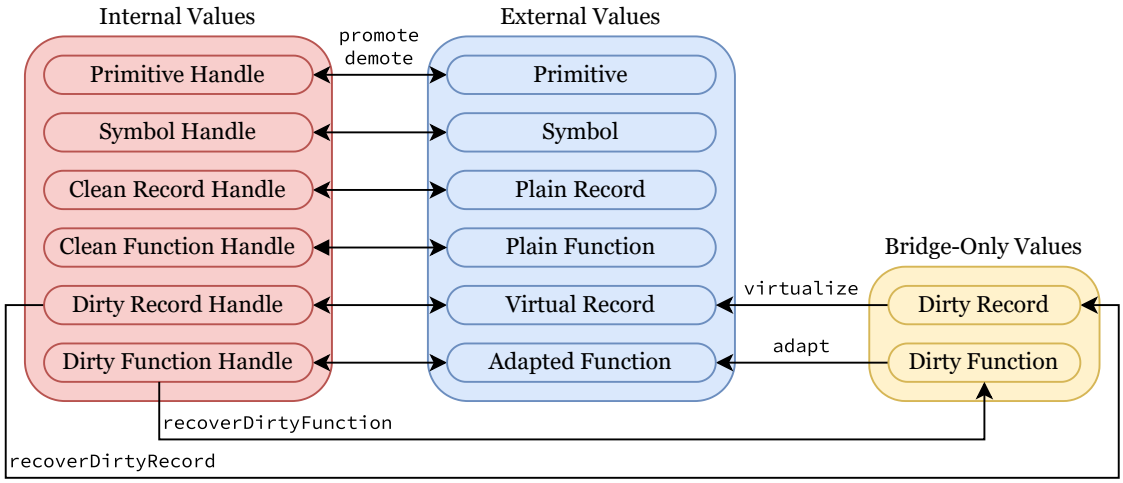


Figure 5.13: Language-agnostic application domain of our membrane

5.3.4 An oracle to enhance precision

In this section, we explain the necessity of an oracle to achieve a gain in precision with respect to handles within the store.

In the previous section, we focused on enabling the presence of handles in the store in a transparent manner. However, to achieve a gain in precision, it is sometimes necessary to bypass the default application of the frontier, which would result in chaining demotion and re-promotion operations. For instance, when invoking a built-in function to retrieve an item from a dirty collection, the provenance of the result should match that of the item. However, the default application of our membrane assigns a fresh provenance to the result, which is sound but incomplete.

Consider Figure 5.14, which illustrates the object graph resulting from promoting a dirty object (named `obj`) that contains a handle for the number `123`. The naming conventions follow those used in the object graph from Figure 5.10. Assume that an analysis has been set up with an advice module located at `advice.mjs` and a frontier module at `frontier.mjs`. Figure 5.15 depicts a sequence diagram for the process of requesting the frontier to apply `%Reflect.get%` on the handle `$obj@0`:

1. Initially, the advice requests the frontier to apply `%Reflect.get%` on the handles `%undefined%@1` (`this` argument), `$obj@0` (first argument), and `"foo"@1` (second argument).
2. Without knowledge of the semantics of `%Reflect.get%`, the frontier ensures transparency by demoting all involved values and forwarding the function call.

3. The engine detects that the first argument is a proxy object (`$obj`) and triggers the relevant trap (`get`) of its handler object, passing it the target object (`obj`).
4. Within the implementation of the `get` trap, the frontier looks up the property of `obj` at `"foo"`, to which the engine responds with the handle `"foo"@1`.
5. Because the `get` trap is designed to be invoked in an external call context, it demotes its result value.
6. This causes the engine to return a number (`123`) for the initial lookup.
7. Finally, to ensure soundness, the frontier must promote this number into a new handle (`123@2`), resulting in a loss of precision, as `123@1` would have been correct.

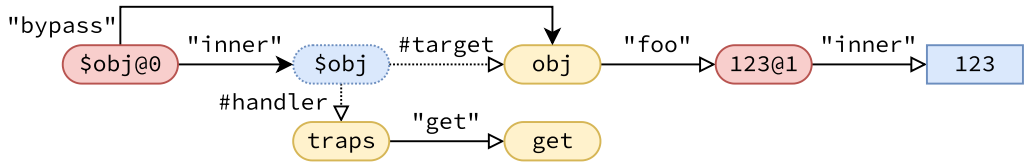


Figure 5.14: Object graph after promoting a dirty object containing a handle for `123`

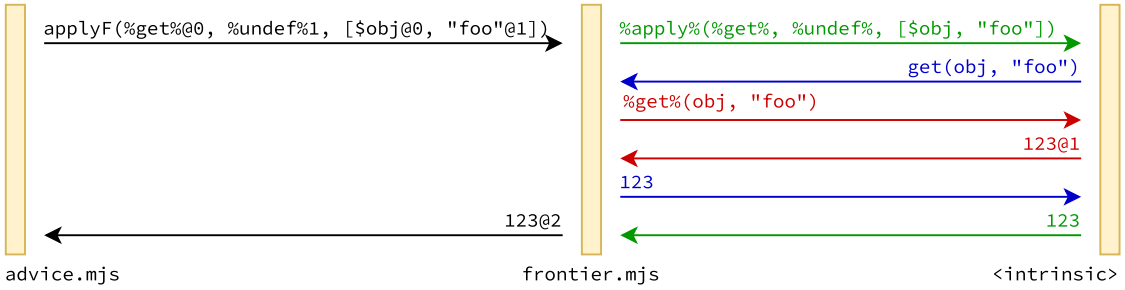


Figure 5.15: Default application of `%Reflect.get%` within our membrane

While there is no general solution to this problem, precision can be enhanced when the logic of the external function is known. This applies to intrinsic functions such as `%Reflect.get%`, which are defined in the ECMAScript specification[4] and are guaranteed to have stable semantics across different runtimes. Figure 5.16 depicts a sequence diagram similar to that in Figure 5.15, except that the frontier has knowledge of the semantics of `%Reflect.get%`. This knowledge can be leveraged by the frontier to forward the lookup directly to the dirty object (`obj`) rather than to the virtual clean object (`$obj`), thereby improving precision.

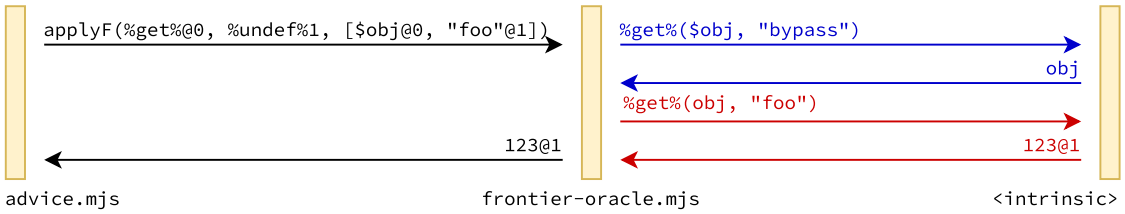


Figure 5.16: Oracle-enabled application of `%Reflect.get%` within our membrane

Because Aran normalizes many syntactic constructs, such as `MemberExpression`, into calls to members of the `%Reflect%` object, it is essential to include these reflective functions in the oracle. However, incorporating other intrinsic functions into the oracle is also beneficial. Consider the well-known intrinsic `%Array.prototype.map%`, which applies a given function to all items in an array and returns a new array containing the results. In Listing 5.24, this intrinsic function is invoked with the identity function, producing an output array in which each item retains the same provenance as the corresponding item in the input array; specifically, `&y0` shares the same provenance as `&x0`.

[4] `https://tc39.es/ecma262/#sec-reflect.get`

```
// @ts-nocheck
const id = (x) => x;
const x0 = 123;
const xs = [x0];
const ys = xs.map(id);
const y0 = ys[0]; // 123
```

Listing 5.24: Default application `%Array.prototype.map%` on `&xs` and the identity function

Figure 5.17 illustrates the object graph generated by executing Listing 5.24 within the standard analysis setup. Figure 5.18 presents a sequence diagram that explains the provenance loss associated with the default application of `%Array.prototype.map%`. The sequence is as follows:

1. Initially, the advice requests the frontier to apply `%Array.prototype.map%` on the handles `$xs@0` (`this` argument) and `$id@0` (first argument).
2. The default behavior of the frontier is to demote all involved values and forward the function call.
3. Since the `this` argument is a proxy, the engine triggers the `get` trap back to the frontier to obtain the length of the array.
4. Based on the length value, the engine triggers the `get` trap a second time to retrieve the sole item of the array. The demotion of this item constitutes the first provenance loss.
5. The engine then applies the mapping function, which is also a proxy, causing the engine to trigger the `apply` trap back to the frontier.
6. The frontier transforms the external call into an internal call and forwards the application to the advice.
7. The advice applies the identity function and returns the provided argument.
8. The frontier demotes the result of the identity function, resulting in a second provenance loss.
9. The engine returns a clean array containing the number `123`.
10. Subsequent accesses to the clean array create new handles.

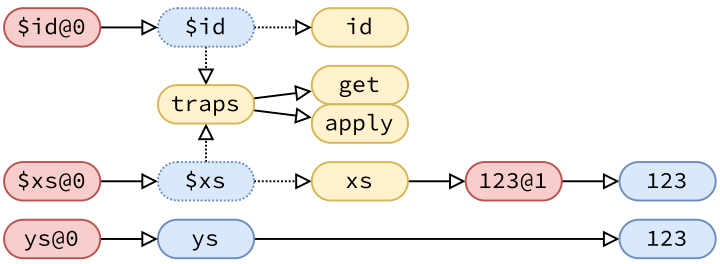


Figure 5.17: Object graph generated by executing Listing 5.24 in the standard analysis setup

Similarly to `%Reflect.get%`, precision can be enhanced by leveraging the semantics of `%Array.prototype.map%`. This oracle should have two primary effects. First, the virtual clean identity function should be bypassed to directly call the dirty identity function with handles. Second, the resulting array should be made into a dirty object, which requires the absence of aliases—a condition that cannot be guaranteed for arbitrary functions.

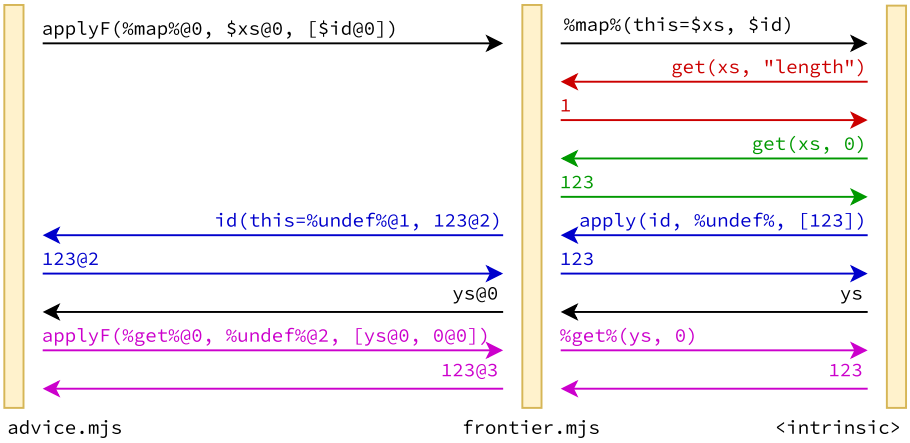


Figure 5.18: Default application of `%Array.prototype.map%` within our membrane

5.4 Applications of provenancial equality

In the previous sections, we discussed how to approximate provenancial equality by promoting values. In this section, we explore the applications of provenancial equality, both in manual use and in the field of dynamic program analysis. This section is organized as follows:

- *Section 5.4.1 discusses the manual use of provenancial equality both during development and in production.*
- *Section 5.4.2 presents a first approach to building provenance-aware analysis via advice extension.*
- *Section 5.4.3 introduces a second approach for developing provenance-aware analysis via advice layering.*
- *Section 5.4.4 further examines advice layering and its relation to existing work.*

5.4.1 Manual use of provenancial equality

In this section, we discuss the simplest application of provenancial equality, which involves exposing it to the target program.

The simplest way to demonstrate the usefulness of provenancial equality arises when the target program is already utilizing it prior to instrumentation. We present two such use cases: one intended for production and the other for development.

Our first manual use case of a provenance provider is in the context of reactive programming. In this paradigm, data dependencies are defined declaratively, enabling automatic change propagation. Typically, changes are propagated only when the data has changed, which enhances performance and prevents infinite cycles. However, if the provenance of the data has changed while its content remains the same, it may still be advantageous to propagate the change along the reactive chain.

For instance, consider Listing 5.25, which creates a reactive value for the user's email, initially set to a default value. Providing a visual cue when the user confirms the default email can be beneficial. While this could be achieved by always propagating changes, it risks exposing the consumer of the `createSignal` function to infinite cycles. Provenancial equality facilitates such propagation without altering the data domain and while remaining resistant to cycles.

Because provenancial equality is conceptually straightforward, it is easy to overlook that it relies on complex mechanisms to reverse-engineer performance optimizations, such as value interning and value inlining, as discussed in Section 2.3. Admittedly, this hidden complexity incurs performance overhead;

```
import { provEq } from "./provenance.mjs";
import { log } from "node:console";

const createSignal = (value, observers) => [
  () => value,
  (new_value) => {
    if (provEq(new_value, value)) return;
    value = new_value;
    for (const notify of observers) notify();
  },
];

const getDefaultEmail = (user) => `${user}@example.com`;
const [getEmail, setEmail] = createSignal(
  getDefaultEmail("alice"),
  [() => { log("Email completed by the user", getEmail()) }],
);

setEmail("alice@example.com"); // ✅ Trigger visual cue
setEmail(getEmail());          // ✅ No infinite cycle
```

Listing 5.25: Reactive change propagation based on provenance

therefore, we believe that provenancial equality is best utilized during development rather than in production environments. In this context, another valuable use case emerges: diagnosing the presence of unexpected values. Here, referential equality may not always suffice, and debugging can benefit from a stricter notion of equality that takes provenance into account.

For instance, consider Listing 5.26, which computes the speed from a flattened list of triples of numbers. During execution, a division-by-zero error occurs. To diagnose why the variable `b` was assigned the value zero, we can iterate through the data and log the index of the element that is provenancially equal to the current value of `b`. This reveals that the faulty index is 5—which is correct. Without provenancial equality, structural equality would implicate all indices with a value of zero, namely indices 0 and 5. In contrast, provenancial equality allows us to precisely identify the origin of the faulty data.

```
import { provEq } from "provenance-provider";

const data = [
  0, 5, 0.123,   // 0.123s to travel 5m
  5, 11, 0,      // 0s to travel 6m >> Invalid Time!
  11, 17, 0.456, // 0.456s to travel 6m
];

const divide = (a, b) => {
  if (b === 0) {
    console.warn("Division by zero");
    for (let index = 0; index < data.length; index++)
      if (provEq(data[index], b))
        console.warn("Faulty position", index);
  }
  return a / b;
}

for (let index = 0; index < data.length; index += 3) {
  const distance = data[index + 1] - data[index];
  const time = data[index + 2];
  console.log("Speed:", divide(distance, time));
}
```

↓↓↓

```
Speed: 40.65040650406504
Division by zero
Faulty position 5
Speed: Infinity
Speed: 13.157894736842104
```

Listing 5.26: Debugging the occurrence of zero using provenancial equality

While the manual use of provenancial equality presents an interesting case that merits further exploration, this dissertation primarily focuses on dynamically analyzing programs that are independent of any provenance provider. Next, we will examine two main approaches for integrating provenance tracking into analysis-related logic.

5.4.2 Ahead-of-time evaluation via advice extension

In this section, we present a straightforward approach for constructing provenance-aware analyses by introducing a simple analysis that identifies candidates for ahead-of-time evaluation.

Our example analysis involves identifying run-time values that are suitable candidates for ahead-of-time evaluation, thereby enabling performance optimizations. In the literature, such candidates are typically identified statically during compilation as part of a technique known as partial evaluation [56]. However, not all suitable candidates for ahead-of-time evaluation can be detected statically, and dynamically identifying them remains of practical interest.

Figure 5.19 depicts the module dependency graph of the analysis. This layout resembles that of Figure 5.3, except that the provenance provider has been removed and a module for generating a generic advice to enforce the frontier has been added. The modules in grey perform instrumentation and are executed at compile time, so they do not belong to any specific region. The original version of the target program generates the internal region (red). The instrumented version of the target program manipulates a mix of internal and external values (including dirty references) and is associated with the bridge region (yellow). All other files of the analysis also generate state locations in the bridge region (yellow). The node runtime is made globally available and should only manipulate external values (blue).

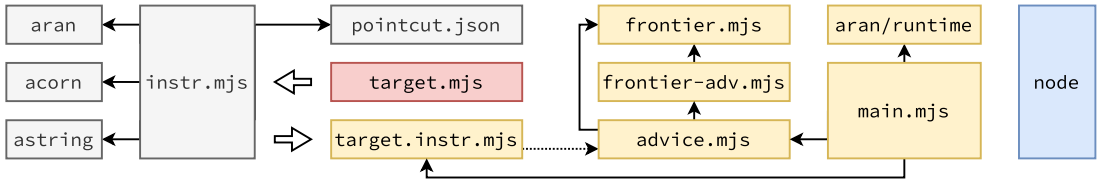


Figure 5.19: Module dependency graph to identify candidates for ahead-of-time evaluation

For brevity, we omit `frontier-adv.mjs` as it closely resembles the advice from Listing 5.17. We also omit `main.mjs` and `instr.mjs`, which contain boilerplate code for setting up the analysis.

Listing 5.27 presents `advice.mjs`, which extends the generic advice from the frontier module in two ways. First, Lines 7–11 override `primitive@after` to tag primitive handles originating from literals as constant. Second, Lines 12–19 override `apply@around` to report the call when all inputs are constants, marking the result as constant as well.

advice.mjs

```
1  import { log } from "node:console";
2  import { promote, demote, apply } from "./frontier.mjs";
3  import { frontier_advice } from "./frontier-adv.mjs";
4  const constants = new WeakSet();
5  export const advice = {
6    ...frontier_advice,
7    "primitive@after": (_state, primitive, _location) => {
8      const handle = promote(primitive);
9      constants.add(handle);
10     return handle;
11   },
12   "apply@around": (_state, callee, that, input, location) => {
13     const output = apply(callee, that, input);
14     if (input.every((arg) => constants.has(arg))) {
15       constants.add(output);
16       log(`AOT ${demote(output)} at ${location}`);
17     }
18     return promote(output);
19   }
20 };
```

Listing 5.27: Advice extension to identify candidates for ahead-of-time evaluation

Listing 5.28 presents the target program which calculates the circumference of a circle using the formula $2 \times \pi \times r$, where r is the radius of the circle provided as a command-line argument. As expected, the analysis reports the multiplication $2 \times \pi$ at line 3 as a candidate for ahead-of-time evaluation.

target.mjs

```
import { log } from "node:console";
const PI = 3.1416;
log(`Circumference: ${2 * PI * process.argv[2]}`);
```

↓↓↓

```
> node main.mjs 5
AOT 6.2832 at target.mjs#$body[3]...
Circumference: 31.416
```

Listing 5.28: Analysis of a program for computing the circumference of a circle

5.4.3 Ahead-of-time evaluation via advice layering

In this section, we discuss an alternative approach to building provenance-aware analyses by adapting our analysis for detecting candidates for ahead-of-time evaluation.

While the logic presented in Listing 5.27 is clear, it conflates two distinct concerns: the analysis concern (responsible for propagating constants) and the frontier concern (responsible for tracking provenance). For instance, when a primitive literal is encountered, two actions are performed. First, the value is promoted, which relates to the frontier concern. Then, the handle is registered as a constant in the global registry, which relates to the analysis concern. In our experience, developing analysis logic that manipulates both internal (i.e., handles) and external values is challenging, error-prone, and difficult to maintain.

In Section 6.1, we describe how static typing can help mitigate some of these issues; however, it is not always sufficient, as type systems struggle to analyze reflective code [67], such as in the case of advices. To truly address this problem, the frontier concern must be separated from the analysis concern and subsequently composed as independently as possible. A radical solution to achieve such separation of concerns is to move the analysis advice from the bridge region to the internal region where only handles are allowed. This enables the advice to leverage provenancial equality similarly to the target programs of our manual use cases from Section 5.4.1.

Figure 5.20 illustrates the overall instrumentation required for composing advices in layers. Initially, the target program is woven with the analysis logic. Subsequently, the analysis-woven program (i.e., the combination of the instrumented version of the target program and the analysis advice) is woven with the frontier logic to track provenance. In other words, it is now the analysis-woven program that generates the internal region, rather than the original version of the target program. It is important to note that the order of weaving is crucial; reversing it would violate the frontier.

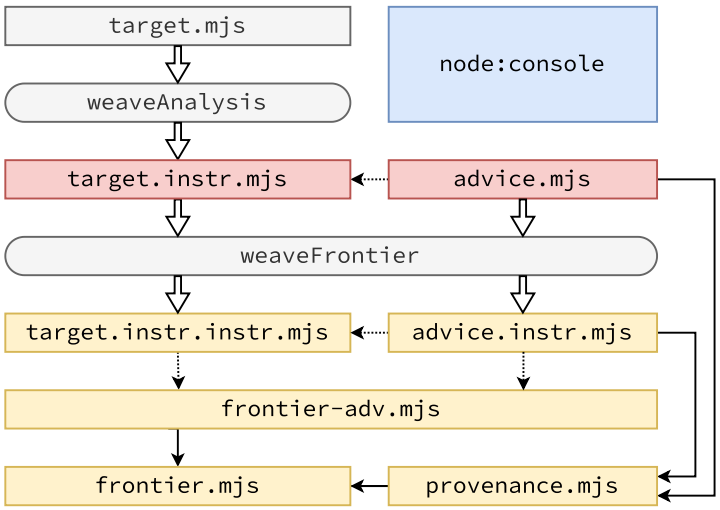


Figure 5.20: The double weaving required for layering advice

Listing 5.29 presents the revised version of the `advice.mjs` file from Listing 5.27. It closely resembles its predecessor, except that it imports a provenance provider rather than a frontier module and that it

no longer includes promotion and demotion operations. The import is reduced to `ProvWeakMap`, which is assumed to be a weak map capable of tracking the provenance of its keys and values. Behind the scenes, such a collection is implemented as a regular weak map whose keys and values are handles, but this complexity is hidden from the advice. Note that function applications are now directed to `%Reflect.apply%` instead of the `apply` export from the frontier. This means that the oracle must possess knowledge of `%Reflect.apply%` to maintain precision.

advice.mjs

```
import { log } from "node:console";
import { ProvWeakSet } from "./provenance.mjs";
const constants = new ProvWeakSet();
export const advice = {
  "primitive@after": (_state, primitive, _location) => {
    constants.add(primitive);
    return primitive;
  },
  "apply@around": (_state, callee, that, input, location) => {
    const output = Reflect.apply(callee, that, input);
    if (input.every((arg) => constants.has(arg))) {
      constants.add(output);
      log(`AOT ${output} at ${location}`);
    }
    return output;
  }
};
```

Listing 5.29: Advice layering for identifying candidates for ahead-of-time evaluation

While both advice extension and advice layering yield identical reports, they differ significantly in their internal mechanisms and present distinct trade-offs.

- The main advantage of advice layering over advice extension is that it separates frontier concerns from the analysis advice, thereby simplifying its logic. Although deployment and instrumentation have become more complex, this part of the analysis is more straightforward and less error-prone. Specifically, it does not require managing a mix of internal and external values, which was the issue identified at the beginning of this section.
- The main drawback of advice layering compared to advice extension is that it incurs additional performance overhead. For example, consider the instrumentation of the primitive `123` shown in Listing 5.30. The advice extension approach requires only a single initial pass that involves invoking the `primitive@after` advice function. However, advice layering requires an additional pass to enforce the frontier. As shown, these passes have a multiplicative effect on the size of the instrumented code, which is expected to incur performance overhead. In Section 6.4, we compare the performance overhead of both composition approaches.

We believe that both approaches have their merits: advice layering is advantageous for rapid prototyping and for novice analysis builders, while advice extension is better suited for analyzing time-sensitive or computation-intensive programs due to its lower performance overhead.

```
123;
```

⇓ ⇓ ⇓

```
%aran.getValueProperty%(_ANALYSIS_ADVICE_, "primitive@after")(
  _state_,
  123,
  "$.body.0.expression",
);
```

⇓ ⇓ ⇓

```
_FRONTIER_APPLY_(
  _FRONTIER_APPLY_(
    _FRONTIER_PROMOTE_(%aran.getValueProperty%),
    _FRONTIER_PROMOTE_(%undefined%),
    %Array.of%(
      _FRONTIER_PROMOTE_(_ANALYSIS_ADVICE_),
      _FRONTIER_PROMOTE_("primitive@after"),
    ),
  ),
  _FRONTIER_PROMOTE_(%undefined%),
  %Array.of%(
    _state_,
    _FRONTIER_PROMOTE_(123),
    _FRONTIER_PROMOTOE("$.body.0.expression"),
  ),
);
```

Listing 5.30: Double weaving of `123` in AranLang

5.4.4 Related work on advice composition

> *In this section, we elaborate on advice layering and its relationship to existing work in aspect-oriented programming, reflective programming, and instrumentation.*

The technique described in Section 5.4.3 separates concerns by organizing them into distinct layers. Figure 5.21 illustrates the resulting layered architecture. The base layer is responsible for executing the logic of the target programs and remains completely unaware of the layers above it. The meta layer is divided into two components: the analysis layer, which executes the logic of the analysis, and the provenance layer, which enforces the frontier. The analysis layer operates largely independently of the provenance layer, except for its explicit import of functionalities from the provenance provider.

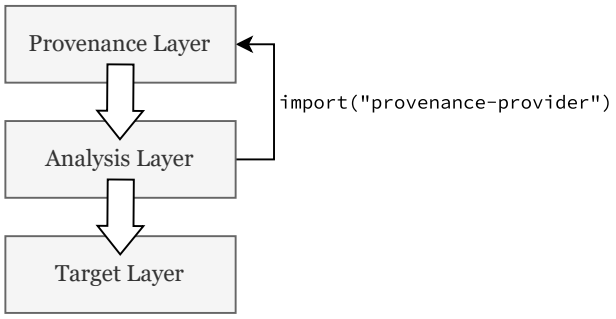


Figure 5.21: Layered architecture resulting from our advice composition technique

Layered architectures have been explored in the past. Both previous work and our own begin with the observation that the meta layer can be reasoned about and can serve as the target of another meta layer. A straightforward approach involves maintaining only two layers, which requires the meta layer to reason about itself. In aspect-oriented programming (AOP), this means that the pointcut of an aspect can be applied to its own advice. While this is feasible, it can lead to issues such as infinite recursion. To mitigate this problem, a more structured approach is recommended, where each layer can only reason about the layer below it, thereby eliminating recursion-inducing cycles.

Layered architectures were first explored in the field of reflective programming through reflective towers [113, 51, 74]. Reflective towers are implemented by stacking meta-circular interpreters, which enforce a complete separation between layers. If reflective towers were applied to our problem domain, they

would enforce the frontier by design. However, they would also hinder selective provenance tracking, as they lack natural mechanisms to expose computations from the layer below to the layers above.

Layered architectures have also been explored in the field of aspect-oriented programming (AOP) with execution levels [118, 119]. Unlike our approach and reflective towers, execution levels allow for dynamic switching between layers at run-time. While this capability is powerful, it necessitates a run-time environment to manage coexisting execution levels, which incurs performance overhead. In the future, it would be interesting to investigate how execution levels would fare in our problem domain. Although we believe that execution levels will not facilitate the enforcement of the frontier as effectively as advice layering, it is possible that the runtime of execution levels incurs less performance overhead than the double weaving prescribed by our approach.

Finally, various approaches to combine program instrumentation have been proposed [129, 126, 45]; they primarily focus on ensuring non-interference between analyses. To the best of our knowledge, the concept of stacking instrumentation to separate analysis concerns into distinct layers has not been previously introduced.

5.5 Measuring the precision of provenance tracking

Previously, we qualitatively discussed the precision gained by expanding the internal region where handles are allowed. In this section, we explore how to quantitatively evaluate the precision of sound approximations of provenancial equality. This section is organized as follows:

- *Section 5.5.1 defines a novel metric for measuring the precision of provenance tracking.*
- *Section 5.5.2 makes our definition more concrete by presenting an analysis for computing our metric.*

5.5.1 Introduction to our provenance metric

In this section, we introduce a tree structure for quantifying the accuracy of a provenance provider independently from consuming analysis.

Let's assume that an interpreter has been modified to introduce provenancial equality as an additional syntactic construct (denoted as `====` with four equal signs), while preserving behavior. As discussed in Section 2.3, making this operator available in the language necessitates reverse engineering important performance optimizations, such as data inlining and value interning.

In this setup, we approximate provenancial equality through value promotion normally, but with the essential requirement of not relying on the ideal provenancial equality operator (`====`). To evaluate the precision of this approximation, we can measure the rate of incorrect results. This is exemplified in Listing 5.31, where the `provEq` export of the provenance provider has been instrumented to record unsound and incomplete answers based on the result of the exact provenancial equality operator.

```
import { provEq as provEqApprox } from "provenance-provider";

export const provEq = (value1, value2) => {
  const approx = provEqApprox(value1, value2);
  const actual = value1 ==== value2; // Exact provenancial equality comparison
  if (approx && !actual)
    console.log("unsound comparison for ", { value1, value2});
  if (!approx && actual)
    console.log("incomplete comparison for ", { value1, value2});
  return approx;
};
```

Listing 5.31: Canonical precision measure for provenance equality

However, as noted in Section 2.2.3, modifying the interpreter renders the approach non-portable across language runtimes, which presents maintenance challenges. The portability criterion is somewhat less critical in this context, as the goal is to benchmark the provenance provider in a controlled environment rather than to deploy the analysis in production-like setups. Nonetheless, as discussed in Section 2.3.2, reverse-engineering inlining and interning is not trivial. Therefore, we believe that developing a portable approach to measure the precision of an approximation of provenancial equality remains of practical interest.

Our approach involves using the approximation of provenancial equality to associate primitives with a score that positively correlates with the precision of the approximation. In other words, for a given interpreter state, as the approximation becomes more precise, the score associated with each primitive in the state must either increase or remain constant. References are excluded from the metric, as referential equality already serves as a perfect approximation for them.

Our provenance metric is based on the amount of information known about the provenance of each individual run-time value. **Therefore, our provenance metric captures provenance information at the level of individual values, not entire executions.** It quantifies the size of a structure resembling a value flow graph (cf. the tracking of `NaN` from Section 2.3.4), which we refer to as a *provenance tree*. The leaves of our provenance tree consist of either reference values or primitive values originating from atomic nodes, such as literals and intrinsic expressions. The nodes of our provenance tree correspond to syntactic function applications, where the provenance trees of both the arguments and the result of the call serve as parents of the node. The parent node for the result may be left empty to indicate an unknown origin. Since the approximation is sound, adding more nodes to this tree can only occur by improving the precision of the approximation. This hypothesis is experimentally verified in Section 6.5.3.

Consider Listing 5.32, which computes the square root of a literal number, assigns it to the property of an object, and retrieves it. Figure 5.22 illustrates two possible provenance trees associated with the value of the variable `num2`. Without provenance tracking through the store, the result of the call to `%Reflect.get%` would have a fresh provenance, meaning that the corresponding apply node will lack a result parent node. In contrast, if provenance is tracked through the store, the result of the `%Reflect.get%` call would have been previously observed and associated with the call to `%Math.sqrt%`, thereby increasing the size of the provenance tree.

```
1 const num1 = Math.sqrt(123);
2 const obj = {};
3 Reflect.set(obj, "foo", num1);
4 const num2 = Reflect.get(obj, "foo");
```

Listing 5.32: A simple program to exemplify the content of provenance trees

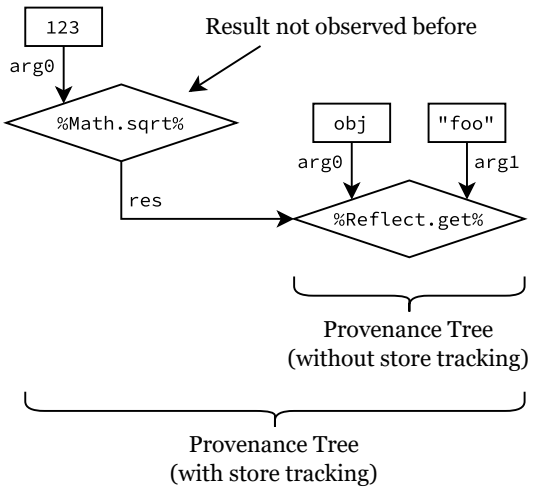


Figure 5.22: Two possible provenance trees for the value of `num2` from Listing 5.32

One of the main themes of this dissertation is to separate the concerns of tracking the provenance of primitive values from the analysis logic. We envision a future where research can be conducted independently between provenance providers and consuming analyses. However, this vision can only be truly achieved if the precision of the provenance provider can be assessed independently from analyses. It is for this reason that we believe our provenance metric is an essential component of our approach, even though it remains preliminary and subject to revision.

5.5.2 Implementation of our provenance metric

> *In this section, we define how to compute our provenance metric and illustrate this process through an analysis based on advice layering.*

Our provenance metric evaluates a provenance tree as follows: each apply node and each leaf primitive node are awarded one point, while each leaf reference node is assigned zero points. We chose not to account for leaf reference nodes because the provenance of a reference is already tracked by the runtime.

Listing 5.33 presents an analysis for computing our provenance metric based on our advice-layering architecture from Section 5.4.3. The advice imports two functionalities from the provenance provider: `ProvWeakMap`, which serves as a constructor for provenance-sensitive weak maps, and `resetProvenance`, which returns its argument with fresh provenance.

- Line 6 instantiates a provenance-sensitive weak map to create a global registry that associates primitive values with their provenance scores.
- Lines 7–12 define two accessors for retrieving and updating the score of a value.
- Lines 14–15 assign a score of one point to primitives originating from intrinsic expressions (`intrinsic@after`) and literal expressions (`primitive@after`). Note that other `@after` join points, such as `await@after`, are not implemented, meaning their result values retain the default score of zero points. This accounts for the fact that such values have an unknown origin and should not be rewarded by the metric.
- Lines 16–26 assign a score to the result of function calls (`apply@around`) after resetting their provenance. Resetting the provenance of the result is crucial to prevent inadvertently increasing the score elsewhere. For example, we want `obj.foo` to be counted in the result but not within the object itself, ensuring that subsequent accesses return the same score. Note that there is no need to advise the `construct@after` join point, as it always computes a reference, which is not accounted for in our metric.
- Lines 27–30 log the provenance score of the program's completion value.

Next, we showcase computations of our provenance metric across four increasingly precise provenance providers:

- `stack`: Stack-only provenance tracking.
- `intra`: Intra-procedural provenance tracking (cf. Section 5.2.1), which extends the internal region to the environment.
- `inter`: Inter-procedural provenance tracking (cf. Section 5.2.3), which extends the internal region to arguments and results.
- `store`: Heap-precise provenance tracking (cf. Section 5.2.3), which extends the internal region to locations in the store.

Table 5.3 presents the provenance scores for the values of standalone expressions across our four provenance providers. These scores agree with the increasing precision of each provenance provider and are computed as follows:

```
import { dir } from "node:console";
import { ProvWeakMap, resetProvenance } from "provenance-provider";
const add = (x, y) => x + y;
const isPrimitive = (value) =>
  (value === null || typeof value !== "object") && typeof value !== "function";
const registry = new ProvWeakMap();
const getScore = (value) => registry.get(value) ?? 0;
const setScore = (value, score) => {
  if (isPrimitive(value)) // References are not scored.
    registry.set(value, score);
  return value;
};
export const advice = {
  "primitive@after": (_state, value, _location) => setScore(value, 1),
  "intrinsic@after": (_state, _name, value, _location) => setScore(value, 1),
  "apply@around": (_state, callee, that, input, _location) => {
    const result = Reflect.apply(callee, that, input);
    return setScore(
      resetProvenance(result),             // The current score should not propagate
      1                                      // Score the apply node itself (1 point)
        + getScore(callee)                   // Score the callee (should always be 0 be)
        + getScore(that)                     // Score the this argument
        + input.map(getScore).reduce(add, 0) // Score arguments
        + getScore(result)                   // Score the result
    );
  },
  "program-block@after": (_state, _kind, completion, _location) => {
    dir(getScore(completion));
    return completion;
  },
};
```

Listing 5.33: Advice layering for calculating our provenance metric

- `Math.sqrt(123)`: Tracking provenance through the value stack is sufficient to maximize the provenance score, which is calculated by summing the following terms: (a) one point for the call to `%Math.sqrt%`, (b) zero points for the `this` argument (`%Math%`) being a reference, (c) one point for the first argument (`123`) originating from a primitive literal, (d) zero points for the result being of unknown origin.

- `123 + 456`: Again, tracking provenance through the value stack suffices to maximize the metric. The provenance score of five is derived from the normalization of the code into a call to `%aran.performBinaryOperation%`. The score is then calculated by summing the following terms: (a) one point for the call to `%aran.performBinaryOperation%`, (b) one point for the `this` argument (`%undefined%`) originating from an intrinsic literal, (c) one point for the first argument (`"+"`) originating from a primitive literal, (d) one point for the second argument (`123`) originating from a primitive literal, (e) one point for the third argument (`456`) originating from a primitive literal, (f) zero points for the result being of unknown origin.

- `const x = 123; x`: The provenance score is either one (literal origin) or zero (unknown origin), depending on whether provenance is tracked through both the scope and the stack.

- `((x) => x)(123)`: Assuming that provenance is tracked through the stack, the provenance score is calculated by summing the following terms: (a) one point for the call to the identity function, (b) one point for the `this` argument (`%undefined%`) originating from an intrinsic literal, (c) one point for the first argument (`123`) originating from a primitive literal, (d) either one point (literal origin) or zero points (unknown origin) for the result, depending on whether provenance is tracked inter-procedurally.

- `Reflect.get({foo:123}, "foo")`: Assuming that provenance is tracked through the stack, the provenance score is calculated by summing the following terms: (a) one point for the call to `%Reflect.get%`, (b) zero points for the `this` argument (`%Reflect%`) being a reference, (c) zero points for the first argument (the target object) being a reference, (d) one point for the second argument (the key `"foo"`) originating from a primitive literal, (e) either one point (literal origin) or zero points (unknown origin) for the result, depending on whether provenance is tracked through the store.

Target	stack	intra	inter	store
`Math.sqrt(123);`	2	2	2	2
`123 + 456;`	5	5	5	5
`const x = 123; x;`	0	1	1	1
`((x) => x)(123);`	3	3	4	4
`Reflect.get({foo:123}, "foo");`	2	2	2	3

Table 5.3: Provenance scores for the values of various standalone expressions

5.6 Conclusion

In this chapter, we discussed tracking the provenance of primitive values by promoting them to handles, a technique that has received less attention than shadow execution in the literature. We also explored various frontier designs for systematically performing promotion and demotion operations. In particular, we adapted a pattern known as membrane to enable the presence of promoted values in the store. We then detailed applications of provenancial equality in the field of dynamic program analysis. Finally, we proposed a new metric to quantify the precision of provenance tracking. To conclude, we outline contributions, review related work, and highlight future research directions.

Main contributions

The main contribution of this chapter is to demonstrate that provenancial equality provides a suitable interface between two core components of any provenance-aware dynamic program analysis: the provenance provider and the analysis-specific logic. Compared to monolithic implementations, this modularization offers two distinct advantages. First, the provenance provider can be reused across the entire range of provenance-aware analyses introduced in Section 2.1.4. Second, specific analyses can easily switch between different provenance providers, enhancing customization experiences. For instance, symbolic execution may benefit from complete provenance tracking, while taint analysis might prefer sound provenance tracking. Our approach is based on three main propositions:

- In Sections 5.1, 5.2, and 5.3, we propose an approach for tracking the provenance of primitive values based on value promotion, which is independent of consuming analyses and has received less attention than shadow execution.
- In Section 5.4, we propose a novel approach for building provenance-based dynamic analysis that achieves a radical separation of concerns by layering advice to separately handle provenance and analysis-specific concerns.
- In Section 5.5, we propose a novel metric to quantitatively evaluate the precision of tracking provenance, which is independent of consuming dynamic program analyses. Although this method is preliminary and subject to revision, we believe it is a core component of the approach, as it enables the research community to focus on improving the precision of tracking provenance independently of any consuming analysis.

Related work

This chapter adapts and combines well-known techniques from the field of reflective programming to provide an interface that shields analysis implementers from the complexity of managing internal and external values. Next, we briefly review work related to these reflective techniques.

Value virtualization Some research has focused on virtualizing values in JavaScript, which could serve as a foundation for shadow execution [16, 5, 59, 121]. However, these approaches necessitate executing JavaScript code in a modified virtual machine which was explicitly ruled out in Chapter 2. While ECMAScript provides a standard virtualization API called `Proxy` [123, 124], this API is limited

to object values and cannot virtualize primitive values. Despite expressed interest[5], it is unlikely that the `Proxy` API will be extended to include primitive values due to performance concerns and semantic guarantees.

Interposition layer The idea of relying on virtual objects to seamlessly mediate the exchange of data between two distinct object graphs is not novel. In the security literature, this approach is known as the *membrane* pattern [54, 79]. This pattern has been specifically proposed for the `Proxy` API [123, 124] under the more general, security-agnostic term *interposition*.

Alternative layering architectures Organizing concerns into levels is natural and has been explored in the past. First are reflective towers [113, 74, 51], which are implemented by stacking meta-circular interpreters. This architecture prevents frontier violations by design but precludes selective instrumentation. More closely related to our advice layering are execution levels [118, 119] from the field of aspect-oriented programming. They aim to prevent infinite recursion by assigning levels to aspects before combining them. In contrast to our approach which is static, execution levels allow dynamically switching between levels at run-time. Finally, several approaches for combining program instrumentation have been proposed [129, 126, 45], but they primarily focus on ensuring non-interference between analyses.

5.6.1 Directions for future research

Exploring unsound provenance tracking In Section 5.1.4, we made the design decision to consistently promote values of unknown origin into fresh handles. This results in an approximation of provenancial equality that is sound but incomplete. In the future, it will be valuable to investigate how different promotion heuristics affect the precision of the provenancial equality derived from them.

Static detection of frontier violations In Section 5.2.2, we formally defined the concept of a frontier as a family of value subsets and a partition of state locations. In the future, it will be worthwhile to further explore this formalism by comprehensively defining a frontier in these terms and investigating how to formally prove that a given analysis enforces said frontier. Currently, our validation that an analysis enforces a frontier is only empirical and limited to run-time checks.

Further establishing our provenance metric In Section 5.5, we proposed a per-value measure of the precision of a provenance provider based on the amount of information known about the origin of values. Currently, this measure lacks intrinsic meaning beyond its definition. Although this definition has clear implications for consuming analyses, it would be interesting to further elucidate how it relates to the precision of representative analyses.

[5] `https://github.com/hugoattal/tc39-proposal-primitive-proxy`

Chapter 6

Linvail: sound provenance tracking for JavaScript

In this chapter, we present Linvail, a JavaScript library that implements our membrane design from the previous chapter. The goal of this chapter is to validate the proposition that separating provenance concerns from analysis-specific logic is feasible and simplifies analysis implementation. This chapter is organized as follows:

- *Section 6.1 introduces Linvail, highlighting key aspects of its implementation and interface.*
- *Section 6.2 demonstrates Linvail's expressiveness by presenting three provenance-aware analyses built upon it.*
- *Section 6.3 experimentally validates Linvail's semantic transparency by applying it to the Test262 test suite.*
- *Section 6.4 evaluates Linvail's performance overhead by applying it to the Octane benchmark.*
- *Section 6.5 assesses Linvail's precision by applying it to Aran itself.*
- *Section 6.6 concludes the chapter by summarizing results and discussing related work.*

6.1 Linvail: implementation of our approach to track provenance

In this section, we detail Linvail, an implementation of our membrane frontier design for JavaScript. This section is organized as follows:

- *Section 6.1.1 details how Linvail adapts our membrane frontier design for JavaScript.*
- *Section 6.1.2 presents Linvail's API through three use cases.*
- *Section 6.1.3 presents Linvail's oracle, which is essential for maintaining precision through calls to known external functions.*
- *Section 6.1.4 addresses the primary limitation of Linvail: its inability to track the provenance of values within exotic objects.*
- *Section 6.1.5 highlights an interesting technical challenge related to the invariants of operations from the meta-object protocol.*

Linvail is a JavaScript library built on top of Aran, available on GitHub[1] and installable as an npm package[2].

[1] `https://github.com/lachrist/linvail`
[2] `https://www.npmjs.com/package/linvail`

6.1.1 Linvail's membrane

In this section, we present several key TypeScript type definitions that drive Linvail's architecture, providing insights into its implementation.

Figure 6.1 illustrates the application domain of Linvail's membrane. It resembles the language-generic membrane described in Section 5.3.3, except that dirty references are not separated based on whether they are records or functions (as references can be both in JavaScript). Instead, they are categorized based on whether they are array objects, coroutine functions (including asynchronous functions or generators), or other object types (referred to as regular objects). Arrays and coroutines require separate handling because virtual arrays must expose a number for their `length` property, and because Aran is able to intercept the results of coroutines. Linvail's membrane performs the following conversions: 1. virtualization of dirty objects, 2. recovery of dirty objects from handles, 3. promotion and demotion across the internal–external boundary.

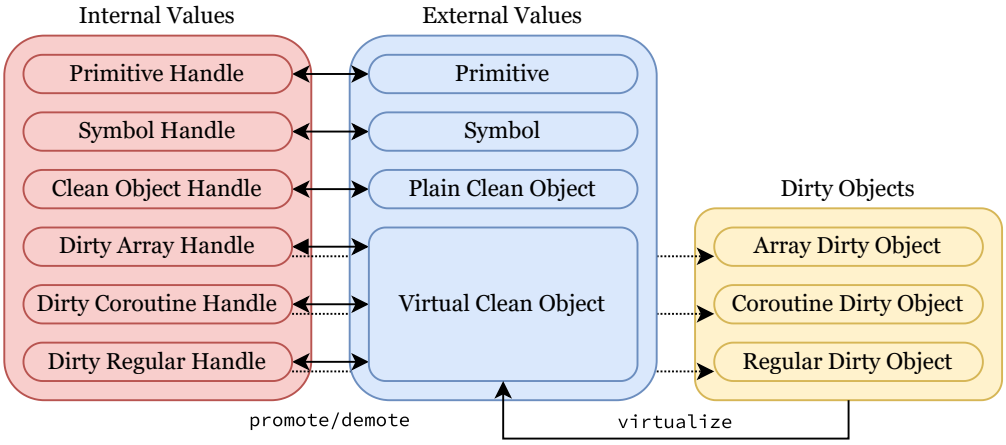


Figure 6.1: Diagram of the application domain of Linvail's membrane

As mentioned in Section 5.4.3, code that manipulates a mix of internal and external values is prone to violating the frontier, which can be difficult to diagnose as it may remain unnoticed for an extended period. Linvail itself consists of such code. To illustrate this difficulty, consider Listing 6.1, which presents an oracle for applying the `%Reflect.getOwnPropertyDescriptor%` intrinsic function. There are six cases to consider: (a) Lines 7–8 throw a type error if the target is not an object. (b) Lines 9–10 promote the descriptor as a plain clean object if the target is a handle to a plain clean object. (c) Lines 12–13 promote the descriptor as a primitive if it is undefined. (d) Lines 14–15 promote the descriptor as a plain clean object if it is an accessor descriptor. (e) Lines 16–17 promote the descriptor as a plain clean object if the target is a handle for a dirty array and the property is `"length"`. (f) Lines 18–22 improve precision by returning a handle to a regular dirty object that has retained the provenance of the descriptor's value.

The non-trivial case analysis in Listing 6.1, required to implement an oracle for `%Reflect.getOwnPropertyDescriptor%` (one of the simplest intrinsic functions), should suffice to demonstrate that preserving frontiers is a delicate and error-prone task. To alleviate some of this burden, we relied on TypeScript to detect many frontier violations at compile time. As mentioned in Section 5.4.3, type systems often struggle with reflective code, and TypeScript is no exception. Nonetheless, we devised a setup that proved valuable for catching violations early. Next, we present the three files that define this setup, as they offer valuable insights into Linvail's internal architecture.

First, Listing 6.2 presents the TypeScript type definitions for the value subsets depicted in Figure 6.1. The type definitions are straightforward; however, the presence of the `__brand` properties in object types requires clarification. This property is not actually present at run time but serves as a marker for the TypeScript type system, indicating that two object types are incompatible. This technique is commonly referred to as a *branded type* and is often employed to provide manual control over type checking while retaining certain static guarantees.

```
import {
  promoteExternalValue,
  promoteDirtyRegularObject,
  promotePlainCleanObject,
} from "./frontier.mjs";
export const getOwnPropertyDescriptorOracle = (target, key) => {
  if (target.type === "PrimitiveHandle" || target.type === "SymbolHandle")
    throw new TypeError("Cannot get property descriptor of primitive");
  if (target.type === "CleanObjectHandle")
    return promotePlainCleanObject(Reflect.getOwnPropertyDescriptor(target.inner, key.inner));
  const descriptor = Reflect.getOwnPropertyDescriptor(target.dirty, key.inner);
  if (descriptor === undefined)
    return promoteExternalValue(descriptor);
  if (!Object.hasOwn(descriptor, "value"))
    return promotePlainCleanObject(descriptor);
  if (target.type === "ArrayDirtyObjectHandle" && key.inner === "length")
    return promoteExternalValue(descriptor);
  return promoteDirtyRegularObject({
    value: descriptor.value,
    writable: promoteExternalValue(descriptor.writable),
    configurable: promoteExternalValue(descriptor.configurable),
    enumerable: promoteExternalValue(descriptor.enumerable),
  });
};
```

Listing 6.1: Oracle for applying `%Reflect.getOwnPropertyDescriptor%`

```
// Primitive and Clean Objects (External Values) //
type Primitive = null | undefined | boolean | number | bigint | string;
type PlainCleanObject =   { __brand: "PlainCleanObject" };
type VirtualCleanObject = { __brand: "VirtualCleanObject" };
type CleanObject = PlainCleanObject | VirtualCleanObject;
type ExternalValue = Primitive | symbol | CleanObject;
// Dirty Objects (Bridge-Only Values) //
type ArrayDirtyObject =     { __brand: "ArrayDirtyObject" };
type CoroutineDirtyObject = { __brand: "CoroutineDirtyObject" };
type RegularDirtyObject =   { __brand: "RegularDirtyObject" };
type DirtyObject = ArrayDirtyObject | CoroutineDirtyObject | RegularDirtyObject;
// Handles (Internal Values) //
type GenericHandle<S, EV, DO> = { __brand: S, type: S, inner: EV, dirty: DO };
type Handle =
  | GenericHandle<"PrimitiveHandle",      Primitive,          null>
  | GenericHandle<"SymbolHandle",         symbol,             null>
  | GenericHandle<"CleanObjectHandle",    PlainCleanObject,   null>
  | GenericHandle<"DirtyArrayHandle",     VirtualCleanObject, ArrayDirtyObject>
  | GenericHandle<"DirtyCoroutineHandle", VirtualCleanObject, CoroutineDirtyObject>
  | GenericHandle<"DirtyRegularHandle",   VirtualCleanObject, RegularDirtyObject>
type InternalValue = Handle;
```

Listing 6.2: Type definitions for the application domain of Linvail's membrane

Second, Listing 6.3 defines helper types for property descriptors, which are regular objects containing information about a property. The `DefineDescriptor` type corresponds to descriptors used to mutate an object's property, whereas the `Descriptor` type corresponds to the descriptor returned when querying an object's property. The use of these types in Listing 6.4 indicates that not all information contained in an object can be tracked. For example, Linvail does not provide a mechanism to track the boolean flag indicating whether a property is configurable. Although it would be possible to overcome this limitation by implementing the entire meta-object protocol (MOP) for dirty objects, doing so would incur significant performance costs. It is unlikely that the resulting overhead would be justified by the gain in precision for typical use cases in provenance-aware analysis.

```
1  import type { CleanObject, ExternalValue } from "./domain";
2  type DefineDescriptor<D> = {
3    __proto__?: null;
4    value?: D;
5    writable?: ExternalValue;
6    get?: ExternalValue;
7    set?: ExternalValue;
8    configurable?: ExternalValue;
9    enumerable?: ExternalValue;
10 };
11 type DataDescriptor<D> = {
12   value: D;
13   configurable: boolean;
14   writable: boolean;
15   enumerable: boolean;
16 };
17 type AccessorDescriptor = {
18   get: CleanObject | undefined;
19   set: CleanObject | undefined;
20   configurable: boolean;
21   enumerable: boolean;
22 };
23 type Descriptor<D> = DataDescriptor<D> | AccessorDescriptor;
24 type NotLength = { __brand: "NotLength" };
```

Listing 6.3: The types of property descriptors used by Linvail to access objects

Third, and most interestingly, Listing 6.4 redefines the type of the `%Reflect%` intrinsic object. This member of the global object contains functions that implement the fundamental operations of the MOP. The goal of this type definition is to provide Linvail with a mechanism for manipulating clean and dirty objects while detecting frontier violations at compile time. This is achieved by providing overloaded type definitions for each function, which flag code locations where Linvail fails to perform the required case analysis. Below, we describe the type definitions for the functions of `%Reflect%`:

- `apply` (Lines 14–19): This function reflectively invokes a function (first argument) with a `this` argument (second argument) and an array of arguments (third argument). (a) If the callee is a clean object, both the input and output of the call should be external values. (b) If the callee is a dirty coroutine, the input should be internal values, while the output will be an external value (either a promise or an iterator). (c) If the callee is a dirty array, the type of the input does not matter, as the reflective call will directly throw a type error. (d) If the callee is any other dirty object, both the input and output of the call should be internal values.
- `construct` (Lines 20–23): This function reflectively calls a function (first argument) as a constructor with an array of arguments (second argument) and a `new.target` argument (third argument). If the callee is a clean (resp. dirty) object, the inputs should be external (resp. internal) values and the output will be an external (resp. internal) value. In both cases, the `new.target` argument is an external value; this does not impact precision because this argument should always be an object. Note that this overload is simpler than that of `apply` because coroutines will throw a type error when invoked as constructor.
- `getOwnPropertyDescriptor` (Lines 25–31) and `defineProperty` (Lines 32–38): These functions respectively query and mutate a single property of a given object. The types clarify that plain clean (resp. dirty) objects contain external (resp. internal) property descriptors. A special case applies to dirty arrays, as the `"length"` property is managed by the runtime and can only hold numeric values.
- `getPrototypeOf` (Line 40) and `setPrototypeOf` (Line 41): These functions respectively query and mutate the prototype of a given object. The types clarify that the prototype of both plain clean and

dirty objects is either `null` or a clean object. While we could have allowed internal values (handles) as prototypes of dirty objects, we decided against this as it enables `%Reflect.has%` to be applied to dirty objects. This performance gain comes at the cost of being unable to track the provenance of the `null` prototype, which is unlikely to be significant in typical use cases.

- `has` (Line 43): This function checks for the presence of a property in a given object and its prototype chain. It can be directly applied to dirty objects because prototypes are guaranteed not to be handles.
- `get` (Line 44) and `set` (Line 45): These functions respectively look up and assign a value to a property of a given object. Unlike the `has` operation, these functions cannot be directly applied to dirty objects because they access the contents of properties in the prototype chain. This limitation is indicated by the absence of an overload case for dirty objects. Instead, these operations are re-implemented in Linvail to perform the necessary case analysis.
- Remainder: The other functions of `%Reflect%` do not have specific semantics within Linvail's membrane.

```
1  import type { DefineDescriptor, Descriptor, NotLength } from "./descriptor";
2  import type {
3    InternalValue as I,
4    ExternalValue as E,
5    CleanObject,
6    DirtyObject,
7    ArrayDirtyObject,
8    CoroutineDirtyObject,
9    RegularDirtyObject,
10 } from "./domain";
11
12 type Reflect = {
13   // Function //
14   apply: {
15     (f:          CleanObject, t: E, xs: E[]): E;
16     (f: CoroutineDirtyObject, t: I, xs: I[]): E;
17     (f:     ArrayDirtyObject, t: I, xs: I[]): never;
18     (f:   RegularDirtyObject, t: I, xs: I[]): I;
19   };
20   construct: {
21     (f: CleanObject, xs: E[], t: E): E;
22     (f: DirtyObject, xs: I[], t: E): I;
23   };
24   // Own Property //
25   getOwnPropertyDescriptor: {
26     (o:          CleanObject, k: E):         Descriptor<E> | undefined;
27     (o:     ArrayDirtyObject, k: "length"):  Descriptor<E>;
28     (o:     ArrayDirtyObject, k: NotLength): Descriptor<I> | undefined;
29     (o: CoroutineDirtyObject, k: E):         Descriptor<I> | undefined;
30     (o:   RegularDirtyObject, k: E):         Descriptor<I> | undefined;
31   };
32   defineProperty: {
33     (o:          CleanObject, k: E,         d: DefineDescriptor<E>): boolean;
34     (o:     ArrayDirtyObject, k: "length",  d: DefineDescriptor<E>): boolean;
35     (o:     ArrayDirtyObject, k: NotLength, d: DefineDescriptor<I>): boolean;
36     (o: CoroutineDirtyObject, k: E,         d: DefineDescriptor<I>): boolean;
37     (o:   RegularDirtyObject, k: E,         d: DefineDescriptor<I>): boolean;
38   };
39   // Prototype Access //
40   getPrototypeOf: (o: CleanObject | DirtyObject) => null | CleanObject;
41   setPrototypeOf: (o: CleanObject | DirtyObject, p: null | CleanObject) => boolean;
42   // Prototype Lookup //
43   has: (o: CleanObject | DirtyObject, k: E) => boolean;
44   get: (o: CleanObject, k: E, r: E) => E;
45   set: (o: CleanObject, k: E, v: E, r: E) => boolean;
46   // Others //
47   preventExtensions: (o: CleanObject | DirtyObject) => boolean;
48   isExtensible: (o: CleanObject | DirtyObject) => boolean;
49   ownKeys: (o: CleanObject | DirtyObject) => PropertyKey[];
50   deleteProperty: (o: CleanObject | DirtyObject, k: E) => boolean;
51 };
```

Listing 6.4: Redefinition of the types of `%Reflect%` functions for use in Linvail

6.1.2 Linvail's usage

In this section, we illustrate Linvail's API through three representative use cases: manual interaction with its membrane, building analyses via advice extension, and inline use of its provenance-sensitive library.

The most straightforward way to use Linvail is to manually interact with its membrane. The entry point for this usage is `linvail/runtime`, which exports the following functions:

- `createRegion`: Instantiates the state necessary to maintain Linvail's membrane. While this state is referred to as `region` in the code, it is only loosely related to our formal definition of a region from Section 5.2.2.
- `createMembrane`: Instantiates several functions for accessing the specified region state.
- `membrane.promoteExternalValue`: Promotes an external value (whether primitive, symbol, or clean object) into an internal value (handle). Clean objects that have not been previously registered are considered to be plain clean objects.
- `membrane.promoteDirtyObject`: Promotes a dirty object (whether an array, coroutine, or regular object) into an internal value (handle). This operation involves instantiating and registering a virtual clean object.
- `membrane.apply`: Reflectively invokes a handle as a function (first argument), with a `this` argument handle (second argument), and an array of argument handles (third argument); the result is guaranteed to also be a handle.
- `membrane.construct`: Reflectively invokes a handle as a constructor (first argument), with an array of argument handles (second argument), and a `new.target` argument handle; the result is guaranteed to also be a handle.

Listing 6.5 demonstrates the manual usage of Linvail. Line 2 instantiates a new region state. Line 3 initializes the membrane functions to access the region state. Lines 4–6 promote three numbers into primitive handles. Line 7 promotes the intrinsic `%Array.prototype.push%` into a clean object handle. Line 8 promotes a dirty array into a dirty array handle. Line 9 internally applies `%Array.prototype.push%`, for which Linvail has oracle knowledge. This results in an array containing three handles for `789`, `456`, and `123`. Line 10 externally applies `Array.prototype.sort`, which sorts the elements of an array in place. Line 11 demonstrates that this last call was executed as if the array contained numbers (non-handles), which would have been required in an analysis context to preserve transparency.

```
1  import { createRegion, createMembrane } from "linvail/runtime";
2  const region = createRegion(globalThis);
3  const { promoteExternalValue, promoteDirtyObject, apply } = createMembrane(region);
4  const $123 = promoteExternalValue(123);
5  const $456 = promoteExternalValue(456);
6  const $789 = promoteExternalValue(789);
7  const $push = promoteExternalValue(Array.prototype.push);
8  const $array = promoteDirtyObject("array", [$789, $456]);
9  apply($push, $array, [$123]);
10 $array.inner.sort();
11 console.log(JSON.stringify($array.dirty));
```

↓↓↓

```
1 [ "type":"primitive","inner":123, dirty: null,
2   "type":"primitive","inner":456, dirty: null,
3   "type":"primitive","inner":789, dirty: null ]
```

Listing 6.5: Manual interaction with Linvail's membrane

To perform provenance-aware dynamic analysis, Linvail's membrane must be enforced on the target program. This involves creating a Linvail-specific advice from a region state and then either directly extending it with the analysis logic (cf. Section 5.4.2) or combining it with the analysis advice (cf. Section 5.4.3). Section 6.2 presents several analysis examples for both architectures. Figure 6.2 illustrates the architecture for deploying Linvail analyses through offline instrumentation and advice extension (color scheme unrelated to regions). The analysis is conducted in two sequential phases:

1. Offline Instrumentation: Instrumentation follows the standard Aran pipeline depicted in Figure 4.2: (a) parse JavaScript code into ESTree AST nodes, (b) transpile these nodes into AranLang AST nodes, (c) weave these nodes with calls to advice functions, (d) retropile the woven AranLang nodes back into ESTree nodes, (e) generate JavaScript code. The main point of interest is that the pointcut is provided as static JSON data from Linvail, which may need to be adapted if the analysis implements advice functions that are not required by Linvail to enforce its membrane.

2. Online Analysis: Setting up the analysis requires the following steps: (a) instruct Linvail to instantiate a region state, (b) instruct Linvail to instantiate a standard Aran advice for generically enforcing the frontier, (c) instruct Linvail to initialize membrane access functions, (d) leverage these functions to extend the Linvail-generated advice with analysis-specific logic, (e) make the extended advice globally available, (f) instantiate the intrinsic registry and also make it globally available. After setup, the analysis proceeds by executing the instrumented version of the target program.

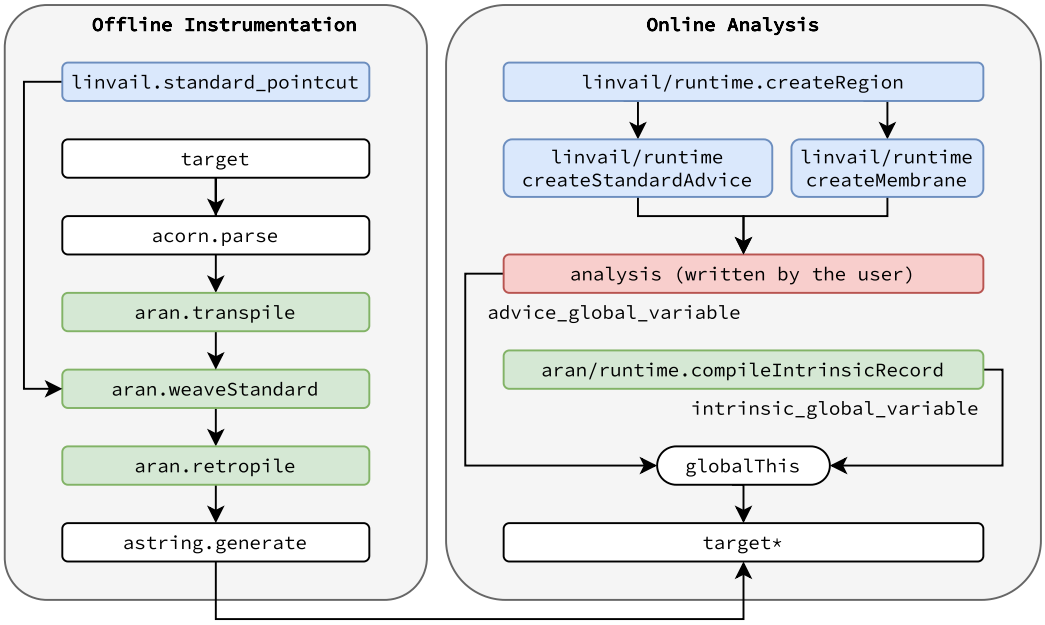


Figure 6.2: Deployment of Linvail for provenance-aware analysis via advice extension

Finally, Linvail can also serve as a provider of provenance-sensitive functions, as outlined in Section 5.4.1. While these functions can be imported into arbitrary code (external region), gains in precision require instrumentation (internal region), which can be accomplished using Linvail's CLI, for instance. The following provenance-sensitive functionalities are available at the `linvail/library` entry point:

- `is`: Referentially compares two handles to implement provenancial equality implemented as referential equality of handles.
- `dir`: Bypasses the membrane to reveal the actual content of a handle, which is useful for debugging.
- `resetProvenance`: Returns the argument with a fresh provenance. This function is simply the external identity function.
- Provenance-Sensitive Collections: Similar to ES6 collections (i.e., `Set`, `WeakSet`, `Map`, and `WeakMap`), except that keys are differentiated based on their provenance and that weak collections (appear to) support primitive keys.

6.1.3 Linvail's oracle

In this section, we present Linvail's oracle, which is essential for maintaining precision when applying external functions with known semantics.

As discussed in Section 3.4, Aran normalizes many syntactic constructs into calls to reflective functions; for instance, array literals may be replaced with calls to `%Array.of%`, and member expressions may be replaced with calls to `%Reflect.get%`. While this normalization aids in instrumentation, it also undermines precision if the callees of these inserted call expressions are treated as arbitrary external functions.

In Section 5.3.4, we discussed how ad-hoc handling of applications to key intrinsic functions plays an important role in precisely tracking provenance. In particular, the pervasive calls to reflective functions require knowledge of their semantics to leverage any precision gains associated with allowing handles in the store. In Linvail, the ad-hoc handling of intrinsic functions is referred to as its *oracle*. Thus, when we say that an intrinsic function is present in Linvail's oracle, we mean that its application is handled by ad-hoc logic rather than by the generic mechanism used for applying arbitrary plain external functions.

Table 6.1 provides an overview of the scope of Linvail's oracle. In addition to the intrinsic functions specific to Aran and Linvail, the intrinsic functions most relevant to the oracle belong to the global objects `%Reflect%`, `%Object%`, and `%Array%`. In ECMAScript 2025 and in the current version of the Aran Linvail stack, these functions account for a total of 126 elements, of which:

- 81 are implemented in the oracle, such as: (a) `%aran.toArgumentList%` which transfers the provenance of the argument values into the new `arguments` object, (b) `%linvail.is%` which preserves the provenance of the operands of provenancial equality, (c) `%Reflect.get%` which returns the provenance of the looked-up property value, (d) `%Object.assign%` which transfers the provenance of the property values of the source object to those of the target object, (e) `%Array.of%` which transfers the provenance of the arguments into the new array, (f) `%Array.prototype.map%` which transfers the provenance of each array element to the input of the mapping function and then transfers the provenance of the function's output into the new array.
- 41 are omitted from the oracle, as implementing them would not yield additional precision. This mainly applies to pure functions that return a primitive value with a fresh provenance, such as `%aran.isConstructor%`, `%linvail.ProvMap.prototype.getSize%`, `%Object.isExtensible%`, and `%Array.prototype.toString%`.
- 4 are excluded from the oracle despite their potential for precision gains. These apply to array-related functions that return a built-in iterator, such as `%Array.prototype.values%`. Section 6.1.4 discusses the limitation that prevents Linvail from turning certain exotic objects, including built-in iterators, into dirty objects.

Namespace	Total	Oracle	No Benefit	Not Poss.
Aran	19	6	13	0
Linvail	26	22	4	0
`%Reflect.*%`	13	7	6	0
`%Object%`	1	1	0	0
`%Object.*%`	23	13	10	0
`%Array%`	1	1	0	0
`%Array.*%`	3	2	1	0
`%Array.prototype.*%`	40	29	7	4
	126	81	41	4

Table 6.1: Account of the functions recognized by Linvail's oracle

6.1.4 Challenges of maintaining provenance through exotic objects

In this section, we examine the reasons why Linvail, in its current implementation, is unable to track the provenance of values through exotic objects.

The ECMAScript specification defines exotic objects as those whose behavior in the meta-object protocol diverges from that of ordinary objects. For instance, arrays are considered exotic objects because they possess a `length` property that updates automatically, which is not the case for ordinary objects. However, in this section, we adopt a slightly broader definition of exotic objects by also including all objects that possess *internal slots*. These internal slots can be viewed as hidden properties that can only be accessed by intrinsic functions.

For example, promise objects contain two internal slots accessed by promise methods such as `%Promise.prototype.then%`. The first slot, `[[PromiseState]]`, stores the current state of the promise, which can be either `pending`, `fulfilled`, or `rejected`. The second slot, `[[PromiseResult]]`, stores the resolution value or the rejection reason, which can be any JavaScript value. Although promise objects do not override the default behavior of the meta-object protocol, we still classify them as exotic objects due to their internal slots, which lead to behavior that differs from ordinary objects in certain intrinsic functions.

Some internal slots, such as `[[PromiseState]]` in promises, cannot hold arbitrary JavaScript values. Consequently, tracking the provenance of data stored in these slots is not feasible within our value-centric approach. In contrast, other slots, such as `[[PromiseResult]]` in promises, can hold arbitrary values, which a priori might seem to enable provenance tracking through handles. Figure 6.3 illustrates the object graph that would be required to track the provenance of both the result value (`123`) and the `"foo"` property value (`456`) within a fulfilled dirty promise (`promise`).

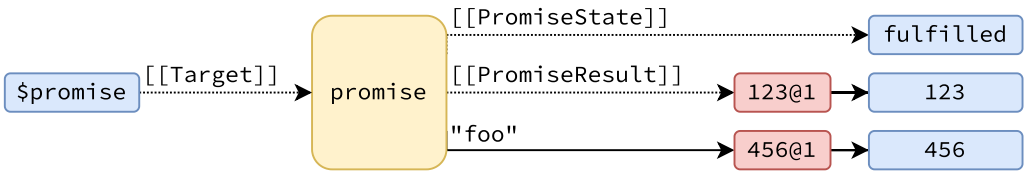


Figure 6.3: Fabricated object graph for tracking the provenance of resolution value in promises

Unfortunately, as demonstrated in Listing 6.6, the virtualization API for JavaScript (`Proxy`) cannot fully virtualize promises. This limitation means that it is not possible to instantiate the `$promise` box, which would denote a virtual clean object for the dirty promise. Since this restriction also applies to other exotic object types, Linvail cannot track the provenance of primitive values embedded within most exotic objects, whether they reside in internal slots or ordinary properties. While there have been ECMAScript proposals, such as the one by Mark S. Miller,[3] to extend the capabilities of the `Proxy` virtualization API to certain internal slots, they have not been adopted, and the `Proxy` API is unlikely to change in the near future.

```
1 (new Proxy(Promise.resolve(123), {})).then((_value) => {}); // TypeError
```

Listing 6.6: The `Proxy` API does not fully virtualize promises

To qualitatively assess Linvail's precision, we outline how Linvail manages several key exotic object types:

- Arrays: Can be instantiated by array literal expressions or intrinsic functions (e.g., `%Array.of%` and `%Array.prototype.map%`). Arrays can be fully virtualized, making them suitable candidates for registration as (array) dirty objects when all their non-length property values have been assigned to handles.
- Functions: Can be instantiated via function literal expressions (closures) or the `%Function%` intrinsic constructor (dynamically evaluated). They can be fully virtualized, allowing closures instantiated

[3] `https://github.com/tc39/proposal-eventual-send`

by instrumented code to be registered as dirty objects (either regular or coroutine) when all their property values have been assigned to handles.

- Proxies: Can be instantiated via the `%Proxy%` constructor. Proxies can be fully virtualized; however, they cannot directly contain primitive handles that have no properties and only object-only internal slots (i.e., `[[Target]]` and `[[handler]]`). Nonetheless, special handling of proxies could enhance precision by preserving the provenance of primitive values as they transition from the arguments of a reflective function to the arguments of the corresponding trap handler. Linvail does not currently implement this intriguing provenance tracking, which is left as future work.

- Promises: Can be instantiated by calling asynchronous functions, the `%Promise%` constructor, or other promise-related intrinsic functions such as `Promise.prototype.then`. As discussed earlier, promises cannot be fully virtualized, meaning that neither their property values nor their resolved value should be handles. This is unfortunate, as it hinders the tracking of provenance for the resolved results of asynchronous functions.

- ES6 Collections: Can be instantiated by invoking ES6 collection constructors, such as `%WeakMap%`. The data for these collections is stored in a specific internal slot, such as `[[WeakMapData]]` for weak maps. Similar to promises, they cannot be fully virtualized, meaning that neither their ordinary property values, nor their keys, nor their values should be handles. This is also unfortunate because tracking the provenance of primitives through ES6 collections would likely lead to significant gains in precision.

- Array Buffers: Can be instantiated via the `%ArrayBuffer%` constructor and were introduced to JavaScript to enhance performance by providing a low-level yet safe API for memory access. The data of array buffers is stored as bytes in the `[[ArrayBufferData]]` internal slot. Since these are raw bytes, tracking them is not feasible within our value-centric approach. Furthermore, similar to promises, array buffers cannot be virtualized, which means their property values should not be handles either.

- Boxed Primitives: Each primitive, except for `null` and `undefined`, can be boxed into an exotic object that contains the primitive data within an internal slot. Similar to promises, boxed primitives cannot be virtualized; thus, neither their inner primitive value nor their property values can be handles. However, since their inner slot is immutable, it would be straightforward to implement a registry linking the identity of a boxed primitive to the provenance of its inner value. Linvail does not currently implement this provenance tracking, which is left as future work.

- Builtin Iterators: While the iterator protocol can be implemented through ordinary objects that support virtualization, the ECMAScript specification defines several exotic objects specifically for built-in iteration, such as those used for iterating over array items or the results yielded by generators. This limitation prevents Linvail from tracking the provenance of the results produced by generators, which is unfortunate.

- Others: Many other exotic objects, such as DOM elements and those created by the `%RegExp%` and `%Date%` constructors cannot be virtualized. Hence neither their internal slot values nor their property values should be handles.

To address this limitation, we explored a method that combines traditional shadow execution with our approach to shadow values [101]. The core idea is to prioritize value promotion for tracking information through ordinary and array objects while switching to shadow execution when dealing with other exotic objects. As discussed in Section 5.3.1, when applied to managed languages, shadow execution involves maintaining a copy of objects from the base layer, filled with metadata. Although this approach achieves clean layer separation, it is subject to synchronization issues. This situation necessitates either compromising the soundness of the analysis or implementing imprecise conservative updates.

6.1.5 Decoupling proxy invariant enforcement from target objects

> *In this section, we address an interesting technical challenge encountered during the implementation of Linvail related to the way the `Proxy` API uses the target object as a bookkeeping mechanism to enforce invariants.*

In object-oriented languages, both plain and virtual objects are expected to uphold invariants. For example, when a field of an object is declared constant, it should remain unchanged throughout the object's lifetime. To detect invariant violations and raise a type error, instances of the `Proxy` API perform checks before returning from certain operations of the meta-object protocol (MOP). Unfortunately, the manner in which these invariant checks are implemented introduces issues in our problem domain.

Consider Listing 6.7 which presents a skeleton implementation of Linvail's membrane. Lines 1–2 define promotion and demotion operations. Lines 3–10 define a handler object that inserts these operations at the appropriate places to ensure virtual clean objects behave like plain clean objects. The virtualization is incomplete, as the handler object only intercepts the `getOwnPropertyDescriptor` operation of MOP. The responsibility of this interception is to demote (Line 7) the value of the descriptor originating from the target object, should it exist.

skeleton.mjs

```
1  export const promote = (inner) => ({ inner });
2  export const demote = ({ inner }) => inner;
3  export const handler = {
4    getOwnPropertyDescriptor: (target, key) => {
5      const descriptor = Reflect.getOwnPropertyDescriptor(target, key);
6      if (descriptor && "value" in descriptor)
7        descriptor.value = demote(descriptor.value);
8      return descriptor;
9    },
10 };
```

Listing 6.7: Skeleton implementation of Linvail's membrane

Listing 6.8 demonstrates that directly applying the skeleton implementation of Linvail's membrane to the `Proxy` API would compromise transparency in an analysis context. Line 2 instantiates an object that should be considered as dirty. Lines 3–5 assign a constant property descriptor (i.e., non-writable and non-configurable) to the dirty object at `foo`. Line 6 virtualizes the dirty object into a virtual clean object. Line 7 attempts to extract the property descriptor at `"foo"`, but an exception is thrown instead of a constant property descriptor being returned, which is not transparent.

```
1 import { promote, handler } from "./skeleton.mjs";
2 const dirty_object = {};
3 Reflect.defineProperty(dirty_object, "foo", {
4   value: promote(123), writable: false, configurable: false,
5 });
6 const virtual_clean_object = new Proxy(dirty_object, handler);
7 Reflect.getOwnPropertyDescriptor(virtual_clean_object, "foo"); // ❌ TypeError
```

Listing 6.8: Non-transparent proxy behavior caused by invariant enforcement

Figure 6.4 presents a sequence diagram that illustrates why an exception is thrown in Listing 6.8. When the intrinsic `%Reflect.getOwnPropertyDescriptor%` is called on `virtual_clean_object`, which is a proxy, it triggers the corresponding trap in the handler object. In this setup, the trap forwards the request to the target object, `dirty_object`. The resulting descriptor is sanitized to eliminate any handles before being returned to the proxy. Upon receiving the descriptor, the proxy determines that it describes a constant property; thus, it checks with the target for descriptor compatibility, bypassing the handler object. Since the `value` property of both descriptors does not match, an exception is thrown, whereas we would have preferred the proxy to return the initial descriptor.

The key idea behind detecting invariant violations in proxy objects is to rely on a separate object as a bookkeeping mechanism, which is sound because every object must uphold its invariants [124]. However, Van Cutsem et al. choose to use the same object (referred to as the proxy's target) both for invariant bookkeeping and as the first argument passed to handler functions. While economical, this design is

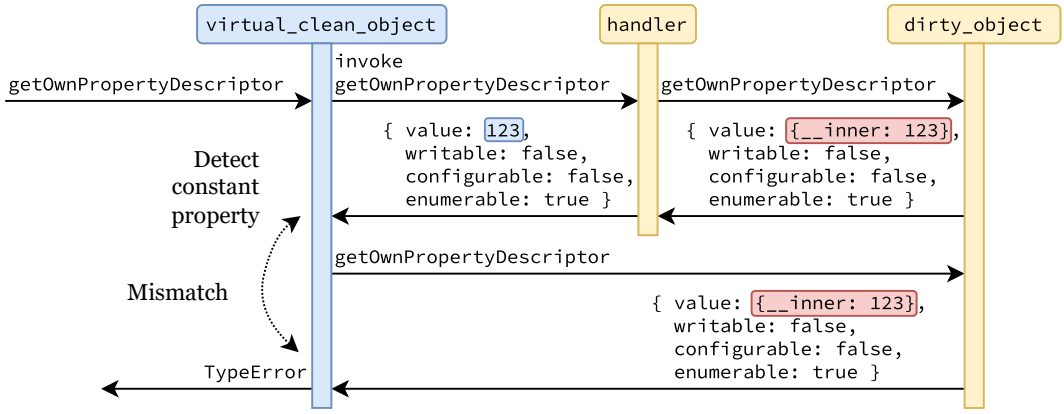


Figure 6.4: Sequence diagram illustrating why an exception was thrown in Listing 6.8

unnecessarily restrictive. In our case, for example, the consistent demotion of descriptor values would have prevented invariant violations.

Still in [124], Van Cutsem et al. describe how to circumvent this limitation by providing an empty dummy object as the actual target of the proxy and making the handler object hold a reference to the logical target. We implemented this strategy in a library called `virtual-proxy`, which is available on GitHub[4] and published on npm[5]. The primary objective of our library is to separate the concerns of bookkeeping information for detecting invariant violations from the parameterization of functions in the handler object. Our library exports `VirtualProxy`, a constructor similar to `%Proxy%`, but it requires three arguments instead of two:

- Integrity Target Object: This object serves as the bookkeeping mechanism used to enforce invariants. It is the actual target object of the created proxy.
- Relaxed Target Value: This value is passed as the first argument to the handler's trap functions. It may be any value, including primitives, and is not used by the `Proxy` API for invariant enforcement.
- Relaxed Handler Object: This object contains the trap functions used to override the operations of the MOP, exactly as in the standard `Proxy` API.

Listing 6.9 illustrates how our `VirtualProxy` API can act as a drop-in replacement for the standard `Proxy` API, requiring only the addition of an empty object to serve as the integrity target. Similar to Listing 6.8, Line 2 instantiates a dirty object, Lines 4–6 define a constant property for that object, and Lines 7–11 virtualize the dirty object into a virtual clean object. However, when Line 13 requests the descriptor of the `foo` property, a (clean) constant descriptor is returned rather than an exception being thrown.

```
1  import { VirtualProxy } from "virtual-proxy";
2  import { promote, handler } from "./skeleton.mjs";
3  const dirty_object = {};
4  Reflect.defineProperty(dirty_object, "foo", {
5    value: promote(123), writable: false, configurable: false,
6  });
7  const virtual_clean_object = new VirtualProxy(
8    {},           // Integrity target object (for invariant bookkeeping)
9    dirty_object, // Parameter target value (for argument to handler functions)
10   handler,      // Handler object
11 );
12 // ✅ { value: 123, writable: false, enumerable: false, configurable: false }
13 Reflect.getOwnPropertyDescriptor(virtual_clean_object, "foo");
```

Listing 6.9: Usage of our `VirtualProxy` library

Figure 6.5 illustrates how our library operates behind the scenes to separate concerns between invariant bookkeeping and handler parameterization. When a proxy instantiated by our library receives a MOP request, it first invokes the corresponding function in the integrity handler object (the actual handler

[4] https://github.com/lachrist/virtual-proxy
[5] https://www.npmjs.com/package/virtual-proxy

object of the proxy). This integrity function (implemented within our library) forwards the operation to the corresponding function of the relaxed handler object (provided by the user), with the relaxed target value passed as the first argument. The relaxed handler function is then free to perform arbitrary operations with the relaxed target value before returning to the integrity function. The integrity function may also perform synchronization with the integrity target object (the actual target object of the proxy) before returning. Finally, the proxy instance may check for invariant violations by querying the integrity target object, which would have been synchronized beforehand by our library.

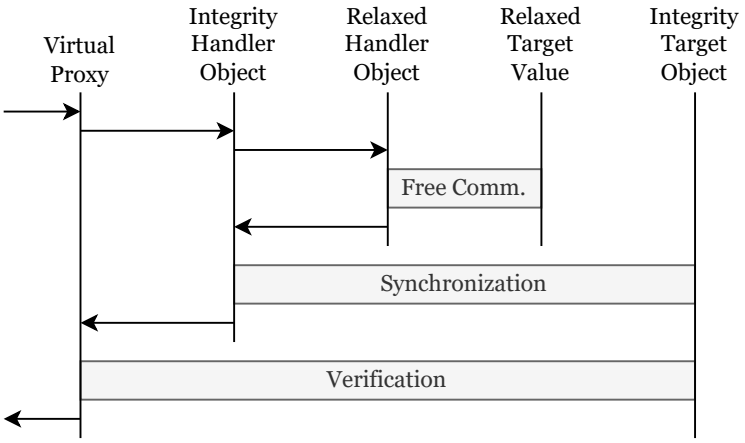


Figure 6.5: Sequence diagram illustrating the inner workings of our `virtual-proxy` library

6.2 Building provenance-aware analyses in Linvail

In this section, we qualitatively evaluate the expressiveness of Linvail by presenting several provenance-aware analyses developed on the Aran Linvail stack. This section is organized as follows:

- *Section 6.2.1 presents an analysis for dynamic symbolic execution based on advice extension.*
- *Section 6.2.2 presents an implementation of virtual values based on advice layering.*
- *Section 6.2.3 presents a dynamic taint analysis based on advice layering and Linvail's notification system.*

6.2.1 Symbolic execution via advice extension

In this section, we demonstrate the Aran Linvail stack by presenting an analysis that employs advice extension to conduct dynamic symbolic execution. The goal is to show that we have effectively addressed our motivating example from Section 2.3.

Dynamic symbolic execution [44, 106] is a prominent dynamic program analysis technique primarily used to automatically generate test inputs that maximize code coverage. The process begins by gathering slices of the value flow graph corresponding to run-time values that influence control flow. This information is then transformed into constraints, which are provided to a solver to generate the next input for the target program. The analysis described in this section focuses on collecting these value flow graph slices.

Figure 6.6 illustrates the module dependency graph of our analysis, which features offline instrumentation. The instrumentation module (`instrument.mjs`) is omitted as it only contains boilerplate code related to Aran. The configuration module (`config.json`) is shared with the online phase of the analysis and provides the global variable names for the intrinsic registry and the analysis advice. Next, we describe the online phase of the analysis.

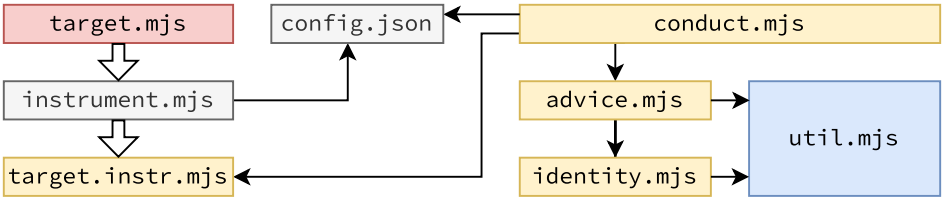


Figure 6.6: Module dependency graph for our symbolic execution

Listing 6.10 presents the entry point of the online phase of the analysis (`conduct.mjs`). Lines 4–5 instantiate the intrinsic registry and assign it to its global variable. Line 6 instantiates the analysis advice and also assigns it to its global variable. Finally, Line 7 imports the instrumented version of the target program.

conduct.mjs

```
1 import { compileIntrinsicRecord } from "aran/runtime";
2 import { createAdvice } from "./advice.mjs";
3 import { global_advice_variable, global_intrinsic_variable } from "./config.json";
4 const intrinsics = compileIntrinsicRecord(globalThis);
5 globalThis[global_intrinsic_variable] = intrinsics;
6 globalThis[global_advice_variable] = createAdvice(intrinsics);;
7 await import("./target.instr.mjs");
```

Listing 6.10: Entry point of the online phase of our symbolic execution

Listing 6.11 presents the utility module of the analysis (`util.mjs`), which is depicted in blue in the dependency module graph, as it should never access (external region). Line 1 exports a logging function from the Node.js runtime. Lines 2–3 define a function for formatting logs that represent function calls. Lines 4–10 define a function for printing a brief description of an arbitrary external value.

util.mjs

```
1 export { log } from "node:console";
2 export const formatCall = ({ callee, input, output, location }) =>
3   `${output} <- ${callee}(${input.join(", ")}) @${location.split("#")[1]}`;
4 export const describe = (value) => {
5   if (typeof value === "function") return value.name ?? "#function";
6   if (typeof value === "object") return value ? "#object" : "null";
7   if (typeof value === "symbol") return value.description ?? "#symbol";
8   if (typeof value === "number") return isNaN(value) ? "NaN" : String(value);
9   return JSON.stringify(value);
10 }
```

Listing 6.11: Utility module for formatting logs during symbolic execution

Listing 6.12 presents a stateful module (`identity.mjs`) for providing a printable identity to handles. Line 2 initializes a global counter. Line 3 initializes a global registry for associating handles with a unique number. Lines 4–10 define an impure function for printing the identity of a handle. If the handle does not exist in the registry, it is added and associated with a unique number obtained by incrementing the global counter.

Listing 6.13 presents the advice module of the analysis (`advice.mjs`), which exports a function for instantiating the analysis advice. Line 10 instantiates a new state region. Line 11 registers the Aran-specific intrinsic functions to Linvail's oracle. Line 12 creates membrane functions to access the state region. Line 14 creates the generic Linvail advice to enforce the frontier. Lines 15–24 overwrite the `apply@around` advice function, which does not require changing the pointcut advertised by Linvail. The added logic simply logs the call in a human-readable format.

To demonstrate that our analysis effectively addresses the motivating example from Section 2.3, we apply it to Listing 6.14, which is a direct adaptation of the Scheme program presented in Listing 2.10.

Listing 6.15 presents an excerpt from the output of the analysis, with user interactions highlighted in blue. As is apparent, these constraints closely resemble those in Listing 2.14, which were generated by logging the identities of values in the naive allocation system. In particular, they can be utilized to reconstruct a value flow graph similar to that in Figure 2.6 and do not exhibit the cycles due to collapsing observed in Figure 2.7. This demonstrates that the Aran Linvail stack effectively reverse-engineered the inlining and interning optimizations implemented by the Node.js engine (i.e., Google's V8).

identity.mjs

```
import { log, describe } from "./util.mjs";
let counter = 0;
const indexing = new WeakMap();
export const identify = (handle) => {
  if (!indexing.has(handle)) {
    indexing.set(handle, ++counter);
    log(`&${counter} := ${describe(handle.inner)}`);
  }
  return `&${indexing.get(handle)}`;
};
```

Listing 6.12: Stateful module for printing the identity of handles

advice.mjs

```
import {
  createMembrane,
  createRegion,
  createStandardAdvice,
  registerAranIntrinsicRecord,
} from "linvail/runtime";
import { identify } from "./identity.mjs";
import { log, describe, formatCall } from "./util.mjs";
export const createAdvice = (intrinsics) => {
  const region = createRegion(globalThis);
  registerAranIntrinsicRecord(region, intrinsics);
  const { apply } = createMembrane(region);
  return {
  ...createStandardAdvice(region),
   "apply@around": (_state, callee, that, input, location) => {
      const output = apply(callee, that, input);
      log(formatCall({
        callee: describe(callee.inner),
        input: input.map(identify),
        output: identify(output),
        location,
      }));
      return output;
    },
  };
};
```

Listing 6.13: Advice for recording symbolic constraints

target.mjs

```
import { question } from "readline-sync";

const solve = ({ a, b, c }) => {
  const delta = Math.pow(b, 2) - 4 * a * c;
  const sol1 = (-b + Math.sqrt(delta)) / (2 * a);
  const sol2 = (-b - Math.sqrt(delta)) / (2 * a);
  return `Sol1 = ${sol1}, Sol2 = ${sol2}`;
};

console.log(solve({
  a: question("a: "),
  b: question("b: "),
  c: question("c: "),
}));
```

↓↓↓

```
a: 2
b: 3
c: 2
Sol1 = NaN, Sol2 = NaN
```

Listing 6.14: Target program which solves second-order polynomial equations

```
a: 2
&30 := "a: "
&31 := "2"
&31 <- (&30)) @9:5
b: 3
&32 := "b: "
&33 := "3"
&33 <- (&32)) @10:5
c: 2
&34 := "c: "
&35 := "2"
&35 <- (&34)) @11:5
&55 := 2
&56 := 9
&56 <- pow(&33, &55)) @3:16
&57 := "*"
&58 := 4
&59 := 8
&59 <- performBinaryOperation(&57, &58, &31)) @3:33
&60 := "*"
&61 := 16
&61 <- performBinaryOperation(&60, &59, &35)) @3:33
&62 := "-"
&63 := -7
&63 <- performBinaryOperation(&62, &56, &61)) @3:16
&64 := "-"
&65 := -3
&65 <- performUnaryOperation(&64, &33)) @4:16
&69 := NaN
&69 <- sqrt(&63)) @4:21
&70 := "+"
&71 := NaN
&71 <- performBinaryOperation(&70, &65, &69)) @4:16
&72 := "*"
&73 := 2
&74 := 4
&74 <- performBinaryOperation(&72, &73, &31)) @4:42
&75 := "/"
&76 := NaN
&76 <- performBinaryOperation(&75, &71, &74)) @4:15
&77 := "-"
&78 := -3
&78 <- performUnaryOperation(&77, &33)) @5:16
&79 := "Math"
&81 := NaN
&81 <- sqrt(&63)) @5:21
&82 := "-"
&83 := NaN
&83 <- performBinaryOperation(&82, &78, &81)) @5:16
&84 := "*"
&85 := 2
&86 := 4
&86 <- performBinaryOperation(&84, &85, &31)) @5:42
&87 := "/"
&88 := NaN
&88 <- performBinaryOperation(&87, &83, &86)) @5:15
&89 := "Sol1 = "
&90 := ", Sol2 = "
&91 := ""
&92 := "Sol1 = NaN, Sol2 = NaN"
&92 <- concat(&89, &76, &90, &88, &91)) @6:9
Sol1 = NaN, Sol2 = NaN
&93 := undefined
&93 <- log(&92)) @8:0
```

Listing 6.15: Excerpt from the trace produced by the symbolic execution analysis

This section demonstrates that our motivating examples require only minimal logic to be resolved, highlighting the expressiveness of the Aran Linvail stack. Most of the logic in our analysis consists of boilerplate code or relates to formatting logs for human readability. The entry point for the inline phase of the analysis (`conduct.mjs`) is boilerplate code. The utility file (`util.mjs`) exports only general-purpose utility functions. Although the identity module (`identity`) is stateful and specific to our technology stack, it is straightforward. Finally, the advice module (`advice.mjs`) merely augments the `apply@around` advice with a logging operation.

6.2.2 Virtual values via advice layering

> *In this section, we further demonstrate the expressiveness of the Aran Linvail stack by implementing virtual values [5] via advice layering.*

As was done in Section 5.4, we could have demonstrated advice layering by adapting our symbolic execution analysis from Section 6.2.1 to layer advice instead. However, we opted to illustrate advice layering within the Aran Linvail stack through a distinct use case (i.e., implementing *virtual values*) as this provides a broader perspective on the capabilities of our approach.

In [5], Austin et al. provide a detailed account of *virtual values*, which can be viewed as an extension of the existing `Proxy` API [123]. While both approaches offer value-centric behavioral intercession, the `Proxy` API focuses on references, whereas virtual values encompass all data types. Due to performance considerations and semantic guarantees, only the more restrictive `Proxy` API has been incorporated into the specification. Nonetheless, virtual values remain a topic of research interest in the field of reflective programming.

To demonstrate the expressiveness of the Aran Linvail stack, we leveraged it to implement virtual values, whose usage is illustrated in Listing 6.16. The program expects three non-standard global functions: `createVirtualValue`, which is similar to the `%Proxy%` constructor but can virtualize any data type; `performUnaryOperation`, which executes unary operations; and `performBinaryOperation`, which executes binary operations. The traps follow the naming convention established in Austin's paper and are defined as follows: (a) `test` which is triggered when the value is used to branch the control flow, (b) `unary` which is triggered when the value is used as the argument of a unary operation, (c) `left` (resp. `right`) which is triggered when the value is used as the left (resp. right) argument of a binary operation.

target.mjs

```
const virtual = createVirtualValue(123, {
  test: (target) => {
    console.log("test", target);
    return target;
  },
  unary: (target, operator) => {
    console.log("unary", operator, target);
    return performUnaryOperation(operator, target);
  },
  left: (target, operator, other) => {
    console.log("left", target, operator, other);
    return performBinaryOperation(operator, target, other);
  },
  right: (target, operator, other) => {
    console.log("right", other, operator, target);
    return performBinaryOperation(operator, other, target);
  },
});
console.log(virtual ? "consequent" : "alternate"); // test 123        >> consequent
console.log(!virtual);                             // unary ! 123     >> false
console.log(virtual + 456);                        // left 123 + 456  >> 579
console.log(789 + virtual);                        // right 789 + 123 >> 912
```

Listing 6.16: Demonstration of virtual values implementation

Figure 6.7 depicts the module dependency graph of our implementation of virtual values, which follows the advice layering architecture from Section 5.4.3. Similar to the dependency graph in Figure 6.6, the original version of the target program is in the internal region. However, the original version of the advice has been moved from the bridge region (a mix of internal and external values) to the internal region. For brevity, we omit `instrument.mjs`.

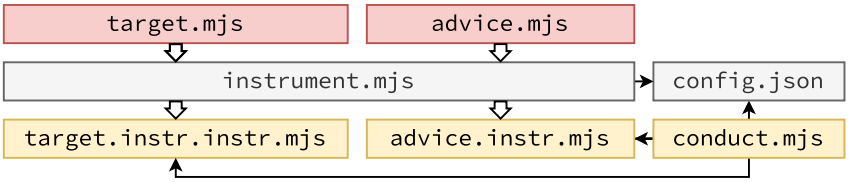


Figure 6.7: Module dependency graph of our implementation of virtual values

Listing 6.17 presents the entry point of the online phase of the analysis (`conduct.mjs`). Lines 10–13 instantiate the intrinsic registry and globally expose it, together with intrinsic functions for performing unary and binary operations. Note that we could have re-implemented these functions here, as they are excluded from Linvail's oracle (their inclusion would yield no benefit since they consistently return primitive values). Lines 14–16 instantiate a region state, register Aran-specific intrinsic functions in its oracle, and make Linvail's library available at the entry point `linvail/runtime`. Line 17 instantiates Linvail's generic frontier advice and globally exposes it. Line 18 executes the instrumented version of the advice and globally exposes its advice export. Finally, Line 19 executes the instrumented version of the target program, which results from weaving the target program twice.

conduct.mjs

```
1  import { compileIntrinsicRecord } from "aran/runtime";
2  import {
3    createCustomAdvice,
4    createLibrary,
5    createRegion,
6    exposeLibrary,
7    registerAranIntrinsicRecord,
8  } from "linvail/runtime";
9  import { INTRINSIC_REGISTRY, FRONTIER_ADVICE, VIRTUAL_VALUE_ADVICE } from "./config.json";
10 const intrinsics = compileIntrinsicRecord(globalThis);
11 globalThis[INTRINSIC_REGISTRY] = intrinsics;
12 globalThis.performUnaryOperation = intrinsics["aran.performUnaryOperation"];
13 globalThis.performBinaryOperation = intrinsics["aran.performBinaryOperation"];
14 const region = createRegion(globalThis);
15 registerAranIntrinsicRecord(region, intrinsics);
16 exposeLibrary(createLibrary(region), { global: globalThis });
17 globalThis[FRONTIER_ADVICE] = createCustomAdvice(region);
18 globalThis[VIRTUAL_VALUE_ADVICE] = (await import("./advice.instr.mjs")).advice;
19 await import("./target.instr.instr.mjs");
```

Listing 6.17: Entry point of the online phase of our implementation of virtual values

Note that the analysis from Section 6.2.1 uses the `createStandardAdvice` export from `linvail/runtime`, whereas the analysis in this section uses the `createCustomAdvice` export from the same entry point. These two functions are similar in that they generate a generic advice to enforce Linvail's membrane. The key difference is that the former produces an advice in Aran's standard format, which requires Aran's standard weaving, while the latter produces an advice in a format specific to Linvail, which requires weaving specific to Linvail. Linvail-specific weaving is similar to Aran's standard weaving and was introduced to reduce performance overhead, for example by not maintaining a local advice state or code locations. We quantitatively compare the performance overhead of both weaving methods in Section 6.4.

Listing 6.18 illustrates the analysis-specific advice for our implementation of virtual values. Similar to the target program, this code generates state locations in the internal region, meaning that it exclusively manipulates handles. This is crucial for the advice code to utilize the functions from the `linvail/library` entry point with precision. The remainder of the analysis reads as follows:

- `createVirtualValue` (Lines 3–4): Our implementation of virtual values leverages a provenance-sensitive weak map to associate a virtual value with its target value and handler object. The virtual value is obtained by resetting the provenance of the target. This is crucial to ensure that intercession is applied to the virtual value and not its target.
- `test@before` (Lines 9–14): This advice function triggers the `test` handler of the `value` argument if applicable. Note that the target of the virtual value is passed to the trap, not the virtual value itself. This is consistent with the `Proxy` API and prevents infinite cycles.

- `apply@around` (Lines 15–30): This advice function serves a purpose similar to `test@before` in that it triggers virtual value traps when applicable. In accordance with Austin's paper, when both operands of a binary operation are virtual values, the left operand takes precedence.

advice.tex

```
1  import { ProvWeakMap, resetProvenance } from "linvail/library";
2  const registry = new ProvWeakMap();
3  globalThis.createVirtualValue = (target, handlers) => {
4    const fresh_target = resetProvenance(target);
5    registry.set(fresh_target, { target, handlers });
6    return fresh_target;
7  };
8  export const advice = {
9    "test@before": (_state, _kind, value, _location) => {
10     const hidden = registry.get(value);
11     if (hidden && hidden.handlers.test)
12       return hidden.handlers.test(hidden.target);
13     return value;
14   },
15   "apply@around": (_state, callee, that, input, _location) => {
16     if (callee.name === "performUnaryOperation" && input.length === 2) {
17       const hidden = registry.get(input[1]);
18       if (hidden && hidden.handlers.unary)
19         return hidden.handlers.unary(hidden.target, input[0]);
20     }
21     if (callee.name === "performBinaryOperation" && input.length === 3) {
22       // Precedence to the left if both operand are proxies
23       const hidden1 = registry.get(input[1]);
24       if (hidden1 && hidden1.handlers.left)
25         return hidden1.handlers.left(hidden1.target, input[0], input[2]);
26       const hidden2 = registry.get(input[2]);
27       if (hidden2 && hidden2.handlers.right)
28         return hidden2.handlers.right(hidden2.target, input[0], input[1]);
29     }
30     return Reflect.apply(callee, that, input);
31   },
32 };
```

Listing 6.18: Advice for our implementation of virtual values

Our implementation of virtual values further illustrates the expressiveness of the Aran Linvail stack. The code in `instrument.mjs` and `conduct.mjs` is mostly boilerplate and not analysis-specific. Only `advice.mjs` contains analysis-specific logic (32 lines of code) that is clear and understandable. Thanks to advice layering, this code is free from frontier concerns and is guaranteed not to violate the frontier. It should be noted that although the analysis is minimal, it implements an advanced reflection mechanism in a fully ECMAScript-compatible manner.

6.2.3 Taint analysis via push-based notifications

> *In this section, we demonstrate the flexibility of Linvail's promotion API by implementing dynamic taint analysis through advice extension.*

Dynamic taint analysis is a classic example of dynamic program analysis that requires sensitivity to value provenance. The technique typically involves tracking how data propagates from predetermined sources to designated sinks by attaching metadata to run-time values, referred to as their *taint* information. A key application of dynamic taint analysis is the enforcement of security policies at run-time [84, 25, 57].

In this section, we propose to implement a basic taint analysis via advice extension that leverages Linvail's ability to notify the analysis implementer when values cross its membrane. The core mechanism of our analysis involves maintaining a stack of contexts that reflects the tainting status of the current call. These contexts are categorized into four subtypes:

- Safe: For calls to dirty functions that have been instrumented and for which taint propagation can be observed.
- Sink: For calls to plain clean functions that represent sinks of information where tainted values must not propagate (`%console.log%` in our case).

- Source: For calls to plain clean functions that represent sources of tainted values. Initially, this context arises when the function is a root source (the export `question` from `readline-sync` in our case). Subsequently, this context also arises when a tainted value is observed crossing the membrane during a call to any plain clean function.
- Neutral: For calls to plain clean functions that are neither sinks nor sources. These contexts contain a `retro` field that stores the plain clean objects involved in the call. This is important for retroactively applying taint to these references when a neutral context becomes a source context. This accounts for the fact that an arbitrary external function may mutate the objects to which it has had access so far.

Listing 6.19 presents our taint analysis implemented as an extension of Linvail's generic frontier advice. The analysis reads as follows:

- `getInitialContext` (Lines 10–16): Computes the initial context for a call, given the callee as an external value. In this simple analysis, the only root source is the `question` function of `readline-sync`, and the only sink is the `%console.log%` intrinsic. Supporting more complex security policies would require adapting this code.
- `propagateTaint` (Lines 18–26): Taints the last element of the context stack which describes the current call. If the current context is a sink, an error is thrown to report a violation of the security policy. If the current context is neutral, all previously registered guest wrappers are tainted retroactively, and the current context is turned into a source. If the current context is safe or already a source, no action is taken.
- `promotePrimitive` (Lines 31–37): This function is called to promote primitive values as they enter the membrane. If the current context is tainted, a special handle is returned that will taint the current context whenever its inner primitive is accessed. These "smart" handles can be seen as a form of virtual values, as they modify the semantics of accessing one of their properties.
- `promotePlainCleanObject` (Lines 38–47): Similar to primitive values, plain clean objects can also be tainted. However, unlike primitive values, which are immutable, guest reference values can be retroactively tainted, necessitating the addition of a `tainted` property. Whenever a handle to a plain clean object is created, it is added to the current `retro` field (if applicable) to support retroactive tainting.
- Advice (Lines 51–69): The analysis-specific logic in the advice maintains the context stack. When entering an internal block, a safe context is pushed, and when leaving an internal block, the last context is popped; this last context should always be a safe context. A similar logic is applied when calling functions.

To demonstrate our analysis, we apply it to the program in Listing 6.20, which implements functionalities of a login prompt. Line 3 defines a function for redacting a login's password. Lines 4–8 wrap around `node:crypto` to hash data with salt. Lines 9–13 define a utility function to hash passwords with random salt. Lines 14–16 synchronously prompt the user for an admin password and hash it with random salt. Lines 17–19 redact the password from the login before displaying it, which does not violate the security policy. In contrast, Lines 20–21 attempt to display the unredacted login, which violates the security policy.

While our taint analysis is straightforward and consists of only 70 lines of code, it performed accurately on Listing 6.20, which exhibits relatively complex information flow despite its apparent simplicity. Below, we provide an account of the information flow and explain how our analysis keeps pace with it.

- `14`: The admin password is synchronously prompted. This external call is contextually tainted as the callee is a root source.
- `15`: The tainted password is stored in a dirty object. The entire object is not tainted; only `plain_pass` property value is.
- `16`: The password is salted by calling the dirty function `saltPassword`.
- `11`: The plain password is hashed by calling the dirty function `hashPassword`.

```
import { log } from "node:console";
import { question } from "readline-sync";
import {
  createMembrane,
  createRegion,
  createStandardAdvice,
  registerAranIntrinsicRecord,
} from "linvail/runtime";

const getInitialContext = (closure) => {
  switch (closure) {
    case question: return { type: "source" };
    case log: return { type: "sink" };
    default: return { type: "neutral", retro: new Set() };
  };
};

const propagateTaint = (contexts) => {
  if (contexts.at(-1).type === "sink")
    throw new Error("Violation of taint policy");
  if (contexts.at(-1).type === "neutral") {
    for (const wrapper of contexts.at(-1).retro)
      wrapper.tainted = true;
    contexts.splice(-1, 1, { type: "source" });
  }
}

export const createAdvice = (intrinsics) => {
  const contexts = [{ type: "safe" }];
  const region = createRegion(intrinsics.globalThis, {
    promotePrimitive: (primitive) => contexts.at(-1).type === "source" ? {
      type: "primitive",
      get inner () {
        propagateTaint(contexts);
        return primitive;
      },
    } : { type: "primitive", inner: primitive },
    promotePlainCleanObject: (guest, name) => ({
      type: "plain-clean-object",
      name,
      tainted: contexts.at(-1).type === "source",
      get inner () {
        if (this.tainted) propagateTaint(contexts);
        else contexts.at(-1).retro?.add(this);
        return guest;
      },
    }),
  });
  registerAranIntrinsicRecord(region, intrinsics);
  const { apply, construct } = createMembrane(region);
  return {
    ...createStandardAdvice(region),
    "block@setup": (_state, _kind, _location) => {
      contexts.push({ type: "safe" });
    },
    "block@teardown": (_state, _kind, _location) => {
      contexts.pop();
    },
    "apply@around": (_state, callee, that, input, _location) => {
      contexts.push(getInitialContext(callee.inner));
      try { return apply(callee, that, input); }
      finally { contexts.pop(); }
    },
    "construct@around": (_state, callee, input, _location) => {
      contexts.push(getInitialContext(callee.inner));
      try { return construct(callee, input, callee); }
      finally { contexts.pop(); }
    },
  };
};
```

Listing 6.19: Advice for preventing the display of sensitive information

```
1  import { question } from "readline-sync";
2  import { createHash } from "node:crypto";
3  const redactLogin = (login) => ({ ...login, pass: "****" });
4  const hashPassword = (data, salt) => {
5    const hash = createHash("sha256");
6    hash.update(data + salt);
7    return hash.digest('hex');
8  };
9  const saltPassword = ({ role, plain_pass }) => {
10   const salt = Math.random().toString().substring(2, 6);
11   const pass = hashPassword(plain_pass, salt);
12   return { role, salt, pass };
13 };
14 const plain_pass = question("Password: ");
15 const plain_login = { role: "admin", plain_pass };
16 const login = saltPassword(plain_login);
17 const safe_login = redactLogin(login);
18 const safe = JSON.stringify(safe_login);
19 console.log(safe);      // ✅ {"role":"admin","salt":"4071","pass":"****"}
20 const sensitive = JSON.stringify(login);
21 console.log(sensitive); // ❌ Error: Violation of security policy
```

Listing 6.20: Password prompt that violates the security policy defined in our analysis

- `5`: A hash object is created by calling the `createHash` method of the `node:crypto` package. This object is a plain clean object and is not initially tainted.
- `6`: The tainted password is concatenated with the salt, resulting in a tainted string that is passed to the `update` method of the hash object. This causes the external call to be contextually tainted, retroactively tainting the hash object.
- `7`: The `digest` method is called on the tainted hash object, which causes the external call to be contextually tainted, resulting in a tainted string.
- `12`: The salted and hashed password (which is a tainted string) is assigned to the `pass` property of the dirty object returned by the internal call to `saltPassword`.
- `17`: The dirty `login` object (whose `pass` property is a tainted string) is passed to the `redact` function.
- `3`: A shallow copy of the first argument is created. The `pass` property is then set to `"****"`, which is not tainted. The resulting object is dirty and does not contain any tainted properties.
- `18`: The redacted login is converted to a string by calling `JSON.stringify`. This function recursively explores all the properties of the given value. Since none of them are tainted, the resulting string is not tainted as well.
- `19`: The safe representation of the login object is passed to the `%console.log%` intrinsic, which is a sink. However, this does not cause an error to be thrown because the provided argument is not tainted.
- `20`: The unredacted login object is passed to the `JSON.stringify` function. However, accessing the `pass` property of the login object contextually taints the call, resulting in a tainted string.
- `21`: The sensitive representation of the login is passed to the `%console.log%` intrinsic which throws an exception that indicates a violation of the security policy.

Note that we do not claim that our taint analysis is sound. First, Linvail does not comprehensively track all data, such as that contained within exotic objects or the boolean flags associated with properties. Second, our analysis tracks only explicit information flow; it does not account for implicit information flow [62], where information is conveyed through control flow. Third, and perhaps most importantly, our context stack represents an unsound approximation. For instance, arbitrary plain clean functions have access to the global object and can bypass our analysis by assigning tainted values to global variables.

Nonetheless, our analysis demonstrates the ability of the Aran Linvail stack to accurately propagate taint through a relatively complex information flow with the straightforward logic presented in Listing 6.19. This positive result stems from the combination of several functionalities of our infrastructure:

- Aran Normalization: Aggressive normalization of many syntactic constructs into call expressions greatly simplifies the advice necessary to maintain the context stack. For instance, the analysis advice does not require specific handling of the spread operator used to shallow copy the login object in the `redact` function.
- Basic Provenance Tracking: Linvail interprocedurally tracks the provenance of values through the stack and the scope, providing built-in basic taint propagation.
- Provenance Tracking in the Store: Thanks to the presence of handles in the store, login objects are tainted precisely at the property level rather than indiscriminately. This prevents over-tainting and allows the safe representation of the redacted login object to be printed.
- Flexible Promotion: While our symbolic execution analysis from Section 6.2.1 focuses on linking the result of a call to its arguments, our taint analysis leverages Linvail's flexible promotion API to also account for implicit inputs and outputs. This enables the detection and propagation of taint when calling `JSON.stringify` on the unredacted login object.

6.3 Evaluating Linvail's semantic transparency by applying it to Test262

In this section, we present another experiment targeting the Test262 suite. The main goal of this experiment is to evaluate the semantic transparency of the Aran Linvail stack. Unlike our first Test262 experiment from Section 4.3, which focused solely on Aran, this experiment incorporates analyses built on the Aran Linvail stack. This section is organized as follows:

- *Section 6.3.1 outlines the setup of the experiment.*
- *Section 6.3.2 discusses the findings of the experiment.*

6.3.1 Setup of the second Test262 experiment

In this section, we detail the setup of our second Test262 experiment whose goal is to evaluate Linvail's semantic transparency by applying it to Test262 cases.

The setup of the second Test262 experiment for Linvail is mostly the same as that for Aran; only the participating analyses differ. In particular, the test runner remains identical, and the software version from Table 4.2 remains unchanged; the only addition is the Linvail dependency, which has version `v7.7.9` and commit SHA `9c98623`[6].

Next, we detail the participating analyses, each of which has two deployment modes. The partial deployment instruments only the main file of the test case and its dynamically evaluated local code. In contrast, the comprehensive deployment also instruments dynamically evaluated global code, test harness files, and dependency modules. The comprehensive deployment instructs Aran to reify the global declarative record, allowing for full control of the global scope. This enables analyses built on top of Linvail to turn the global object and the global declarative records into dirty objects, which improves precision. Our four participating analyses are:

- `none`[7]: This no-op analysis is identical to the `none` analysis from the similar experiment in Section 4.3.1. Since it performs no operations, it has only one deployment mode. It is retained to establish a baseline execution for calculating the slowdown factor of subsequent analyses.

[6] `https://github.com/lachrist/linvail/tree/9c98623`
[7] `https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/identity.mjs`

- `bare`[8,9]: The two deployment modes of this analysis correspond to the `bare-main` and `bare-comp` analyses from Section 4.3.1, respectively. They are retained to compare the slowdown factors of Aran and Linvail against the slowdown of Aran alone.
- `stnd`[10,11]: These analyses deploy Linvail using standard Aran advice, which necessitates standard Aran weaving. Although these analyses do not perform any operations, they would enable provenancial equality if Test262 cases were to import the `linvail/library` entry point.
- `cust`[12,13]: These analyses deploy Linvail using Linvail-specific advice, which requires custom weaving which was briefly introduced in Section 6.2.2. Although these analyses are similar to the `stnd` analyses, they should be slightly faster because the weaving has been optimized for Linvail.

6.3.2 Semantic overhead observed during the second Test262 experiment

In this section, we discuss the semantic discrepancies observed during our second Test262 experiment.

The corpus for this experiment comprises 80,205 Test262 cases, which were obtained by excluding the following cases from the initial corpus of 92,728 cases:

- We excluded the 11,562 cases that failed during our Test262 experiment targeting Aran alone.
- An additional 670 cases were excluded because they were automatically generated large test files designed to exercise escape properties in regular expressions. These files were not relevant for assessing the transparency of Linvail and hindered our functional test suite's performance.
- An additional 291 cases were excluded due to their slow performance, most of which targeted regular expressions, array buffers, and typed arrays. We believe these cases were also not relevant for uncovering transparency issues in Linvail.

By carefully designing Linvail, we reduced the semantic discrepancies to only 40 failures out of a corpus of 80,205 Test262 cases, resulting in a failure rate of just 0.05%. Given the complexity of the Linvail library, we consider this a remarkable achievement. Below, we detail these discrepancies:

- Wrong realm for default array prototype (18): When creating objects, the prototype may depend on the realm of the `new.target` argument. Specifically, when an array is created via `%Array%`, if the `new.target` argument has its `prototype` property set to `undefined`, the intrinsic `%Array.prototype%` of the `new.target` argument's realm is used. Since realms cannot be accessed from within JavaScript, Linvail cannot retrieve the `%Array.prototype%` from any realm other than the ambient realm. Consequently, the Test262 assertion in Listing 6.21 fails via Linvail advising.
- No cycle detection in prototype chain (8): Normally, if updating the prototype of an object would introduce a cycle in the prototype chain, an instance of `%TypeError%` should be thrown. However, the ECMAScript specification mandates stopping detection as soon as a proxy is encountered in the prototype chain. This is to avoid triggering the `getPrototypeOf` trap, which can have side effects. Due to the pervasive use of proxies by Linvail, cycles in the prototype chain may not be detected. Consequently, the Test262 assertion in Listing 6.22 fails via Linvail advising.
- Elusive dynamic code evaluation (12): The comprehensive deployment assumes that every bit of code is instrumented to effectively emulate the global scope. This involves intercepting calls to built-in functions capable of dynamically evaluating a string as JavaScript code to insert instrumentation. Our detection is not foolproof, which can lead to discrepancies; however, it only occurred in six test cases. For example, invoking `%eval%` reflectively through `%Reflect.apply%` is not recognized as dynamic code evaluation by our test runner.

[8] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/bare-main.mjs
[9] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/bare-comp.mjs
[10] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/linvail/stnd-main.mjs
[11] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/linvail/stnd-comp.mjs
[12] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/linvail/stnd-main.mjs
[13] https://github.com/lachrist/aran/blob/161d57e/test/262/staging/spec/linvail/stnd-comp.mjs

- Bug in V8 related to proxy (2): We discovered a bug in V8 related to the Proxy, which we reported[14]. The issue requires a convoluted setup to be observed: a custom `this` argument must be provided to `%Array.fromAsync%`, which should create an array with a non-writable length wrapped around a proxy. This scenario is expected to throw an instance of `%TypeError%`, but instead, a sparse array is returned.

```
const other = $262.createRealm().global;
const new_target = new other.Function();
new_target.prototype = undefined;
assert.sameValue(
  Reflect.getPrototypeOf(
    Reflect.construct(Array, [1], new_target),
  ),
  other.Array.prototype,
);
```

Listing 6.21: Test262 assertion for realm preservation in `new.target`

```
const external = globalThis;
const internal = { __proto__: external };
assert(!Reflect.setPrototypeOf(external, internal));
```

Listing 6.22: Test262 assertion for detecting cycles in the prototype chain

6.4 Evaluating Linvail's performance overhead by applying it to Octane

In this section, we present a second experiment targeting the Octane benchmark suite. Unlike our first Octane experiment in Section 4.4, this one evaluates provenance-aware analyses built with the Aran Linvail stack. This section is organized as follows:

- *Section 6.4.1 discusses the setup of the experiment.*
- *Section 6.4.2 discusses the findings of the experiment.*

6.4.1 Setup of the second Octane experiment

In this section, we detail the setup of our second Octane experiment whose goal is to evaluate Linvail's performance overhead by applying it to Octane benchmarks.

This experiment is similar to our first Octane experiment described in Section 4.4.1, as it targets the same set of benchmark programs and utilizes the same test runner. As such, in contrast to both our Test262 experiments, instrumentation is performed offline. This means that dynamically evaluated code cannot be instrumented, which can pose challenges for analyses that rely on comprehensive instrumentation, such as those requiring Aran to reify the global declarative record.

During this experiment, we encountered memory issues; to mitigate some of these, we executed each Octane benchmark with the following memory-related Node flags:

- `--max-old-space-size=8192`: To prevent out of memory errors, we raised the maximum memory size for storing long-lived objects to 8 GB. Note that as this value increases, the frequency of major garbage collections decreases, but each cycle takes longer.
- `--max-semi-space-size=256`: We also increased the maximum memory size for storing short-lived objects to 256 MB. As this value increases, the frequency of minor garbage collections decreases, but each cycle may take longer. The 256 MB value was determined empirically through a few trial-and-error runs during our experiment.

[14] `https://github.com/nodejs/node/issues/56952`

We now present the analyses participating in this second Octane experiment, which focuses on Linvail. As with the other experiments, the first analysis is a no-op analysis that does not perform any instrumentation. It is labeled as `none`[15] and serves as a baseline for computing the slowdown factor for subsequent analyses.

The first set of four analyses simply deploys Linvail, which has no observable effects other than enabling provenancial equality via the `linvail/library` entry point. Table 6.2 summarizes these four basic Linvail analyses, providing their label and path[16]. These four analyses result from exploring two orthogonal design decisions:

- The first design decision consists of choosing the format of the advice and the weaving it requires to function. The first option adheres to the standard Aran advice format, enabling standard Aran weaving. The second option employs the custom Linvail advice format and its corresponding custom weaving. We expect this second option to be slightly faster and to result in less code bloat.
- The second design decision involves determining whether to internalize the global scope. Internalizing the global scope involves maintaining a shallow copy of the global object and the reified declarative record with internal values. This process enables tracking primitives through the global scope but requires comprehensive instrumentation to function properly.

Global Scope	**Standard Aran Weaving**	**Custom Linvail Weaving**
External	`stnd: linvail/standard/external`	`cust: linvail/custom/external`
Internal (*)	`stnd*: linvail/standard/internal`	`cust*: linvail/custom/internal`

Table 6.2: The four naked Linvail analyses from our second Octane experiment

The second set of four analyses utilizes Linvail to collect symbolic constraints that can be provided to a constraint solver for generating input to drive execution along a specific path. Unlike the similar analyses in Sections 5.4.2 and 5.4.3, which keep in memory the entire value flow graph, these analyses mitigate memory issues by logging symbolic constraints during execution. None of our symbolic execution analyses internalize the global scope. Table 6.3 presents our four symbolic execution analyses providing their label and path [17]. These four analyses result from exploring two orthogonal design decisions:

- The first design decision relates to advice composition, specifically whether to adopt the advice extension approach from Section 5.4.2 or the advice layering approach from Section 5.4.3. As a reminder, in the advice extension approach, Linvail's generic frontier advice is augmented with analysis logic. In contrast, the advice layering approach separates concerns by advising on provenance and analysis independently. Since advice layering requires instrumenting the target program twice, we expect it to incur greater performance overhead.
- The second design decision is whether to record symbolic constraints. The no-op record is useful for isolating the cost of the analysis itself, while the file record provides a more realistic estimate of performance overhead. To minimize overhead, our file recording relies on a buffering system that flushes to the trace file. To ensure memory is freed, this flushing is performed synchronously because Octane benchmarks execute synchronously as well.

Recording	**Advice Extension**	**Advice Layering (+)**
No-op	`sym: symbolic/intensional/void`	`sym+: symbolic/extensional/void`
File (`!`)	`sym!: symbolic/intensional/file`	`sym+!: symbolic/extensional/file`

Table 6.3: The four symbolic execution analyses from our second Octane experiment

[15] `https://github.com/lachrist/aran/tree/1c51bf8/test/bench/meta/none`
[16] Relative to `https://github.com/lachrist/aran/tree/1c51bf8/test/bench/meta`
[17] Relative to `https://github.com/lachrist/aran/tree/1c51bf8/test/bench/meta`

6.4.2 Performance overhead observed during the second Octane experiment

In this section, we discuss the performance overhead and failures observed during our second Octane experiment.

The results of the second Octane experiment are summarized in Tables 6.4 and 6.5.

First, we discuss the failures observed during the experiment, which are categorized as follows:

- Timeout (2): To conduct the experiment within a reasonable timeframe, we set a timeout of 2 hours for each repetition of any Octane benchmark. This restriction caused the `sym+` analysis to fail when executing the `gbemu` and `typescript` benchmarks.
- Heap (3): We configured the maximum heap allocation to 8 GB during both instrumentation and execution. This restriction caused the analyses `stnd*`, `cust*`, and `sym+` to fail when instrumenting `mandreel`. We further discuss these failures in our review of the code bloat factor.
- Trace (2): To preserve system stability, we limited the trace recorded by the `sym!` analysis to 256 GB. This limitation caused failures for the `crypto` and `mandreel` benchmarks. These performance-related discrepancies should not be attributed to the Aran Linvail stack; rather, it is the responsibility of the analysis to bound the amount of recorded data.
- RefErr (2): The analyses that rely on comprehensive instrumentation, namely `stnd*` and `cust*`, both triggered a `ReferenceError` during their application to the `code-load` benchmark. This behavior is expected, as this benchmark features dynamic code evaluation. These semantic discrepancies should not be attributed to the Aran Linvail stack; they are caused by the test runner, which does not support instrumentation at run-time.

Second, we focus on the code bloat factor, which is statistically summarized in Figure 6.8. Below, we present our main findings:

- After Aran normalization, which incurs approximately a 10× code bloat, the standard weaving required by Linvail incurs an additional multiplicative 10× code bloat. This is less than the `full` analysis from our first Octane experiment, which is expected since Linvail does not need to advise all standard join points.
- The custom Linvail weaving produces code that is roughly twice as compact as standard Aran weaving. A major reason for this outcome is that the functions of the custom Linvail advice do not require a state or a location argument.
- Internalizing the global scope results in a noticeable 50% increase in code bloat for both standard and custom weaving. This is explained by the fact that when the global scope is internalized, global variable lookups become explicit, which causes more AST nodes to be inserted. We believe this is the reason why both `stnd*` and `cust*` ran out of memory during the instrumentation of `mandreel`, whereas both `stnd` and `cust` did not.
- The instrumentation required for advice layering generates code that is approximately three times the size of code generated by the instrumentation required for advice extension. This is expected because advice layering stacks instrumentation, which, as discussed in Section 5.4.3, has compounding effects. This explains why `sym+` ran out of memory during the instrumentation of `mandreel`.

Our review of the code bloat factor indicates that instrumentation may become problematic for large files, especially when they heavily rely on global variables. To address this issue, the target program should be separated into smaller files. For instance, if the build chain features bundling, better performance can be achieved by instrumenting input modules separately instead of instrumenting the entire output bundle at once.

Third and finally, we focus on the slowdown factor, which is statistically summarized in Figure 6.9. Below, we present our main findings:

	Size [KB]	Time [ms]
	box2d	
none	305	3.47
bare	4712	473
cust	16012	7086
cust*	17865	7099
stnd	52820	9221
stnd*	59856	9321
sym	52820	12826
sym!	52820	44154
sym+	134856	3588155
	crypto	
none	47	1.74
bare	768	163
cust	2545	10621
cust*	4401	10668
stnd	6197	11566
stnd*	11205	11649
sym	6197	21148
sym!	6197	Trace
sym+	19334	2223182
	code-load	
none	117	2.54
bare	287	3.68
cust	695	8.59
cust*	1170	RefErr
stnd	1641	9.49
stnd*	2655	RefErr
sym	1641	11.0
sym!	1641	22.1
sym+	4694	482
	earley-boyer	
none	149	2.55
bare	1753	890
cust	5573	7154
cust*	11959	7184
stnd	15376	8863
stnd*	37923	9232
sym	15376	11535
sym!	15376	28709
sym+	45021	1877719

	Size [KB]	Time [ms]
	deltablue	
none	22	0.05
bare	388	94.0
cust	1294	539
cust*	1835	542
stnd	3281	723
stnd*	4882	741
sym	3281	886
sym!	3281	1799
sym+	9999	56585
	gbemu	
none	450	18.3
bare	4999	1956
cust	16306	30272
cust*	18549	30262
stnd	49707	36815
stnd*	56426	37692
sym	49707	57678
sym!	49707	197082
sym+	130866	Timeout
	mandreel	
none	5975	19.3
bare	28791	8396
cust	102175	139221
cust*	Heap	N/A
stnd	345280	245687
stnd*	Heap	N/A
sym	345280	348048
sym!	345280	Trace
sym+	Heap	N/A
	navier-stokes	
none	18	3.4
bare	291	37.1
cust	940	7448
cust*	1198	7446
stnd	2539	8696
stnd*	3310	8705
sym	2539	14506
sym!	2539	53587
sym+	7256	1701559

Table 6.4: Results of the second Octane experiment (1/2)

	Size [KB]	Time [ms]
	pdfjs	
none	1455	10.4
bare	6632	507
cust	21446	9913
cust*	22571	9859
stnd	89555	12194
stnd*	94717	12499
sym	89555	17896
sym!	89555	98968
sym+	195511	4791681
	raytrace	
none	28	0.705
bare	382	262
cust	1302	1906
cust*	1824	1920
stnd	3803	2383
stnd*	5581	2416
sym	3803	3056
sym!	3803	8093
sym+	10623	228075
	regexp	
none	131	7.03
bare	436	104
cust	1549	1243
cust*	1927	1244
stnd	4479	1628
stnd*	5626	1635
sym	4479	2344
sym!	4479	6090
sym+	12569	206650

	Size [KB]	Time [ms]
	richards	
none	17	0.075
bare	269	35.4
cust	875	278
cust*	1295	277
stnd	2249	361
stnd*	3478	367
sym	2249	475
sym!	2249	1126
sym+	6771	37685
	splay	
none	14	0.273
bare	201	8.31
cust	650	82.8
cust*	979	82.3
stnd	1698	85.2
stnd*	2673	87.3
sym	1698	112
sym!	1698	278
sym+	5040	37685
	typescript	
none	2267	144
bare	11735	11490
cust	32187	746662
cust*	35607	818823
stnd	117407	737694
stnd*	123567	700154
sym	117407	772515
sym!	117407	623273
sym+	260372	Timeout

Table 6.5: Results of the second Octane experiment (2/2)

Analysis	**Mean** [KB]	**P25** [KB]	**P50** [KB]	**P75** [KB]
`bare`	12	5	14	16
`cust`	38	17	46	52
`cust*`	57	35	66	77
`stnd`	108	62	121	136
`stnd*`	165	110	194	209
`sym`	108	62	121	136
`sym!`	108	62	121	136
`sym+`	316	252	370	405

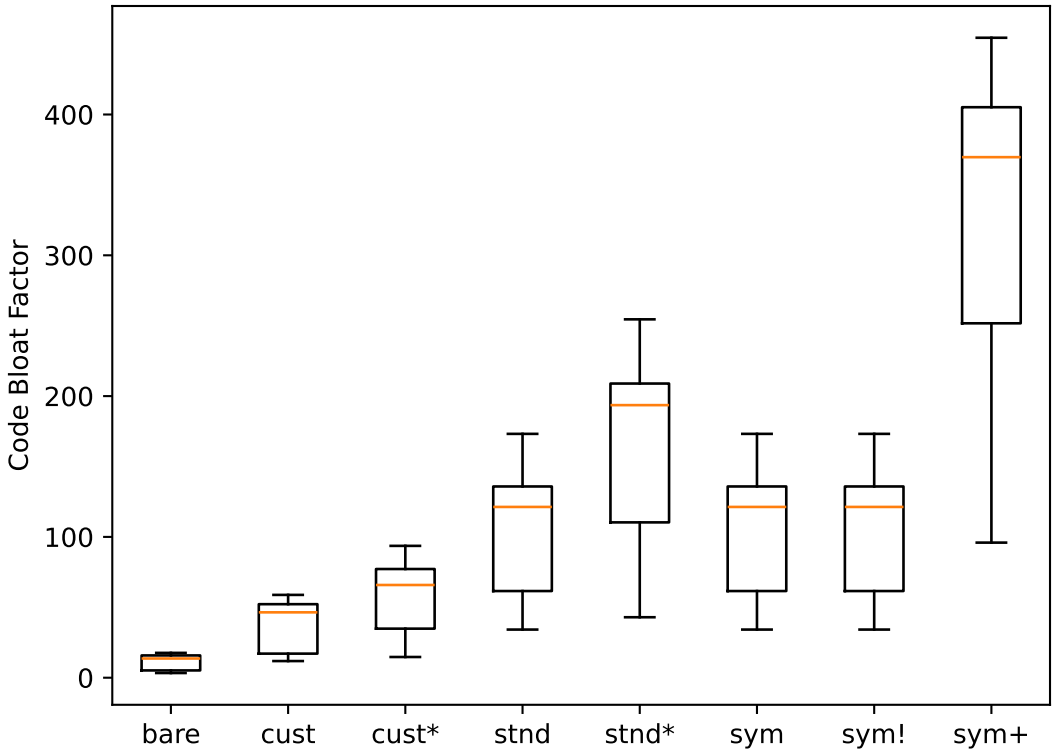


Figure 6.8: Statistical summary of the code bloat factor after instrumenting each Octane benchmark (excluding `code-load`) across all participating analyses

- As discussed in Section 4.4, the `bare` analysis already incurs an approximate slowdown of 100×. The naked deployment of Linvail introduces an additional multiplicative slowdown factor of approximately 10×, resulting in execution times that are thousands of times slower.
- Custom weaving is noticeably faster than standard weaving, with a performance increase of about 20%.
- Internalizing the global scope improves performance by approximately 10%. We believe this is due to the Octane benchmark's heavy reliance on global variables, which causes a significant amount of crossing across Linvail's membrane when the global scope is external.
- The simple analysis logic introduced by symbolic execution resulted in a noticeable performance decrease of about 40%.
- During symbolic execution, the writing of constraints to a file roughly doubled the overhead of the analysis, resulting in an overall slowdown reaching $10,000\times$.
- Advice layering for symbolic execution incurs a multiplicative slowdown factor of approximately 100× over advice extension, leading to an overall slowdown factor reaching several $100,000\times$.

Our review of the slowdown factors indicates that analyses built with advice extension can be expected to incur a slowdown factor reaching the thousands. This slowdown factor is comparable to that observed during our first Octane experiment for the `track` analysis. Therefore, we believe that Linvail is as applicable as an ad-hoc solution for tracking primitive values within the Aran instrumentation framework.

In contrast, advice layering incurs a significant performance overhead, and analyses developed using this architecture can be expected to experience a slowdown factor reaching the hundreds of thousands. This severely limits the applicability of such analyses to target programs that execute extremely quickly, such as unit tests.

Analysis	Mean	P25	P50	P75
bare	310	49	107	371
cust	3523	1657	2702	5178
cust*	3266	1480	2455	4190
stnd	4583	2015	3379	5116
stnd*	3947	1849	3057	4865
sym	6311	3157	4333	6325
sym!	11705	6942	11238	13871
sym+	613422	358293	500915	959938

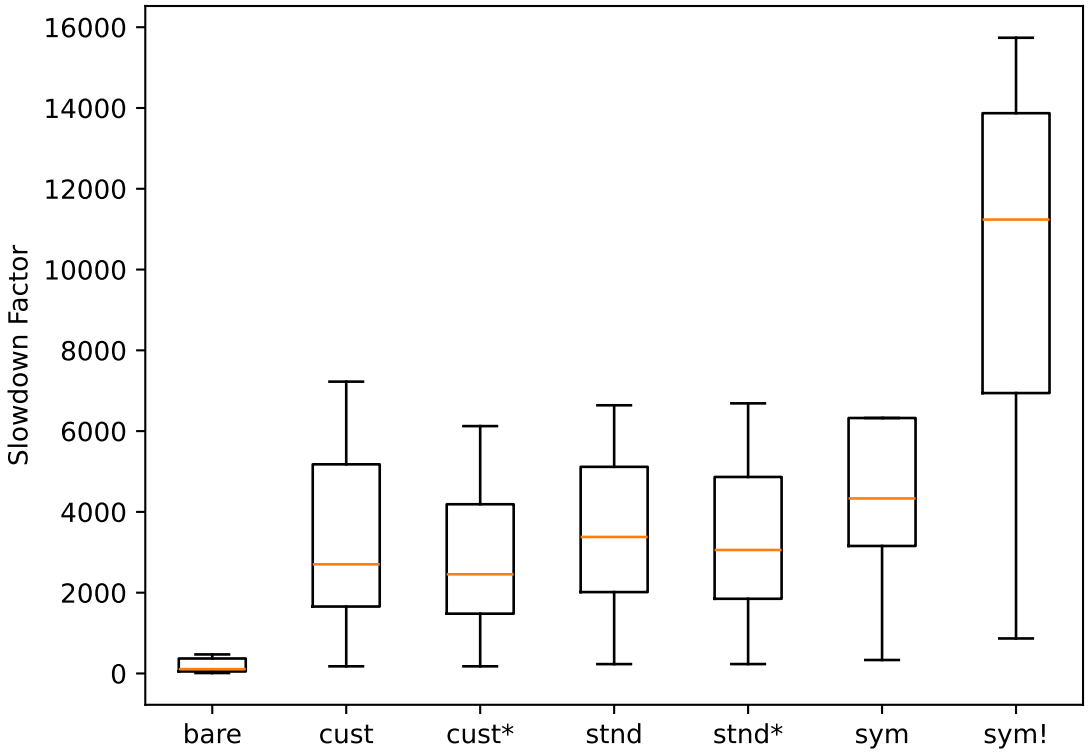


Figure 6.9: Statistical summary of the slowdown factors observed when executing each Octane benchmark (excluding `code-load`) across all participating analyses (excluding `sym+` from visualization)

6.5 Evaluating Linvail's tracking precision by applying it to Aran

In this section, we present the third and final experiment in which we apply analyses from previous experiments to Aran itself. The goal is to demonstrate that the Aran Linvail stack can manage large codebases and provide an initial account of our provenance metric. This section is organized as follows:

- *Section 6.5.1 discusses the setup of the experiment.*
- *Section 6.5.2 discusses the performance overhead observed during the experiment.*
- *Section 6.5.3 discusses the provenance scores observed during the experiment.*

6.5.1 Setup of the meta-Aran experiment

In this section, we describe both the target program of the experiment and the participating analyses.

Our final experiment applies the analyses from Sections 6.3 and 6.4 to Aran itself. The objective is to assess the overhead and precision of Linvail on executions that fall between short Test262 cases and longer Octane benchmarks. This setup differs from previous experiments in that the target programs are significantly larger and more up-to-date with the latest JavaScript standard; specifically, native modules are used in place of the now-discouraged reliance on global variables.

Table 6.6 lists the participating analyses, providing their label, weaving process, whether they internalize the global scope, and a brief description. We categorize these 17 analyses as follows:

- Baseline (1): A no-op analysis included to provide a baseline execution.
- Aran (3): Three representative Aran-only analyses, as this experiment was not reported in Chapter 4.
- Linvail (4): The four naked Linvail analyses from our second Test262 experiments.
- Symbolic (4): The four symbolic execution analyses from our second Octane experiment.
- Provenance Metric (5): Five analyses for recording the provenance score (cf. Section 5.5) of values used to branch the control flow at the `test@before` join point. These analyses are implemented via advice extensions and enforce a frontier around an increasingly larger internal region:
  - `stack`: Stack-only provenance tracking obtained by restricting the internal region to the state locations within the value stack (or continuation for a CESK-based interpreter).
  - `intra`: Intra-procedural provenance tracking obtained by extending the internal region to state locations within the environment (cf. Section 5.2.1).
  - `inter`: Inter-procedural provenance tracking obtained by extending the internal region to argument and result state locations (cf. Section 5.2.3).
  - `store`: Heap-level provenance tracking enabled by Linvail and obtained by extending the internal region to state locations within the store (cf. Section 5.3.3).
  - `store*`: Similar to the `store` analysis, but extends the internal region to the global scope, which includes both the global declarative record and the global object. This analysis requires comprehensive instrumentation to be transparent.

The target program is an instrumenter built with the following pipeline: code parsing with Acorn, AST instrumentation with Aran, and code generation with Astring. Our experiment involves exercising it on three increasingly complex programs. The first input program is minimal and consists of a primitive literal (`"123;"`). The second input program is slightly more complex and is shown in Listing 6.23. The third input program is the `deltablue` benchmark from Octane and consists of 25,726 characters.

Label	Weaving	Global Scope	Analysis
`none`	(None)	(N/A)	Nothing (no instrumentation)
`bare`	(None)	(N/A)	Nothing (Aran normalization)
`full`	Standard	(N/A)	Comprehensive forward logic.
`track`	Standard	(N/A)	Track origin analysis from Section 4.3.1
`stnd`	Standard	External	Enable provenancial equality
`stnd*`	Standard	Internal	Enable provenancial equality
`cust`	Custom	External	Enable provenancial equality
`cust*`	Custom	Internal	Enable provenancial equality
`sym`	Standard	External	Track symbolic constraints
`sym!`	Standard	External	Record symbolic constraints
`sym+`	Custom → Standard	External	Track symbolic constraints
`sym+!`	Custom → Standard	External	Track symbolic constraints
`stack`	Standard	External	Record stack-restricted provenance score
`intra`	Standard	External	Record intra-procedural provenance score
`inter`	Standard	External	Record inter-procedural provenance score
`store`	Standard	External	Record Linvail provenance score
`store*`	Standard	Internal	Record Linvail provenance score (through the global scope)

Table 6.6: Summary of analyses participating in the meta-Aran experiment

```
class Person {
  name = "anonymous";
  age = 0;
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log(`Hello, I'm ${this.name}; ${this.age}!`);
  }
}
const alice = new Person("Alice", 30);
alice.greet(); // "Hello, my name is Alice, I'm 30!"
```

Listing 6.23: The `person.mjs` target program

The test runner for this experiment performs offline instrumentation, similar to our Octane experiments. Consequently, dynamically evaluated code cannot be instrumented; however, this limitation is not problematic since neither Aran, Acorn, nor Astring utilizes this feature. To facilitate offline instrumentation, our test runner relies on Esbuild `v0.25.2`[18] to bundle the target instrumenter program into a single file, which consists of 1,004,188 characters.

The hardware and software configurations presented in Table 4.2 remain applicable, while the version for Linvail remains `v7.7.9`.

6.5.2 Performance overhead observed during the meta-Aran experiment

In this section, we discuss the performance overhead observed during the experiment.

Table 6.7 summarizes the time required to instrument each of the three instrumentation cases across the 17 participating analyses. All executions were repeated five times to account for transient run-time noise. The reported times were obtained by trimming the minimum and maximum values and averaging the remaining ones.

Analysis	123 [ms, X]		person [ms, X]		deltablue [ms, X]	
none	0.281	1	2.315	1	23.501	1
bare	1.149	4	11.739	5	274.204	12
full	15.672	56	151.395	65	3301.085	140
track	15.937	57	233.378	101	7423.863	316
cust	8.788	31	154.518	67	5724.437	244
cust*	8.804	31	154.633	67	5863.425	250
stnd	15.514	55	232.555	100	7895.623	336
stnd*	15.707	56	231.888	100	7880.129	335
stack	18.079	64	363.927	157	14824.941	631
intra	18.531	66	392.629	170	25093.819	1068
inter	18.613	66	389.250	168	15308.385	651
store	20.810	74	439.734	190	17077.646	727
store*	20.865	74	443.992	192	17767.659	756
sym	18.091	64	298.158	129	10683.605	455
sym!	26.640	95	565.180	244	41371.534	1760
sym+	733.796	2611	71947.688	31074	4120278.108	175326
sym+!	706.770	2515	65020.307	28082	3399940.516	144674

Table 6.7: The time and slowdown factor necessary to conduct each of the three instrumentation cases across the 17 participating analyses

Figure 6.10 provides a linear visualization of the data from Table 6.7. Both symbolic analyses implemented via advice layering were removed due to their significantly larger slowdown factors, which made the graph harder to read. However, they are included in Figure 6.11, which provides a comprehensive visual account of the data on a logarithmic scale.

In the remainder of this section, we discuss the main findings regarding the overhead introduced by Aran and Linvail in the meta-Aran experiment.

Anomaly for the `intra` analysis In the `deltablue` instrumentation case, the `intra` analysis incurs a significantly greater performance overhead compared to the `inter` analysis. This is surprising, as these analyses are closely related; in the other two cases, this abnormal "spike" is not observed. To investigate this issue, we recovered the five measurements of time for each analysis. This data is shown in Table 6.8 for the `intra` and `inter` analyses when applied to the `deltablue` instrumentation case. It turns out that three of the five `intra` measures are comparable to the `inter` measures, while two of them are much

[18] `https://github.com/evanw/esbuild/tree/v0.25.2`

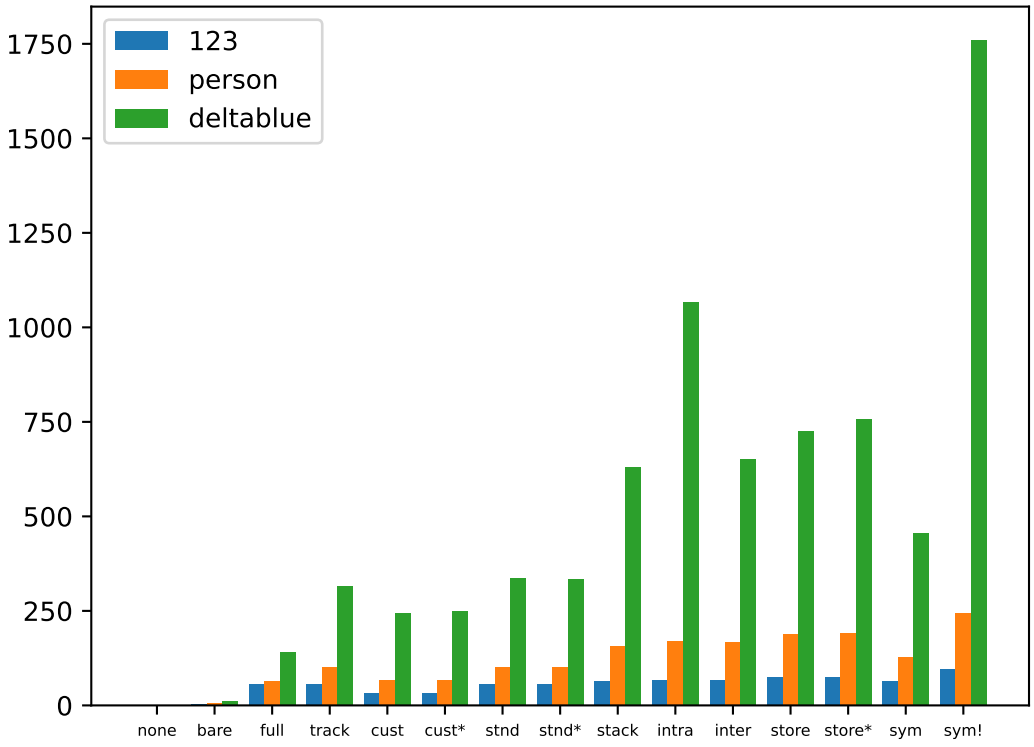


Figure 6.10: Slowdown factors across all three instrumentation experiments for each participating analysis (excluding `sym+` and `sym!` from visualization)

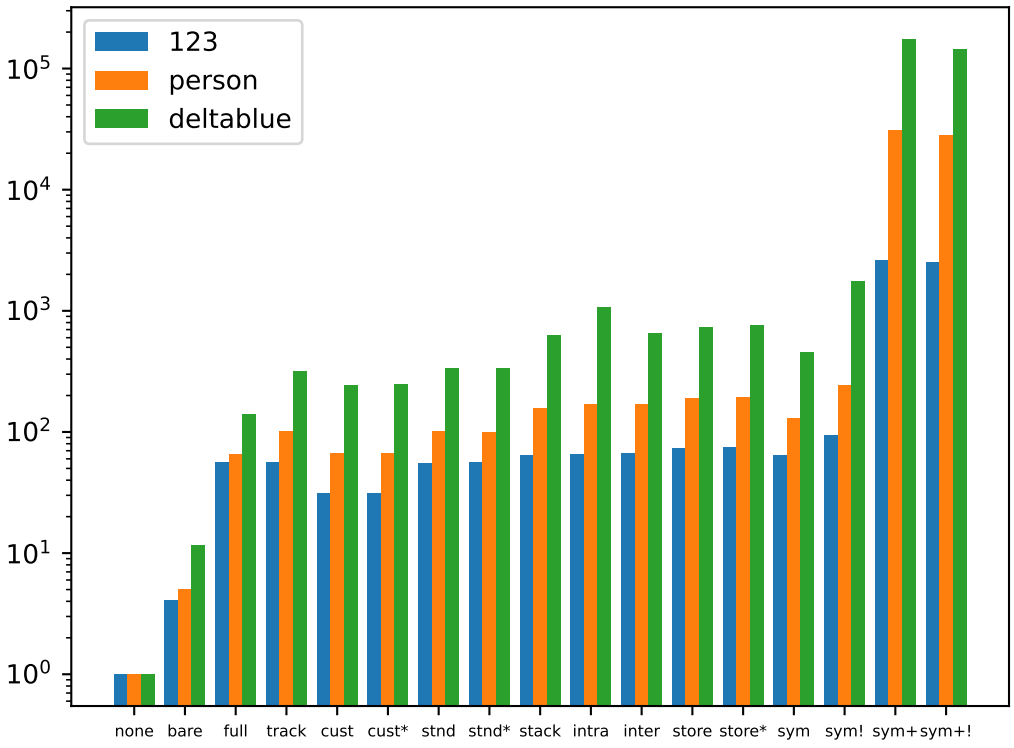


Figure 6.11: Logarithmic plot corresponding to Figure 6.10, including analyses based on advice layering

larger. We are uncertain about the cause of this inconsistency; it could be due to garbage collection or background transient workloads. What is clear is that this spike should not be attributed to the analysis itself. We manually examined the other time measurements and found no other significant measure gaps like this one.

Analysis	Run #1	Run #2	Run #3	Run #4	Run #5
`intra`	15757.716	15340.970	19426.892	40096.850	47840.941
`inter`	15713.980	15185.089	15026.086	17426.420	14916.777

Table 6.8: The time required to instrument `deltablue` under the `intra` and `inter` analyses for each of the five requested repetitions

Order-of-magnitude assessment The data in Table 6.7 indicate that simple Aran normalization incurs a performance overhead of about 10×, while the other analyses incur overheads ranging from 50× to 700×. The exceptions are `sym+` and `sym+!`, the only two analyses based on advice layering, incurring approximately 100,000× overhead. We discuss these three points separately below:

- The observed performance overhead due to Aran normalization alone is significantly lower than that observed in our Octane experiments. This surprising finding may be explained by the fact that Octane is a popular benchmark, and V8 could have been fine-tuned for optimal performance in it. Future work is required to verify this hypothesis.
- While the complexity of the analysis has a noticeable effect on performance, the effect appears to be linear, and no analyses, except for the one implemented via advice layering, particularly stand out from the others.
- In contrast, the double instrumentation prescribed by advice layering incurs a stark performance overhead compared to all other analyses. Although advice layering is an interesting idea for separating concerns, future work is needed to investigate its applicability. This experiment suggests that it may only be suitable for target programs that execute extremely quickly, such as unit tests.

Analysis complexity compounds with baseline complexity One of the more interesting findings is that the slowdown factor is positively correlated with the baseline execution time of the target program. In other words, the analysis overhead appears to increase more than linearly with the baseline execution time. This is evidenced by Figure 6.10, where the `deltablue` data points dominate the `person` data points, which in turn dominates the `123` data points. This finding is consistent with our observation that the overheads from the Test262 experiment are considerably lower than those from the Octane experiment. Future work is needed to understand this compounding effect.

Notable performance improvement for custom weaving We observe that the `cust` analyses based on custom Linvail weaving are noticeably faster than the `stnd` analyses based on standard Aran weaving; on average, they are 45% faster. In contrast, our second Octane experiment yields only a 20% increase in performance for Linvail custom weaving. This difference in performance gain may be attributed to the larger average size of the target program in the meta-Aran experiment compared to the Octane experiment. Consequently, the reduction in code bloat provided by custom weaving could have a more significant impact. Future work is required to verify this hypothesis.

Cost of tracing analysis data The analyses that write data to the file system include the five analyses computing our provenance metric, along with two symbolic execution analyses: `sym!` and `sym+!`. All of these analyses utilize a buffer of 1024 entries to temporarily store data, which reduces system call overhead. Nonetheless, the periodic synchronous flushing of the buffer is expected to result in additional performance overhead. We believe this is the reason why, despite being simpler, the analyses for computing our provenance metric incur greater performance overhead than `sym`, which does not actually record anything. The surprising result here is that `sym+!` is actually faster than `sym+`. It could be that V8 uses periodic synchronous system calls to perform maintenance operations, such as garbage collection,

and that performing these operations more frequently actually improves overall performance. Further investigation is required to verify this hypothesis.

6.5.3 Provenance scores observed during the meta-Aran experiment

In this section, we discuss the provenance scores observed during the experiment for values used to branch the control flow.

The meta-Aran experiment features five analyses with increasingly larger internal regions that record the provenance score of values appearing at the `test@before` join point. As a reminder, this join point arises in explicit branching structures such as `if` statements, and also in less obvious code locations such as logical "and" expressions (`&&`) due to Aran normalization. In what follows, we focus on the `deltablue` instrumentation case, as it offers a larger sample size; however, our conclusions hold for the other two simpler instrumentation cases.

Figure 6.12 presents a statistical summary of the provenance scores recorded at the `test@before` join points during the analysis of the `deltablue` instrumentation across all five participating analyses. It appears that the mean increases significantly with the size of the region capable of tracking primitive values; however, the variation is concentrated in the last percentile. Another finding is that internalizing the global scope does not improve provenance scores, as evidenced by the identical means of both analyses. This can be explained by the absence of global variables in the target program.

To investigate the concentration of score increases in the last percentile, we categorize the recorded scores according to the type of ESTree node from which the `test@before` join point originates. Table 6.9 presents this data for the `store` analysis as it is applied to the `deltablue` instrumentation case. It appears that the original node type has a large impact on recorded scores.

Type	Count	Mean	P25	P50	P75	P95
SwitchCase	8251039 [57%]	9505	1	1	8	8
Identifier	2030510 [14%]	9	3	3	3	24
ForStatement	1482552 [10%]	23	1	1	15	63
IfStatement	1023257 [7%]	18851	4	8	19	90128
LogicalExpression	829751 [6%]	14784	8	8	8	53256
FunctionExpression	204904 [1%]	1	0	0	0	7
ObjectPattern	202867 [1%]	4	4	4	4	4
AssignmentExpression	162688 [1%]	11643	4	4	7	94924
MemberExpression	132862 [1%]	150e6	4	8119	254e6	743e6
ConditionalExpression	57134 [0%]	95702	8	9	14478	250419
FunctionDeclaration	51628 [0%]	0	0	0	0	0
WhileStatement	22097 [0%]	56572	18363	55633	90655	119630
Property	5194 [0%]	8	8	8	8	8
ArrayPattern	2433 [0%]	4	3	3	7	7
ForOfStatement	1211 [0%]	4	1	3	7	7
SpreadElement	334 [0%]	7	7	7	7	7
ForInStatement	39 [0%]	44	23	43	63	79
ClassBody	23 [0%]	4	3	4	4	4
ClassExpression	14 [0%]	3	2	4	4	4
MethodDefinition	6 [0%]	3	3	3	3	3
UnaryExpression	2 [0%]	3	3	3	3	3

Table 6.9: Provenance scores recorded by the `store` analysis during the instrumentation of `deltablue` at `test@before` join points

For instance, Figure 6.13 provides a statistical summary of the provenance score recorded by all five participating analyses at `test@before` join points originating from `ConditionalExpression`. This distribution differs significantly from that depicted in Figure 6.12, which includes all origins. It appears that the top quartile increases noticeably when the tracking of primitives is extended to the environment and

Analysis	Mean	P25	P50	P75	P95	P99
stack	2	0	1	4	6	7
intra	6	1	18	7	7	8
inter	536	1	15	7	7	8
store	1391430	1	16	7	8	8
store*	1391430	1	16	7	8	8

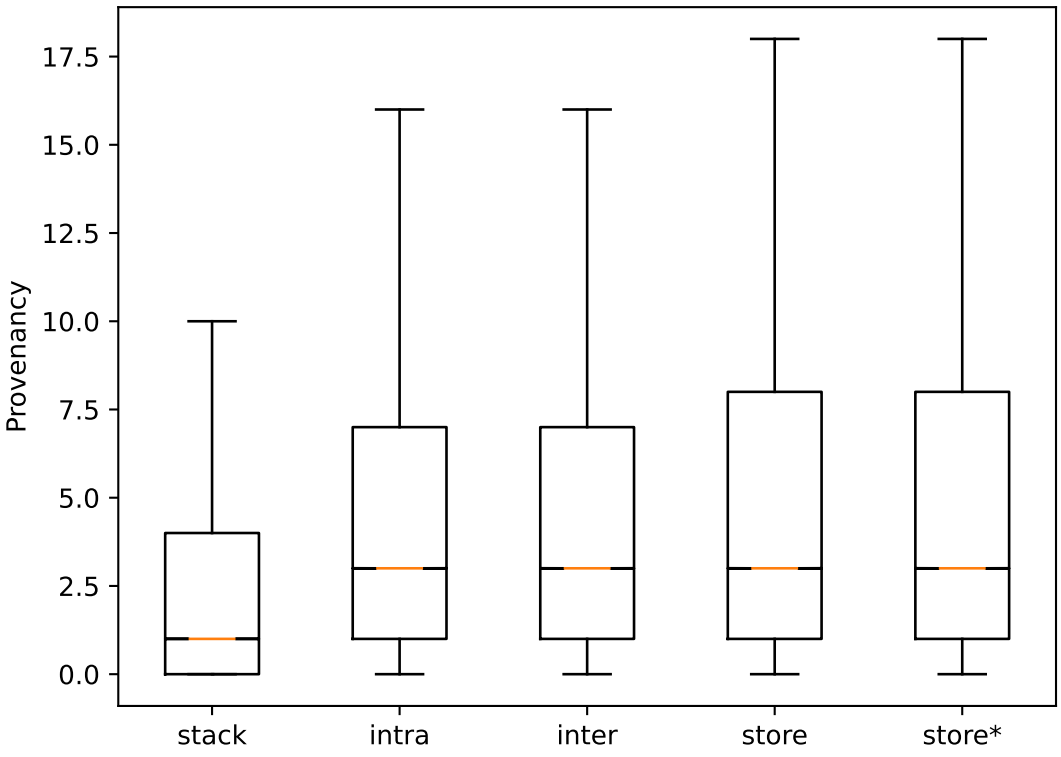


Figure 6.12: Statistical summary of the provenance scores recorded by all five participating analyses at the `test@before` join point during the instrumentation of `deltablue`

then across function calls; however, the sharpest increase occurs when Linvail extends the tracking of primitives to parts of the store.

Analysis	Mean	P25	P50	P75	P95	P99
stack	4	2	4	4	7	7
intra	6	5	5	7	7	10
inter	9	7	9	12	14	19
store	95702	8	9	14478	250419	1547687
store*	95702	8	9	14478	250419	1547687

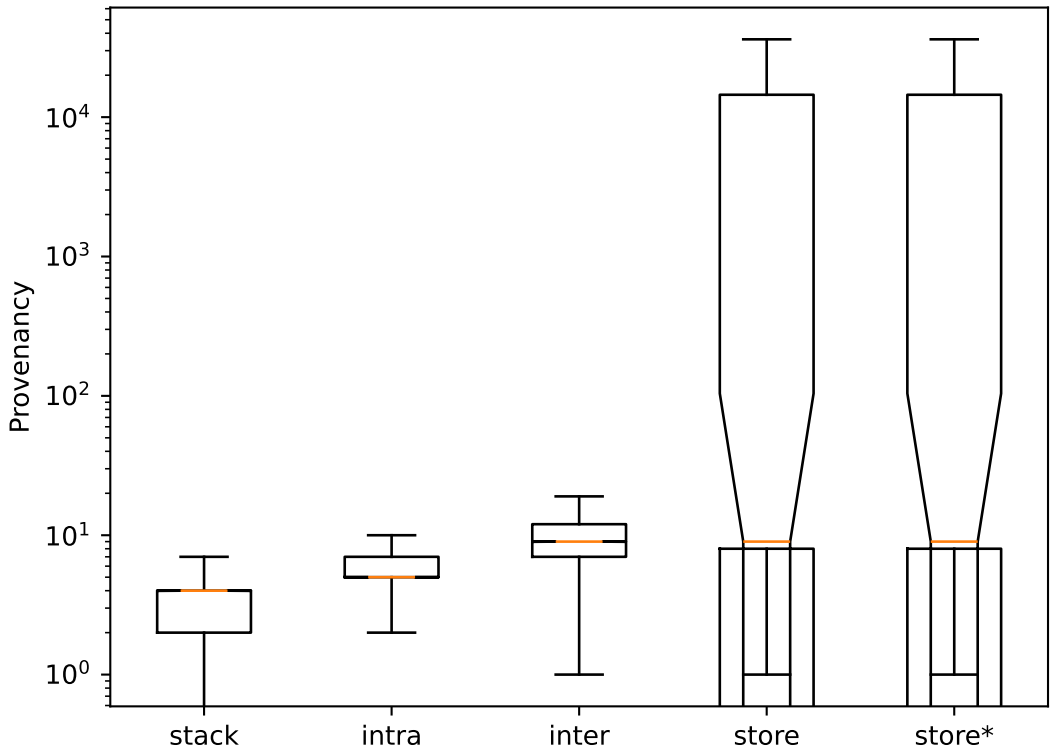


Figure 6.13: Statistical summary of the provenance scores recorded by all five participating analyses at `test@before` join points originating from `ConditionalExpression`

Upon manual inspection, we found that a major reason for this sharp increase is that Acorn indexes the current parsing position on a stateful object, as exemplified by Listing 6.24. Every time the position is modified, its score also increases. For instance, in Listing 6.24, the score of `this.pos` increased by four after the increment operation: one point for the function call (`%aran.performBinaryOperation%`), one point for the `this` argument (`%undefined%`), one point for the operator literal (`"+"`), and one point for the increment literal (`2`). As the index score increases, so does the score of the character retrieved by looking up the code at that position. Manual inspection reveals that this is also the reason why the scores originating from `MemberExpression` nodes are extremely high.

```
1 // @ts-nocheck
2 // https://github.com/acornjs/acorn/blob/0d046aa/acorn/src/tokenize.js#L117
3 let ch = this.input.charCodeAt(this.pos += startSkip);
```

Listing 6.24: Excerpt from Acorn's tokenizer

To summarize, our meta-Aran experiment demonstrates that our provenance metric is correlated with the size of the region capable of tracking primitive values, which is a positive outcome. However, it shows a bias toward index values, as their scores increase with each increment operation. Furthermore,

the score of index values can propagate to non-index values, as observed in Acorn with the characters retrieved from the current position. Future work is needed to explore methods for mitigating the scale of the provenance score associated with index values.

6.6 Conclusion

In this chapter, we presented Linvail, an implementation of the membrane design from the previous chapter that targets JavaScript programs. First, we demonstrated its API and highlighted several points of interest in the implementation. We then proceeded to qualitatively evaluate its expressiveness by presenting three representative analyses built on top of Linvail: an analysis for gathering symbolic constraints, an implementation of virtual values, and a dynamic taint analysis. Finally, we quantitatively evaluated the transparency and precision of Linvail by applying it to Test262, Octane, and Aran itself. To conclude, we summarize results and limitations, outline contributions, and review related work.

Assessment and opportunities for improvement

In Section 6.1.4, we highlighted that the main limitation of Linvail is its inability to track the provenance of values assigned to exotic objects. This limitation arises because the hidden properties of exotic objects cannot be intercepted using the standard virtualization API of ECMAScript (i.e., the `Proxy` API). We have explored alternative tracking mechanisms; however, they are hindered by the two-body problem and synchronization issues. We argue that a complete solution can only be achieved through a mechanism capable of virtualizing the hidden fields of exotic objects. In this context, it would be worthwhile to investigate the engineering efforts required to implement such a change within mainstream runtimes.

In Section 6.3.2, we discussed the semantic discrepancies introduced by Linvail. Although these discrepancies had a very limited incidence rate on Test262 (less than 0.05%) they should be taken seriously, as one of the main strengths of dynamic analysis is precision, which relies on the analysis to be transparent. In the future, it will be worthwhile to explore methods for eliminating these discrepancies.

In Section 6.4.2, we discussed the overhead introduced by Linvail. By maintaining its membrane, Linvail incurs a multiplicative overhead factor of 10× on top of the already significant 100× overhead introduced by Aran. The performance impact of analyses that extend Linvail's advice is controlled, resulting in a performance impact of less than 10×. Consequently, we believe that these analyses have a similar application range to those built solely on Aran. In contrast, analyses developed via advice layering experience an additional multiplicative performance slowdown factor of 100×, resulting in a total slowdown factor of approximately 100,000×. Consequently, these analyses should target programs that execute extremely quickly, such as unit tests.

In Section 6.5.3, we evaluated the impact of progressively broadening frontier designs on the provenance score measured at key points during execution. While the effect on tracking provenance through the store was limited in the second Test262 experiment, it was significantly more pronounced in the meta-Aran experiment, which more accurately reflects realistic program executions. This observation has strengthened our confidence that our provenance metric is a valid measure of the precision of provenance tracking. However, further work is needed to establish this metric as a benchmarking standard.

Main contributions

The main contribution of this chapter is to demonstrate the feasibility of the three propositions from Chapter 5 for separating provenance concerns from analysis-specific logic:

- Overall, the use of value promotion combined with value virtualization to establish a membrane is a clear win, as it introduces almost no semantic discrepancies and incurs a multiplicative performance overhead of 10×, which is acceptable.

- Our assessment of stacking instrumentation to achieve advice layering is more nuanced. While we demonstrated that it simplifies the implementation of analyses, it incurs a steep multiplicative performance overhead of 100×, which could render it impractical for many scenarios.
- Our provenance metric is straightforward to compute and correlates well with the extent of provenance tracking. However, its utility ultimately depends on the degree to which it is adopted by the research community.

Related work

Shadow values (i.e., the capability of attaching metadata to base values) are a key component of provenance-aware dynamic program analyses, such as dynamic taint analysis [84, 25] and dynamic symbolic execution [103, 44]. Traditionally, shadow values have been implemented via shadow execution, which involves mirroring the global execution state. In contrast, our approach is based on value promotion and mirrors each value separately. It alleviates the challenge of synchronizing shadow memory when control returns from arbitrary code. But, it requires virtualization capabilities provided by the target language. Next, we discuss frameworks capable of performing provenance-aware dynamic program analyses, all of which are based on shadow execution.

Jalangi We already discussed Jalangi [105] in Chapter 4 as a pre-Harmony JavaScript instrumentation framework. Jalangi also offers support for implementing shadow values via shadow execution. Behind the scenes, Jalangi implements an analysis similar to our shadow execution shown in Listing 4.8. However, it exposes an interface that shields analysis implementers from some of the complexity of mirroring the stack, the environment, and the store. Later, when the record-and-replay feature was abandoned, this interface evolved into a set of helper functions. These helper functions provide an interface that can be compared to that of our approach in the advice extension architecture.

Dynamic taint analysis A primary application of shadow execution is *dynamic taint analysis*, which enforces security policies at run-time by propagating security-related labels alongside run-time values. Given the significance of security in JavaScript programs, several taint analysis approaches have been proposed [132, 115, 58]. Most of these methods depend on a modified runtime to perform the analysis. While run-time instrumentation is appealing for experimental purposes, it faces portability and maintenance challenges that preclude widespread adoption. In contrast, our tool utilizes source code instrumentation, making it deployable on any JavaScript runtime. To the best of our knowledge, only Ichnaea [58] employs source code instrumentation. There are two major distinctions between their work and ours. First, they offer an API that constrains users to a specific shadow execution model optimized for taint analysis, whereas we provide a flexible aspect-oriented API. Second, similar to Jalangi, Ichnaea only supports pre-Harmony JavaScript.

Dynamic symbolic execution Another prominent application of shadow execution is *dynamic symbolic execution*, which attempts to provide inputs to a program that will lead it to a specific execution path by labeling run-time values with symbols. It is often used to generate test inputs, an approach known as *concolic testing* [42, 103]. Multiple dynamic symbolic execution frameworks have been proposed for JavaScript [100, 71, 99]. Similar to dynamic taint analysis, these approaches often rely on a modified runtime, whereas we explored source code instrumentation for applicability reasons. Aside from our work on inter-process symbolic execution [128], the only other symbolic-execution framework based on source code instrumentation that we are aware of is Kudzu [100], which relies on Jalangi instrumentation and inherits its limitations.

Low-level shadow execution There is also an interesting body of work on performing shadow execution on the lower-level representation of programs. For instance, Valgrind [83] is a popular framework for instrumenting binaries for the purpose of dynamic analysis and is fully capable of carrying out shadow execution. However, because JavaScript is primarily interpreted, and although it is sometimes JIT-compiled, it is unclear how this approach could be applied to a wide range of rapidly evolving runtimes.

Chapter 7

Orchestration of dynamic analysis for distributed applications

In this chapter, we present an approach for orchestrating the analysis of distributed applications, a growing trend in recent years due to technologies such as cloud computing, microservices, and the Internet of Things. Distributed programming is particularly relevant for JavaScript, given its widespread use in client-server applications on both the client side in browsers and the server side with runtimes like Node.js. Our orchestration approach involves executing analysis-specific logic in a separate process, effectively shielding the analysis implementer from the complexities of distribution. However, this introduces significant performance overhead due to extensive synchronous communication.

- *Section 7.1 motivates the need for an approach to handle the complexities of analyzing distributed applications.*
- *Section 7.2 provides an overview of our approach.*
- *Section 7.3 offers insights into the implementation of our approach and describes a protocol for calling remote procedures in a synchronous yet non-blocking manner.*
- *Section 7.4 evaluates our approach by symbolically executing a collaborative drawing application.*
- *Section 7.5 concludes the chapter by summarizing results and discussing related work.*

7.1 Motivating example

We start our exposition with a motivating example that illustrates the difficulties of lifting a dynamic analysis for single-process programs to distributed ones. This section is organized as follows:

- *Section 7.1.1 details a simple analysis of single-process applications.*
- *Section 7.1.2 presents a naive attempt to lift the analysis to distributed systems.*
- *Section 7.1.3 discusses the challenges of analyzing distributed systems.*

7.1.1 Invariant checking of a single-process Node.js application

In this section, we present an analysis of single-process Node.js applications that detects when a file is about to be opened twice.

Concurrent writes to files are a common source of bugs. For example, a log of HTTP requests may become corrupted when two server processes fail to concatenate their information atomically. Consider a Node.js application for which the developers implemented a mechanism to prevent concurrent file openings. To test this mechanism, the developers built a dynamic analysis on top of our approach that inserts run-time checks into the application to detect violations of the invariant "a file should never be opened twice".

Listing 7.1 illustrates the implementation of the analysis as a standard Aran advice. Knowing that their application consistently manipulates files by first invoking `node:fs.open`, the developers needed only to intercept function applications through the `apply@around` join point. The advice checks whether the invocation would result in a file being opened twice. If so, an exception is thrown; otherwise, the call is performed reflectively.

```
1  import { open, readdirSync, readlinkSync } from "node:fs";
2  const listOpenFiles = () => readdirSync("/proc/self/fd")
3    .map((fd) => readlinkSync(`/proc/self/fd/${fd}`));
4  export const advice = {
5    "apply@around": (_state, callee, that, input, _location) => {
6      if (callee === open && input.length > 0)
7        if (listOpenFiles().includes(input[0]))
8          throw new Error("File already opened");
9      return Reflect.apply(callee, that, input);
10   }
11 };
```

Listing 7.1: Single-process analysis to prevent files from being opened twice

7.1.2 Naive invariant checking of a distributed Node.js application

In this section, we attempt to lift the analysis from the previous section to make it target distributed Node.js programs.

To enhance performance, the developers of our hypothetical application opted to distribute it across multiple Node.js processes. The invariant "a file should never be opened twice" must now be enforced across all application processes. To adapt their analysis for this distributed architecture, the developers have little choice but to deploy an analysis instance on each process of the target application and to conduct inter-process communication. They choose to implement this communication according to the client–server model using HTTP requests and WebSocket messaging.

Listing 7.2 illustrates the client side, which consists of an analysis instance deployed on each application process. In contrast to the single-process analysis shown in Listing 7.1.1, this distributed analysis now performs synchronous HTTP requests at Lines 14–18 to the server-side of the analysis to verify that a file has not already been opened by another Node.js process within the application. Additionally, the analysis instances must provide their list of opened files upon request from the analysis server, which they do using the WebSocket protocol at Lines 8–10.

Listing 7.3 depicts the server side of the analysis, which functions as an orchestrator, managing both HTTP requests and WebSocket connections from the analysis instances. Incoming HTTP requests indicate that one of the application's processes intends to open a file. To determine whether this operation is permissible, the server broadcasts at Lines 15–18 a request to all application processes to retrieve their currently opened files, *including the one from which the HTTP request originated.* If the requested file is already opened, the server responds with a status of `403` at Line 11.

```
import { open, readdirSync, readlinkSync } from "node:fs";
import { env } from "node:process";
import { WebSocket } from "ws";
import { XMLHttpRequest } from "xmlhttprequest";
const listOpenFiles = () => readdirSync("/proc/self/fd")
  .map((fd) => readlinkSync(`/proc/self/fd/${fd}`));
const port = env.ANALYSIS_PORT ?? "8000";
const socket = new WebSocket(`ws://localhost:${port}`);
socket.onmessage = () => { socket.send(listOpenFiles()); };
await new Promise((resolve) => { socket.onopen = resolve; });
export const advice = {
  "apply@around": (_state, callee, that, input, _location) => {
    if (callee === open && input.length > 0) {
      const request = new XMLHttpRequest();
      request.open("GET", `http://localhost:${port}/${input[0]}`, false);
      request.send();
      if (request.status === 403)
        throw new Error("File already opened");
    }
    return Reflect.apply(callee, that, input);
  },
};
```

Listing 7.2: Client of a distributed analysis to prevent files from being opened twice

```
import { createServer } from "node:http";
import { env } from "node:process";
import { WebSocketServer } from "ws";
const wss = WebSocketServer({ noServer: true, clientTracking: true });
const server = createServer();
server.on("upgrade", wss.handleUpgrade.bind(wss));
server.on("request", (request, response) => {
  let counter = wss.clients.length;
  const handler = (paths) => {
    if (paths.includes(request.url))
      response.writeHead(403);
    if (!--counter)
      response.end();
  };
  for (const client of wss.clients) {
    client.send("list-opened-paths");
    client.once("message", handler);
  }
});
server.listen(parseInt(env.ANALYSIS_PORT ?? "8000"));
```

Listing 7.3: Server of a distributed analysis to prevent files from being opened twice

Unfortunately, the communication between Listings 7.2 and 7.3 is susceptible to deadlock. In an event-driven language like JavaScript, synchronous communication is typically implemented in a blocking manner, preventing the virtual machine from processing any new events in the event queue. This behavior is crucial to satisfy the *run-to-completion*[1] requirement, which states that an event must be fully processed before handling the next one. Consequently, while the client analysis waits for the server's response, the WebSocket's `onmessage` event handler cannot be triggered, preventing the server from responding. Although the client analysis could be modified to first verify its own locally opened files, thereby alleviating the server's need to perform a loopback request, this change would not entirely eliminate deadlocks. Two analysis instances could still issue synchronous requests simultaneously, resulting in a situation where both remain blocked indefinitely.

7.1.3 Problem statement

In this section, we outline the main difficulties in developing analyses for distributed systems.

Figure 7.1 depicts the architecture of our attempt at solving the motivating example. It shows that analyzing a distributed application requires an analysis that is itself distributed in order to reason about the shared state of the application. Given the inherent difficulties of distributed programming, this fact alone suffices to motivate the need for support in adapting single-process analyses to distributed environments.

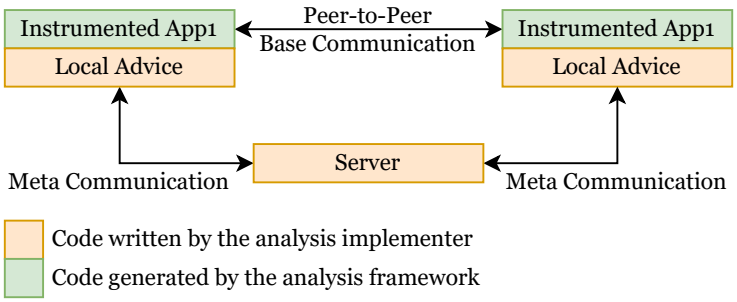


Figure 7.1: Process layout for analyzing distributed applications

The requirement for the analysis to occasionally make synchronous decisions based on the shared state of the application further complicates the implementation of analyses for distributed systems. In event-based systems, asynchronous non-blocking communication is often preferred; however, synchronous blocking communication is necessary when decisions must be made within an advice function. Section 7.3 provides a more detailed discussion of this requirement. Consequently, distributed analyses must often blend asynchronous and synchronous communication, which is error-prone and further complicates their development.

7.2 Overview of the approach

In this section, we present an overview of our approach to analyzing distributed systems. This section is organized as follows:

- *Section 7.2.1 discusses the design decisions of our approach.*
- *Section 7.2.2 demonstrates how our approach addresses the issues mentioned in our motivating example by revisiting it.*

[1] `https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop`

7.2.1 Important design decisions

In this section, we outline the two key design decisions of our approach: centralized analysis and remote references.

The primary design decision of our approach is to centralize the analysis logic, implemented by the user, within a single process called the analysis process. This strategy radically addresses the challenges outlined in our problem statement by completely concealing the complexities of distribution from the user. Figure 7.2 illustrates the process layout of our approach. The code written by the analysis implementer is centralized within the analysis process, which interfaces with the orchestrator. Communication is fully managed by our approach, occurring between the central orchestrator and the local advice deployed on each process of the distributed application.

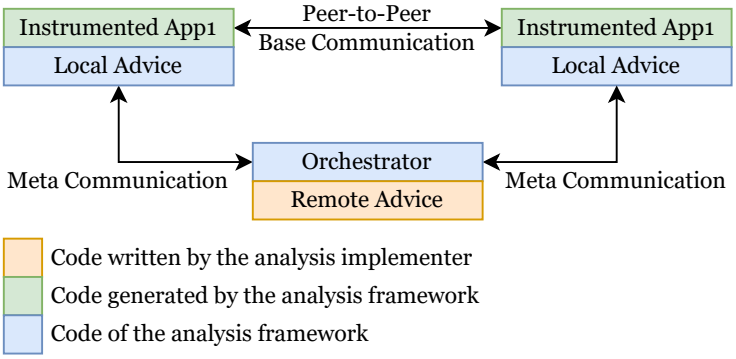


Figure 7.2: Centralized process layout for the analysis of distributed applications

In cases where advice functions must process immutable values, the data represented by these values can be easily shared by copy across the distributed analysis. However, when advice functions manipulate references, the situation becomes more complex. If the advice functions only perform read operations, structured cloning may suffice to abstract away the distribution. However, the analysis may sometimes require mutations, such as when it needs to mark a value as being tainted. In such cases, the approach must provide a means to propagate the mutation from the central analysis process to the target processes and vice versa. Remote references are a common technique to facilitate such communication and can be viewed as placeholder values for resources residing in another process

7.2.2 Revisiting the motivating example

In this section, we demonstrate the interface of our approach in both its asynchronous and synchronous forms.

We now demonstrate our approach by revisiting the motivating example. First, we present our asynchronous API, which is summarized in Listing 7.4 and reads as follows:

- `AsyncAnalysis` (Lines 28–31): An asynchronous analysis exports two asynchronous functions: `connect` and `disconnect`, which are invoked when a new JavaScript process enters or leaves the analysis, respectively. The `connect` method returns an asynchronous advice object that can be extended with additional properties for state management; this object will subsequently be passed to `disconnect` to facilitate cleanup operations. Additionally, an analysis may export `shouldInstrument` to specify which files should be instrumented and `intrinsic_global_variable` to explicitly define the name of the global variable for holding intrinsic values.
- `RemoteReference` (Lines 1–10): An asynchronous remote reference is an object featuring several asynchronous methods that mirror those from `%Reflect%`, along with a custom `invoke` method, which serves as shorthand for chaining `get` followed by `apply`. These methods expect remote values and return promises.

- `RemoteValue` (Lines 12–14): An asynchronous remote value can be either a primitive or an asynchronous remote reference. Symbols require special treatment to preserve their identity across different processes.

- `AsyncAdvice` (Lines 16–26): An asynchronous advice mirrors Aran's standard advice format, except that it expects remote values and returns promises, similar to the methods of asynchronous remote references.

```
1  type RemoteReference = {
2    // Property Lookups:
3    get (key: RemoteValue, receiver: RemoteValue): Promise<RemoteValue>,
4    set (key: RemoteValue, value: RemoteValue, receiver: any): Promise<boolean>,
5    has (key: RemoteValue): Promise<boolean>,
6    // Other methods mirroring the ones from Reflect:
7    // ... (always return a promise)
8    // Additional invoke method for convenience:
9    invoke (that: RemoteValue, input: RemoteValue[]): Promise<RemoteValue>,
10 };
11
12 type Primitive = null | undefined | boolean | string | number | symbol | bigint;
13
14 type RemoteValue = Primitive | RemoteReference;
15
16 type AsyncAdvice = {
17   "apply@around": (
18     state: RemoteValue,
19     callee: RemoteValue,
20     that: RemoteValue,
21     input: RemoteValue[],
22     location: RemoteValue,
23   ) => Promise<RemoteValue>,
24   // Other join points mirroring the ones of Aran's standard advice format:
25   // ... (always return a promise)
26 }
27
28 type AsyncAnalysis<A extends AsyncAdvice> = {
29   connect: (global: RemoteReference) => Promise<A>,
30   disconnect: (advice: A) => Promise<void>,
31   shouldInstrument?: (path: string) => boolean,
32   intrinsic_global_variable?: string,
33 }
```

Listing 7.4: TypeScript definitions of the asynchronous API for remote analysis

Listing 7.5 illustrates a remote analysis built on our asynchronous API. The primary difference from the single-process analysis in Listing 7.1 is that this analysis must manage multiple file system objects. This is accomplished by storing them in a registry of remote references, each representing the file system object of a connected target process. The listing is described as follows:

- `fss` (Line 1): The global variable `fss` is a collection of remote references representing the file system object of each currently connected process.

- `connect` (Lines 2–7): When a new process enters the analysis, the `connect` export is invoked by our infrastructure with an asynchronous remote reference for the global object of the process. We retrieve the remote file system object of this new process and add it to both the global registry and the return value as an additional property of the asynchronous advice.

- `disconnect` (Line 8): When a process leaves the analysis, we remove its remote file system object from the global registry.

- `advice` (Lines 9–24): When a function is called, if it corresponds to the `open` method of the remote file system object, we check for conflicts before forwarding the call. We list all open file descriptors of the connected processes, including the process from which the function application originated. For each file descriptor, we retrieve the path and report any conflicts.

Listing 7.6 provides a potential communication history between the central analysis process and two target processes.

- Line 1: A target process has intercepted a function application that may involve opening a new file.

```
const fss = new Set();
export const connect = async (global) => {
  const mainModule = await (await global.get("process")).get("mainModule");
  const fs = await mainModule.invoke("require", ["fs"]);
  fss.add(fs);
  return { __proto__: advice, _fs: fs };
};
export const disconnect = async ({ _fs: fs }) => { fss.delete(fs); };
const advice = {
  async "apply@around" (_state, callee, that, input, _location) {
    if (callee === (await this._fs.get("open")) && input.length > 0) {
      for (const fs of fss) {
        const descriptors = await fs.invoke("readdirSync", ["/proc/self/fd"]);
        const length = await descriptors.get("length");
        for (let index = 0; index < length; index++) {
          const descriptor = await descriptors.get(index);
          const path = await fs.invoke("readlinkSync", [`/proc/self/fd/${descriptor}`]);
          if (path === input[0]) throw new Error("File already opened");
        }
      }
    }
    return callee.apply(that, input);
  }
};
```

Listing 7.5: Asynchronous remote analysis to prevent files from being opened twice

- Lines 2–3: The analysis process responds to the target process by retrieving the `open` property of its file system object.
- Between Line 3 and 4: Since there is a match between the callee of the initial request and the remote reference for `fs.open`, the analysis process begins querying connected target processes for open files.
- Line 4: It starts by invoking `readdirSync` in the process from which the initial request originated.
- Line 5: This call results in a remote reference representing the array `["0", "1", "2"]`, indicating that the process has only opened its standard input and output descriptors.
- Line 11: As none of these lead to a matching path, the analysis process proceeds to invoke `readdirSync` on another target process.
- Line 12: This call results in a remote reference representing the array `["0", "1", "2", "3"]`, indicating that the process has opened an additional file beyond its standard input and output.
- Lines 14–15: Invoking `readlinkSync` on the last file descriptor of the process yields the path `"f.txt"`.
- Line 16: Since this path matches the first input argument of the initial request, an error is signaled to the originating process, thus preventing `f.txt` from being opened twice.

```
app1 >> apply@around(app1#2, undefined, ["f.txt", "a"])
app1 << get(app1#1, "open")
app1 >> app1#2 // fs.open
app1 << invoke(app1#1, "readdirSync", ["/proc/self/fd"])
app1 >> app1#3 // ["0", "1", "2"]
app1 << get(app1#3, "length")
app1 >> 3
app1 << invoke(app1#1, "readlinkSync", ["/proc/self/fd/0"])
app1 >> "tty-pipe#0"
...
app2 >> invoke(app2#1, "readdirSync", ["/proc/self/fd"])
app2 << app2#3 // ["0", "1", "2", "3"]
...
app2 >> invoke(app2#1, "readlinkSync", ["/proc/self/fd/3"])
app2 << "f.txt"
app1 >> !!! "File already opened"
```

Listing 7.6: Example communication during the analysis from Listing 7.5

There are two key insights that can be derived from this communication example. First, a significant volume of intricate communication can arise from simple analyses, leading to performance overhead, which we discuss in Section 7.4. Second, the target process from which the `apply@around` request originated must remain responsive, even though this request must be processed synchronously due to the synchronous

nature of the `apply@around` advice within the target process. We address this technical challenge in Section 7.3.

While our asynchronous API already shields the analysis implementer from much of the complexity associated with analyzing distributed applications, our synchronous API goes a step further in creating the illusion that the target application is not distributed, even though it actually is. Listing 7.7 illustrates how the asynchronous analysis from Listing 7.5 can be expressed in our synchronous API. There are three main differences:

- Synchronous Analysis (Lines 2–8): The `connect` and `disconnect` exports operate synchronously.
- Synchronous Remote Advice (Lines 9–24): The advice within the analysis process is synchronous and adheres to the same format as standard Aran advice. Advice functions should not return promises and they receive and return synchronous remote references.
- Synchronous Remote Reference (Lines 3, 4, 11, 13, 14, 16, and 17): Manipulating remote references occurs synchronously, necessitating synchronous communication behind the scenes. Furthermore, they are implemented as virtual values via the proxy API, allowing them to be treated like local references.

```
1  const fss = new Set();
2  export const connect = (global) => {
3    const { process: { mainModule } } = global;
4    const fs = mainModule.require("fs");
5    fss.add(fs);
6    return { __proto__: advice, _fs: fs };
7  };
8  export const disconnect = ({ _fs: fs }) => { fss.delete(fs); };
9  const advice = {
10   "apply@around" (_state, callee, that, input, _location) {
11     if (callee === this._fs.open && input.length > 0) {
12       for (const fs of fss) {
13         const descriptors = fs.readdirSync("/proc/self/fd");
14         const { length } = descriptors;
15         for (let index = 0; index < length; index++) {
16           const descriptor = descriptors[index]
17           const path = fs.readlinkSync(`/proc/self/fd/${descriptor}`);
18           if (path === input[0]) throw new Error("File already opened");
19         }
20       }
21     }
22     return callee.apply(that, input);
23   }
24 };
```

Listing 7.7: Synchronous remote analysis to prevent files from being opened twice

We conclude this section by comparing the merits of the two APIs. The synchronous version of the distributed analysis is not affected by asynchronous noise and resembles the analysis of single-process programs from which we started. However, the synchronous API conceals the distribution so effectively that the analysis developer may forget that some values are remote references and that performing operations from the meta object protocol on them is costly. In contrast, our asynchronous API offers greater control over analysis performance at the expense of slightly more complex lifting. Regardless, both APIs protect the analysis developer from concerns related to distributed programming at the cost of a high communication load.

7.3 Implementation of the Approach

In this section, we provide insights into the implementation of orchestrating the analysis of distributed applications by centralizing the analysis logic in a separate process. This section is organized as follows:

- *Section 7.3.1 explains the necessity of our approach to communicate in a synchronous but non-blocking manner.*
- *Section 7.3.2 details our synchronous non-blocking communication protocol for remote procedure calls.*
- *Section 7.3.3 discusses remote references.*
- *Section 7.3.4 outlines the architecture of our prototype.*

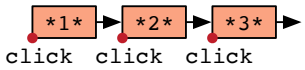
7.3.1 Limitations of existing communication protocols

In this section, we explain why existing communication protocols cannot support the communication requirements of our approach out-of-the-box.

JavaScript is an event-based synchronous language, meaning that events are processed one *after* the other, and during the processing of an event, expressions are evaluated sequentially. Therefore, synchronous I/O operations are discouraged, as they require pausing the entire JavaScript application. Instead, I/O operations should be performed asynchronously, either by passing a callback argument or by returning a promise value. Specifically, within browser runtimes, the only remaining synchronous I/O operations are executed by the `XMLHttpRequest` API, which are discouraged on the main thread due to their potential to make the UI unresponsive. In this section, we provide a compelling use case to justify the non-legacy use of this synchronous communication mechanism.

To motivate our use of synchronous communication, we propose to study the instrumentations of Listing 7.8 in the context of a remote central analysis as proposed by our approach. The program is a simple "counter" button based on the document object model (DOM): every time the button is clicked, the `counter` variable is incremented. An analysis to verify that the counter is incrementing as expected can be constructed by advising the increment operation. In our approach, this requires the instrumented program to communicate with the analysis process.

```
1 const button = document.createElement("button");
2 document.body.appendChild(button);
3 let counter = 0;
4 button.onclick = () => {
5   counter = counter + 1;
6   alert(counter);
7 };
```

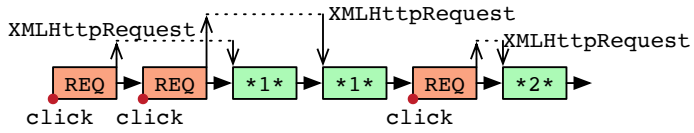


Listing 7.8: Simple DOM-based counter button

First, we instrument Listing 7.8 under the assumption that communication is performed asynchronously, as is generally preferred. To achieve this communication, we rely on the asynchronous call to `XMLHttpRequest`; however, the same discussion applies to other asynchronous technologies such as `WebSocket`, which provide bidirectional communication channels. Unfortunately, this instrumentation is flawed. If the user triggers the button in quick succession, the increment operation may be executed on stale counter values. This occurs because two asynchronous requests may have been initiated concurrently. Consequently, the instrumentation fails to preserve the program's original behavior, violating the run-to-completion principle and leaving the counter in an inconsistent state.

Listing 7.10 examines the synchronous alternative, where the event handler blocks until a request is completed. This approach addresses the inconsistencies found in Listing 7.9; however, it is susceptible to

```
const button = document.createElement("button");
document.body.appendChild(button);
let counter = 0;
button.onclick = () => {
  const REQ = new XMLHttpRequest();
  REQ.open("POST", "/aran/binary", false);
  REQ.send("['+","+counter+",1]");
  REQ.onload = () => {
    counter = parseInt(REQ.responseText);
    alert(counter);
  };
};
```



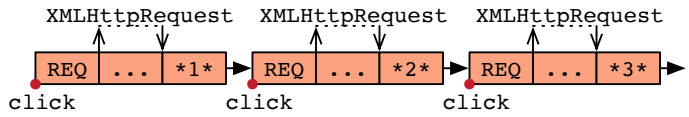
Listing 7.9: Asynchronous instrumentation of Listing 7.8

deadlocks, as discussed in Section 7.1. Specifically, when the target process awaits a response, it becomes unresponsive and is unable to handle potential loopback requests from the analysis process.

```
const button = document.createElement("button");
document.body.appendChild(button);
let counter = 0;
button.onclick = () => {
  const REQ = new XMLHttpRequest();
  REQ.open("POST", "/aran/binary", true);
  REQ.send("['+","+counter+",1]");

  counter = parseInt(REQ.responseText);
  alert(counter);

};
```



Listing 7.10: Synchronous instrumentation of Listing 7.8

To summarize, our approach cannot rely on asynchronous communication without violating the run-to-completion principle of JavaScript and compromising semantic transparency. However, blocking synchronous communication can lead to deadlocks in the event of loopback requests from the analysis process. These observations highlight the need for a new communication protocol.

7.3.2 Synchronous non-blocking remote procedure calls

In this section, we present a novel protocol for implementing synchronous remote procedure calls in a non-blocking manner.

We now present the protocol for performing synchronous yet responsive remote procedure calls that manage analysis-related communication within our approach. Our protocol follows a classic client-server model, where both the central analysis process and the target processes act as clients, while the server operates in a separate process called the orchestrator process. This protocol relies on three communication channels that can be easily implemented using existing technologies:

- An immediate push notification from the client to the server. If the client resides in the browser, this channel can be implemented using standard HTTP requests or WebSocket. If the client is located elsewhere, other TCP-based technologies can be utilized, provided they maintain message

boundaries and deliver messages immediately. This last requirement is crucial, as waiting for the current event to be processed before sending the notification may lead to deadlock.

- A ping notification from the server to the client. If the client is in the browser, this channel can be implemented with server-sent events (SSE) or WebSocket. If the client is located elsewhere, other TCP-based approaches can be employed without specific requirements.
- A synchronous pull request from the client to the server. If the client resides in the browser, this channel can only be implemented via synchronous XMLHttpRequest. If the client is located elsewhere, other TCP-based methods can be used, provided they implement a synchronous request/response mechanism, which is often discouraged in event-based systems. For example, if the client is in a Node.js process, this channel can be implemented with synchronous functions from `child_process` and `fs`. However, in our implementation, we rely on C++ bindings to enhance performance.

Within our protocol, information is passed as messages that correspond to either a remote procedure call (RPC) or a remote procedure return (RPR). In both cases, messages contain three fields:

- `origin`: If the message is an RPC, this field contains the alias of the client from which the request originated. If the message is a RPR, this field is left empty—i.e., it is `null` in the implementation.
- `token`: A number generated by the client that originated the RPC/RPR pair, enabling the association of the response with the corresponding request.
- `data`: The payload of the message, which corresponds to either the input or the output of a remote procedure call.

Listing 7.11 describes the server-side of our protocol, implemented as a JavaScript module. Below, we detail the implementation by explaining how the four server-side events of our protocol are handled:

- `connect`: A new client connects; its connection is represented by a mailbox, which is a FIFO queue of messages, and an optional callback indicating that the client is currently pulling a message.
- `disconnect`: A client disconnects; its representation should be removed from the set of active connections.
- `pull`: A client pulls a message. If the mailbox is empty, the callback is stored, indicating that the client is waiting for a new message to arrive; otherwise, the oldest message from the mailbox is provided.
- `push`: A client pushes a message to a recipient. If the recipient is already waiting for a message, the new message is provided directly; otherwise, the message is stored in the recipient's mailbox. Regardless, a ping notification is sent to the recipient if the message is an RPC.

To enhance understanding of our protocol, we modeled a single client connection within the server-side protocol as a pushdown automaton (PDA), as shown in Figure 7.3. The stack of the automaton simulates the mailbox of the client connection and contains only two types of elements: the initial symbol `Z` and a marker symbol `X`. The automaton features three states:

- `flow`: The client is not pulling; this is the initial state. If the client sends a pull request, we either transition to the `pull` state if the mailbox is empty, or remain in the `flow` state and provide the oldest message in the mailbox if it is not empty. If the client receives a push notification, we either transiently move to the `ping` state if it is an RPC, or remain in the `flow` state if it is an RPR.
- `pull`: The client is waiting for a new message to arrive. If it is an RPC, we transiently move to the `ping` state; if it is an RPR, we directly return to the `flow` state.
- `ping`: A simple transient state indicating that push notifications corresponding to an RPC should result in a ping to the client.

Listing 7.12 describes the client-side of our protocol, implemented as a JavaScript module. The state of the clients consists of two components. The first component is `counter`, a positive integer representing the number of RPC messages that have been pulled from the server without a ping. The second component is

```
import { listen } from "push-pull-ping";
const connections = new Map();
const { ping, close } = listen({
  host: "localhost:8080",
  onconnect: (alias) => {
    if (connections.has(alias))
      throw new Error("Duplicate connection");
    connections.set(alias, { mailbox: [], pending: null });
  },
  ondisconnect: (alias) => {
    if (!connections.has(alias))
      throw new Error("Missing connection");
    connections.delete(alias);
  },
  onpull: (origin, callback) => {
    if (!connections.has(origin))
      throw new Error("Missing connection");
    const connection = connections.get(origin);
    if (connection.pending !== null)
      throw new Error("Already pulling");
    if (connection.mailbox.length === 0) {
      connection.pending = callback;
    } else {
      callback(connection.mailbox.pop());
    }
  },
  onpush: (recipient, message) => {
    if (!connections.has(recipient))
      throw new Error("Missing connection");
    if ("origin" in message)
      ping(recipient);
    const connection = connections.get(recipient);
    if (connection.pending === null) {
      connection.mailbox.unshift(message);
    } else {
      connection.pending(message);
      connection.pending = null;
    }
  },
});
```

Listing 7.11: Server-side for synchronous non-blocking remote procedure calls

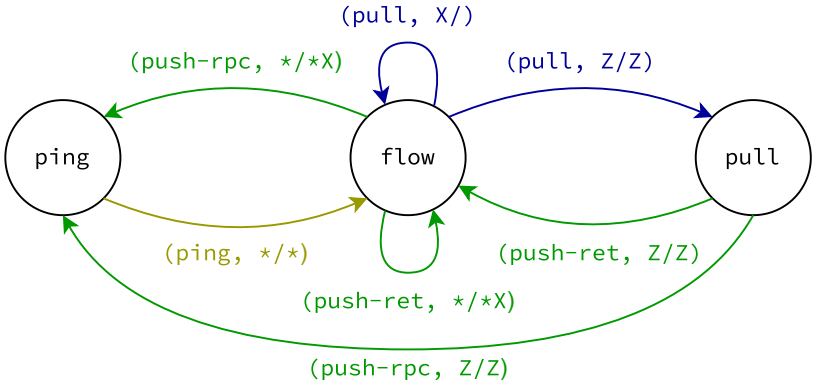


Figure 7.3: Pushdown automaton for a single client connection

`stack`, which is a stack of optional RPR messages representing the number of concurrent RPC messages sent from the client. Unlike a local call stack, this remote call stack supports out-of-order resolution. This state enables proper handling of the two main events for the client:

- `ping`: Ping events originate from the server to indicate that an RPC was available at some point in the past but might have already been pulled. These events are considered root events, meaning they cannot occur in the middle of processing another event. There are two cases to consider: If the counter is greater than zero, the corresponding RPC message has already been pulled from the server, and the only action is to decrement the counter. If the counter is zero, the RPC message still resides on the server and can be pulled.
- `call`: This event corresponds to an invocation of `rpcall`, which can occur while the client is idle or within the `handle` procedure if the client is processing other remote procedure calls. When this event occurs, an RPC message is first sent to the server with a token representing the position in the stack where the associated RPR message should be stored. While the stack remains empty at the marked position, messages are pulled from the server. If a RPR message is received, it is handled normally, which may trigger a concurrent `call` event. If a RPC message is received, the stack is updated accordingly.

```
import { connect } from "push-pull-ping";
let counter = 0;
const stack = [];
const alias = "my-alias";
const handle = (input) => { /* the main client's procedure (can use rpcall) */ };
const { pull, push, disconnet } = connect({
  host: "localhost:8080",
  alias,
  onping: () => {
    if (stack.length > 0)
      throw new Error("ping event while stack is not empty");
    if (counter > 0) {
      counter--;
    } else {
      const { origin, token, data } = pull();
      if (origin === null)
        throw new Error("pulled a response on an empty stack");
      push(origin, { origin: null, token, data: handle(data) });
    }
  },
});
export const rpcall = (recipient, data) => {
  const token = stack.length;
  stack.push(null);
  push(recipient, { origin: alias, token, data });
  while (stack[token] === null) {
    const { origin, token, data } = pull();
    if (origin) {
      counter++;
      push(origin, { token, data: handle(data) });
    } else {
      stack[token] = data;
    }
  }
  if (stack.length !== token + 1)
    throw new Error("This should never happen");
  return stack.pop().data;
};
```

Listing 7.12: Client-side for synchronous non-blocking remote procedure calls

As for the server-side of our protocol, we have also modeled the client-side as a pushdown automaton (PDA), which is depicted in Figure 7.4. This automaton has two stacks: the first stack (`stack` in the code) models the size of the client's remote call stack, while the second stack (`counter` in the code) represents the number of remote procedure calls retrieved from the server without a ping. Similar to our server-side PDA, both stacks contain two types of elements: the initial symbol `Z` and a marker symbol `X`. We present the automaton by detailing three workflows:

- Sequential Response (`idle` → `pull` → `handle`): 1. Initially, the client is idle and receives a ping from the server while its ping counter stack is empty; 2. the client pulls a message from the server, expecting it to be an RPC message; 3. the client passes the input data to the `handle` procedure, which directly computes the output; 4. the output is sent back to the origin inside an RPR message; 5. the client loops back to being idle.

- Sequential Request (`idle` → `send` → `pull`): 1. Initially, the client is idle and is requested to conduct a remote call; 2. the input data is sent to the recipient inside an RPC message; 3. the corresponding response is pulled from the server, and the data is returned as the result of the remote call; 4. the client loops back to being idle.
- Concurrent Request/Response: Both previous sequential workflows are essentially stateless; only the handling of concurrent RPC/RPR messages requires the ping counter stack and the RPR message stack. Our model indicates that concurrency arises when a `call` event occurs from a `handle` state, corresponding to the situation where the `handle` procedure calls `rpcall`.

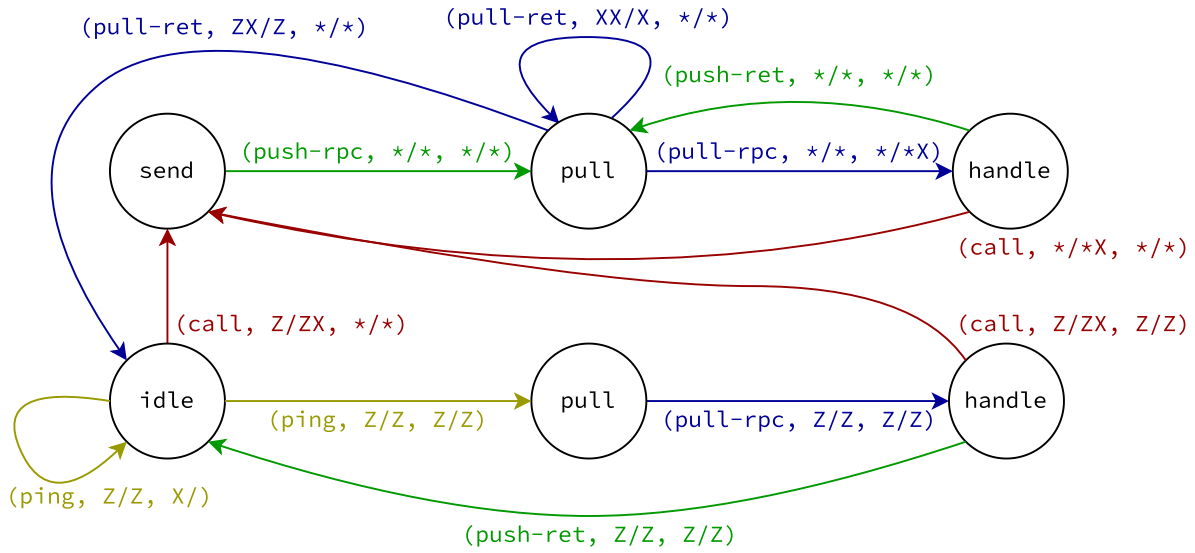


Figure 7.4: Two-stack pushdown automaton modeling the client side of our protocol

7.3.3 Synchronous responsive remote references

> *In this section, we motivate the use of synchronous remote references and discuss their implementation.*

In our approach, the central advice within the analysis process receives remote references that represent objects residing within the target processes. To accommodate asynchronous remote references, our asynchronous API had to adapt Aran's standard advice format and make its functions asynchronous as well. This necessary propagation of asynchronicity across the call stack is sometimes referred to as "async contagion". In contrast, synchronous remote references are isomorphic to objects and do not require modifications to the advice format.

As a result, we can conclude that synchronous remote references serve primarily as a convenience when they are located within the analysis process. We will now examine the scenario in which remote references reside within target processes. This situation may arise, for example, when the analysis wraps a run-time value into an object that exists within the analysis process. In such cases, asynchronous remote references are ill-suited due to the complex instrumentation required to manage async contagion. Consequently, isomorphic remote references are not merely a convenience within target processes; they are an absolute necessity.

We now explain how remote references can be made isomorphic to local references. First, they must be based on a synchronous communication protocol to prevent async contagion. However, similar to advice-related communication, the protocol should be non-blocking to avoid deadlock, which is why synchronous references leverage our protocol as well. Second, remote references must behave like regular objects, which necessitates value virtualization. In JavaScript, this is achieved through the `Proxy` mechanism [123]. Since the object for which the remote reference stands resides in another process, we encounter similar issues as described in Section 6.1.5. Namely, the target object is used to enforce invariants, yet there is no suitable

direct candidate for it. To address this issue, we reuse our `VirtualProxy` API, which offers the following advantages:

- It automatically synchronizes the target object with the proxy to avoid spurious signaling of invariant violations.
- It does not invoke handlers when their results can be deduced from inspecting the target object, thereby reducing the communication load associated with the use of remote references.
- It eliminates the need to provide a handler for derived MOP operations (i.e., `get`, `set`, and `has`) which simplifies implementation.

7.3.4 AranRemote: implementation of our approach to centralize analysis

In this section, we provide a brief overview of the implementation of our prototype for analyzing distributed systems in a centralized manner.

Our approach has been realized in a prototype named AranRemote, which is available on GitHub. Figure 7.5 depicts a process view of the platform after initialization and instrumentation while it is actively conducting the analysis. Below, we explain the architecture of our prototype by detailing its components:

- Otiluke – `https://github.com/lachrist/otiluke`: A platform for developing and deploying JavaScript code instrumenters, which are themselves written in JavaScript. Otiluke provides a uniform interface for deploying instrumenters across both browser and Node.js environments.
- Aran – `https://github.com/lachrist/aran`: Our JavaScript code instrumenter that is compatible with Otiluke for automated instrumentation of modular JavaScript applications.
- Linvail – `https://github.com/lachrist/linvail`: Our implementation of the membrane pattern to track the provenance of run-time values. In the synchronous version of our API for AranRemote, the remote advice is synchronous and can be composed with the provenance advice of Linvail as usual, either by extension or by layering.
- PosixSocket – `https://github.com/lachrist/posix-socket`: A C++ binding that exposes the low-level POSIX Socket API to JavaScript code in Node.js processes. This has greatly improved performance, as existing synchronous blocking communication mechanisms available to Node.js, such as synchronous file writing, involve significant overhead.
- Melf – `https://github.com/lachrist/melf`: A communication library that supports both non-blocking synchronous requests and asynchronous requests between JavaScript processes via a uniform interface, whether they are in the browser or in Node.js. The asynchronous mode of Melf is used exclusively when the analysis is implemented within our asynchronous API. Conversely, the synchronous mode of Melf is employed by the target process and by the analysis process if the analysis is written within our synchronous API.
- VirtualProxy – `https://github.com/lachrist/virtual-proxy`: Our mechanism to virtualize values based on the existing `Proxy` API, which avoids spurious signaling of invariant breakage by synchronizing the target object on an as-needed basis.
- MelfShare – `https://github.com/lachrist/melf-share`: A synchronous implementation of remote references based on Melf and VirtualProxy. Unlike other implementations of remote references, MelfShare returns actual values instead of promises. Although this makes them easier to use, it comes at the cost of performance overhead, as only Melf-related events can be processed while manipulating them.
- AranRemote – `https://github.com/lachrist/aran-remote`: The cornerstone of our approach, this module imports all other components except Linvail, which is only needed for provenance-aware analyses. After launching an AranRemote orchestrator, the application tiers must be launched within an AranRemote agent to handle code instrumentation and analysis-related communication. Note that AranRemote has not been updated to align with the new interface of Aran and currently relies on an outdated version of Aran.

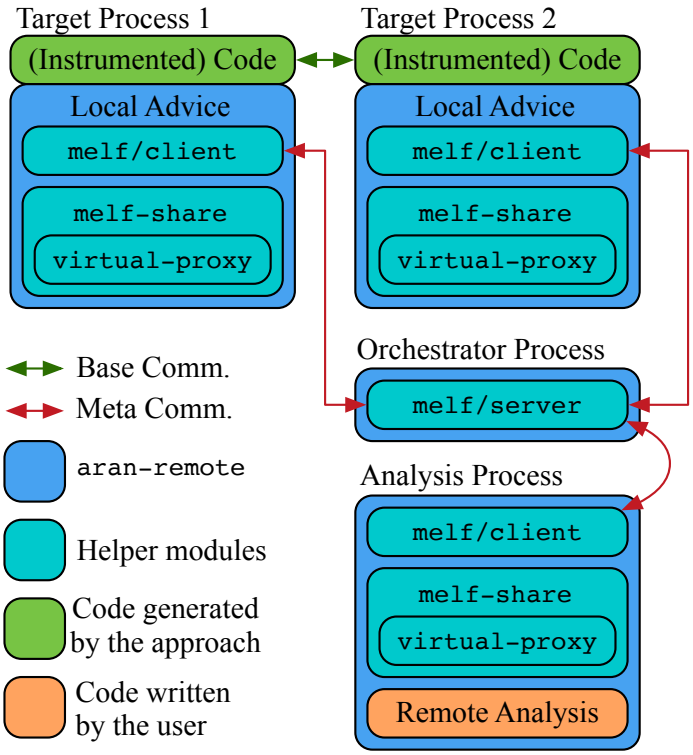


Figure 7.5: System architecture diagram of AranRemote after deployment

7.4 Demonstration: distributed origin analysis

In this section, we assess our approach by lifting a provenance-aware analysis to a distributed drawing application.

As a case study, we analyze a collaborative drawing editor called Whiteboard[2], which serves as a demonstrator for the popular `socket.io`[3] communication library. Although the application consists of only 120 lines of code, it generates significant communication to handle collaborative drawing. For each run-time value, the analysis collects the slice of the value flow graph (VFG) associated with that value. Whenever a value is used as an argument to a method of a canvas rendering context[4], its associated VFG is printed to the console.

Our case study analysis is presented in Listing 7.13 and reads as follows:

- Selective Instrumentation (Line 10): We only instrument the main file of the Whiteboard application; in particular, the `socket.io` library is not instrumented.
- Linvail Membrane (Lines 11–17): The most striking aspect of the analysis is that Linvail can be used out-of-the-box by our analysis. This is enabled by the transparency of synchronous remote references. As with regular Linvail analysis, a region must first be created (Line 11) to subsequently create membrane accessors (Line 12) and a provenance advice (Line 31). To set up Linvail's oracle, we register the intrinsic registry of the remote process as it connects (Line 15).
- Reporting (Lines 35–38): If the intercepted function application is a method of a canvas HTML element, the VFG of each argument is reported (Line 36). For debugging purposes, all method invocations are logged as well (Line 38).

[2] `https://github.com/socketio/socket.io/tree/master/examples/whiteboard`
[3] `https://github.com/socketio/socket.io`
[4] `https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D`

- Value Flow Graph (Lines 51–63): VFGs include three types of nodes: nodes of unknown origin which are the default (e.g., literal), nodes originating from a binary operation (Lines 51–57), and nodes originating from a `MouseEvent` (Lines 58–63).
- Inter-Process Tracking (Lines 19–29, and 39–50): The only logic related to distribution involves tracking values across processes. For this analysis, such tracking is managed by maintaining a centralized stack of promoted values. Whenever a promoted value is emitted through a socket, it is pushed onto the stack (Line 44). When a callback is registered on a socket, it is replaced by another callback (Line 48) that will substitute the received bare value with the promoted one on top of the stack (Line 28). Admittedly, this global mechanism is hacky and may lead to inconsistent tracking for larger applications with complex event interleaving. Integrating inter-process value tracking into the approach would be a worthwhile endeavor in the future.

We utilize AranRemote to lift the analysis of Listing 7.13 and deploy it on the distributed Whiteboard application, which was configured with two collaborative users. This setup results in the analysis of three processes: one Node.js process for the application server and two browser processes for the connected clients. For replication purposes, we have made our experiment available in an open-source repository[5].

Listing 7.14 depicts an excerpt of the output produced by the analysis. The initial message indicates that a line was drawn on the canvas of the second client at line 33 of the main file. The JSON object below the message represents the VFG of the first argument, which corresponds to the abscissa of the line's destination point. This value resulted from a successive division and multiplication of the same number: 1032. Inspecting the code reveals that this number corresponds to the width of the clients' canvas and that these operations are necessary to accommodate different canvas sizes. The division occurred in the first client at line 46 of the main file, while the multiplication took place in the second client at line 91 of the same file. The initial value 238 was obtained by accessing the `clientX` property of a mouse event[6]. In summary, our analysis was able to track a value originating from a click event on the first client all the way to its use in drawing a line on the second client.

Figure 7.6 clarifies how Linvail interacts with synchronous remote references. When the value at the `clientX` property from the mouse event is accessed, it is promoted inside the analysis process to track its provenance. The resulting wrapper is returned to the target process as a remote reference. After drawing its own line, the client assigns this value to an object that serializes the line-drawing operation and passes it to the `socket.io` library. However, since this library is not instrumented, it should not access the object directly, as it is contaminated by promoted values. Instead, it is provided as a remote reference to a Linvail proxy that enforces the frontier. In short, our approach successfully lifted Linvail to operate in a distributed context, which highlights its expressiveness.

It is challenging to quantify the performance overhead of our analysis on an event-driven program. Although we were able to interactively use the program under analysis, as illustrated in Figure 7.7, the performance overhead resulted in some jagged lines. As discussed in previous chapters, the Aran Linvail stack already incurs performance overhead when deployed locally. Deploying it in a distributed context is expected to compound this overhead. For instance, consider the scenario where the `socket.io` library accesses the property of an object representing a point. During analysis, this requires resolving three levels of indirection, with each resolution necessitating a remote procedure call.

[5] `https://github.com/lachrist/aran-remote-whiteboard`
[6] `https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent`

```
import {
  createRegion,
  createMembrane,
  createStandardAdvice,
  registerIntrinsicRecord,
} from "linvail";
import { log } from "node:console";
import Chalk from "chalk";
const { blue, green } = Chalk;

export const shouldInstrument = (path) => path.endsWith("/whiteboard/index.js");

const region = createRegion();

const { apply, wrapFreshHostReference } = createMembrane(region);

export const intrinsic_global_variable = "__INTRINSIC__";

export const connect = (global) => {
  registerIntrinsicRecord(global[intrinsic_global_variable]);
  return advice;
};

export const disconnect = (_advice) => {};

const stack = [];

const wrapCallback = (callback) => wrapFreshHostReference(function (...input) {
  if (input.length !== 1)
    throw new Error("Expected exactly one argument");
  if (stack.length === 0)
    throw new Error("Empty stack");
  const top = stack.pop();
  if (top.inner !== input[0])
    throw new Error("Mismatched value");
  return Reflect.apply(callback, this, [top]);
}, { kind: "function" });

const advice = {
  ...createStandardAdvice(region),
  "apply@around": (_state, callee, that, input, location) => {
    const callee_name = callee.inner?.name;
    const constr_name = that.inner?.constructor?.name;
    if (constr_name === "CanvasRenderingContext2D")
      log(blue(`At ${location}, context2d.${callee_name} was called with: ${input}`));
    if (constr_name)
      log(green(`Invoking ${constr_name} ${callee_name} at ${location}`));
    if (
      location.startsWith("client") && constr_name === "r" ||
      location.startsWith("server") && constr_name === "Socket"
    ) {
      if (callee_name === "emit") {
        stack.push(input[1]);
      } else if (callee_name === "on" && input[1].type === "host") {
        if (input.length !== 2)
          throw new Error("Expected exactly two arguments");
        input[1] = wrapCallback(input[1].plain);
      }
    }
    if (callee_name === "performBinaryOperation") {
      const [operator, left, right] = input;
      return {
        ... apply(callee, that, input),
        location,
        origin: { operator, left, right },
      };
    } else if (constr_name === "MouseEvent" && callee_name === "getValueProperty") {
      return {
        ... apply(callee, that, input),
        location,
        origin: `mouse.${input[1].inner}`,
      };
    } else {
      return apply(callee, that, input);
    }
  },
};
```

Listing 7.13: Remote analysis for recording a distributed value flow graph

```
At client2@http://localhost:3000/main.js#33:4, context2d.lineTo was called with:
```

```
[
  {
    "type": "primitive",
    "inner": 238,
    "location": "client2@http://localhost:3000/main.js#91:39",
    "origin": {
      "operator": { "type": "primitive", "inner": "*" },
      "left": {
        "type": "primitive",
        "inner": 0.23062015503875968,
        "location": "client1@http://localhost:3000/main.js#46:10",
        "origin": {
          "operator": { "type": "primitive", "inner": "/" },
          "left": {
            "type": "primitive",
            "inner": 238,
            "location": "client1@http://localhost:3000/main.js#61:35",
            "origin": "mouse.clientX"
          },
          "right": { "type": "primitive", "inner": 1032 }
        }
      },
      "right": { "type": "primitive", "inner": 1032 }
    }
  },
  ...
]
```

Listing 7.14: Distributed value flow graph recorded from the Whiteboard application

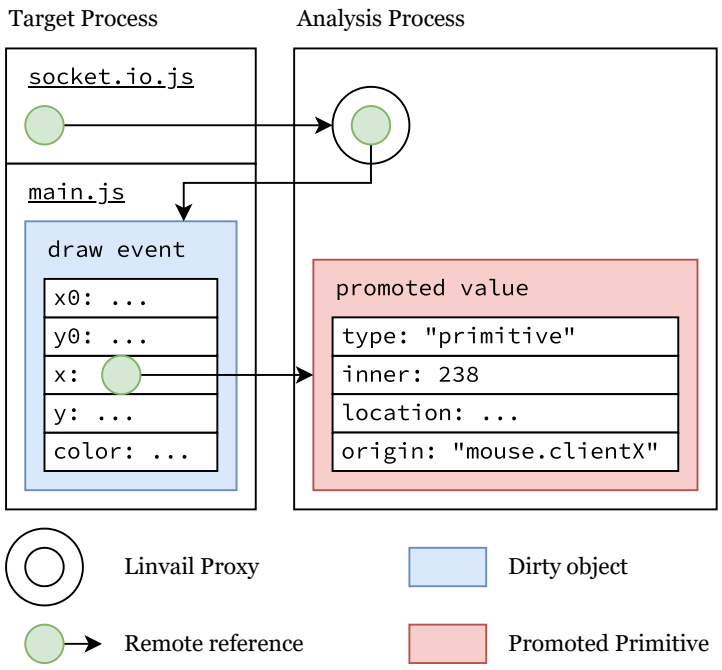


Figure 7.6: Interplay between Linvail's value promotion and remote references

Figure 7.7: Screenshot of the Whiteboard application during the analysis

7.5 Conclusion

In this chapter, we presented a novel approach to building dynamic analyses for distributed event-based programs. To effectively reason about shared state, our approach advocates centralizing the analysis logic within a remote process. This involves extensive communication between the central analysis process and the processes of the applications under analysis. To support this communication, we propose a protocol for conducting remote procedure calls synchronously but in a non-blocking manner. We implemented our approach in a tool called AranRemote and employed it to symbolically execute a collaborative drawing application. Our experiments demonstrate the expressiveness of the approach and show that it interacts seamlessly with our implementation of shadow values. To conclude, we summarize results and limitations, outline the main contributions, and review related work.

Assessment and opportunities for improvement

The primary limitation of our approach to dynamic analysis of distributed programs is its overhead, which affects both transparency and applicability. First, we eliminate the possibility of parallelization, as many operations performed on the target processes are handled sequentially by the central analysis process. Second, and more significantly, our approach generates a heavy communication load that cannot be executed concurrently within each target process without violating the run-to-completion principle of event-based systems.

For extensive pointcuts, this limitation has restricted the applicability of our approach to distributed programs with light computational loads. To alleviate this constraint and diminish performance overhead, it would be beneficial to develop a hybrid approach. In this approach, only the analysis logic that requires reasoning about the shared state of the distributed application would be executed in the central analysis process, while the remaining analysis logic would be executed locally.

While this approach will reduce performance overhead, it will also reintroduce some of the complexities associated with developing analyses for distributed applications. To preserve expressiveness and maintain the capacity for analysis implementers to write their analysis as a single advice, we could leverage static program slicing [48] to automatically split it across both the analysis process and the target processes. This approach has been previously explored in the context of distributed systems and is commonly

referred to as tierless programming [88, 82]. The distinction here is that the analysis meta layer will be split, rather than the base application layer.

Our approach can also be improved in the following areas:

- Instrumentation Independence: Currently, Aran is integrated into our approach, which is reflected in its name: AranRemote. Future work should explore how to make it independent of specific instrumentation, thereby enhancing modularity and promoting reuse.
- Inter-Process Tracking: One of the main applications of our approach consists of creating provenance-aware analyses capable of reasoning about distributed applications as a whole. While our approach can lift Linvail to track primitive values in each participating process separately, it does not facilitate such tracking across processes. In the future, it will be interesting to investigate how this could be facilitated by our approach without relying on any particular tracking library, such as Linvail.
- Protocol Verification: Although our protocol has been empirically validated, we have not formally verified key properties such as the absence of deadlock or the eventual delivery of messages. In the future, it would be beneficial to prove these critical properties by leveraging our formal description of the protocol as pushdown automata.
- Robust Connection: Our prototype implementation is not resilient to connectivity issues. Currently, when a target process disconnects, its associated remote references become unusable, and manipulating them results in exceptions being thrown. In the future, it will be interesting to improve the robustness of our approach by handling disconnections gracefully and re-enabling impacted remote references after reconnection.

Main contributions

The main contributions of this chapter are two-fold:

- We demonstrate the feasibility of conducting dynamic analysis targeting distributed systems in a centralized manner, which is highly expressive but incurs performance overhead. Our experiments show that this approach is compatible with complex techniques developed for single-process applications, such as our implementation of shadow values.
- We developed a new protocol for performing remote procedure calls in a synchronous yet non-blocking manner. We describe the protocol both with a JavaScript implementation and as pushdown automata.

Related work

In the literature, dynamic analyses for distributed programs have primarily taken the form of monitors and tracers (e.g., [112, 38, 7, 140]). For example, Dapper [112] traces communication within large-scale distributed programs by instrumenting key libraries. It has been utilized by Google developers for program comprehension and identifying performance issues. Another notable example is Sahand [3], which assists web developers in understanding full-stack JavaScript programs. However, the traces generated by these analyses often lack sufficient information for conducting provenance-aware analysis post-mortem.

There is also a body of work focused on conducting dynamic taint analysis at the system level. Although these approaches cannot handle distributed applications per se, they can still reason about the information flow between multiple processes within a single machine. For instance, Panorama [138] detects malware by conducting system-wide information flow tracking through dynamic binary transformation within the QEMU [9] virtual machine. Another notable work is TaintBochs [18], which detects failures to remove sensitive information. Unlike Panorama, TaintBochs does not rely on code instrumentation; instead, it utilizes a modified version of the Bochs[7] virtual machine. By operating at the system level, this body of work can be viewed as another approach to centralizing the analysis of distributed applications. However,

[7] `http://bochs.sourceforge.net`

this approach has two shortcomings: it can only operate at the binary code level and can only analyze processes that reside within the same system.

There has been limited research on conducting provenance-aware analysis in a truly distributed manner. First and most notably, TaintDroid [33] detects the leakage of sensitive information on the Android platform by performing system-wide information flow analysis across multiple Dalvik VM instances. Another example is DistTaint [39], which analyzes distributed Java applications through source code instrumentation. Both approaches utilize ad-hoc techniques to reason about shared state, which is the type of effort from which we aim to relieve users of our approach.

Chapter 8

Conclusion

In this dissertation, we presented a novel approach to conducting dynamic program analysis (with an emphasis on provenance-aware analyses) based on point-based instrumentation of JavaScript programs. We conclude by summarizing key results and contributions, as well as discussing related work and future directions. This chapter is organized as follows:

- *Section 8.1 revisits core contributions.*
- *Section 8.2 assesses our approach against the criteria from Section 2.4.*
- *Section 8.3 highlights limitations and proposes directions for future research.*
- *Section 8.4 provides some closing remarks.*

8.1 Summary and contributions

In this section, we synthesize this dissertation by revisiting the contributions previously introduced in Section 1.3. Compared to the next section, which is more quantitatively oriented, the discussion here is more qualitative.

Core variant Instead of addressing the full complexity of the target language directly, we propose that analyses should focus on a core variant. Although this idea is not novel, existing approaches such as CIL [81] and Soot [122] target low-level languages. To the best of our knowledge, no work has successfully represented the full complexity of a mainstream managed language as a core variant while remaining at the same level of abstraction (e.g., retaining closures and the meta-object protocol). In this dissertation, we choose JavaScript to illustrate our approach, and the core variant is named AranLang. This contribution spans the following sections:

- Section 3.2 formally defines the syntax of AranLang. The Abstract Syntax Tree (AST) nodes of AranLang are not only easier to reason about, but there are also significantly fewer of them.
- Section 3.3 informally defines the semantics of AranLang by outlining the transpilation process to JavaScript.
- Section 3.4 informally describes how to express the most challenging features of JavaScript within AranLang.
- Section 4.3 evaluates the ability of AranLang to represent the full complexity of JavaScript by considering the Test262 cases that fail after transpilation to and back from AranLang.
- Section 4.4 assesses performance overhead of AranLang by measuring the slowdown factors observed in the Octane benchmark after transpilation to and back from AranLang.

Comprehensive join point model To further facilitate the implementation of analyses, we propose incorporating program transformation into our approach by introducing an aspect-oriented interface. Aspect-oriented programming has long been leveraged for dynamic analysis through tools such as Racer [10], based on AspectJ, and DiSL [75], which conducts binary code instrumentation. However, to the best of our knowledge, our work is the first to introduce a comprehensive join point model for a high-level programming language that is capable of shadowing both the value stack and the scope. This contribution spans the following sections:

- Sections 4.1.1 and 4.1.2 present the standard join point model of AranLang, which exhibits clear semantics.
- Section 4.1.3 demonstrates the expressiveness of our standard join point model by leveraging it to implement an analysis for shadowing both the value stack and the scope.
- Section 4.3 evaluates the semantic transparency of our standard join point model by applying an analysis with an exhaustive pointcut onto Test262.
- Section 4.4 evaluates the performance overhead of our standard join point model by implementing an analysis with an exhaustive pointcut onto Octane.

Value-centric shadowing Traditionally, shadow values have been implemented through shadow execution [72, 83, 138, 14, 105], which is location-centric and involves mirroring parts of the run-time state. In contrast, our approach is value-centric and relies on value promotion and value virtualization. The advantage of our method over shadow execution is that it avoids synchronization issues when transferring control to unknown built-in functions or uninstrumented code areas. This contribution spans the following sections:

- Section 2.3 frames the issues of tracking provenance in a value-centric manner through the formal definition of an ideal provenancial equality operator.
- Section 5.2.1 introduces frontiers as a necessary mechanism to preserve transparency by controlling access to promoted values.
- Section 5.2.2 formally defines the concept of a frontier within a program. Although we do not explicitly define any frontier in this formalism, we find it helpful to anchor related discussions.
- Section 5.3 describes our adaptation of the membrane pattern from security, which relies on value virtualization to extend frontier design to parts of the store.
- Section 6.2 illustrates the expressiveness of our approach through the implementation of a dynamic taint analysis and a dynamic symbolic executer.
- Section 6.3 experimentally validates the semantic transparency of our shadow values by deploying them onto Test262.
- Section 6.4 experimentally evaluates the performance overhead of our shadow values by deploying them onto Octane.

Advice layering To disentangle analysis-specific logic from provenance tracking concerns, we propose a novel approach for composing advice that relies on stacking program transformations. The closest related work is found in the field of aspect-oriented programming (AOP), specifically through execution levels [118, 119], which allow aspects to advise other aspects while controlling cycles and avoiding infinite recursion. In contrast to our approach, where advice layering is fixed at instrumentation time, execution levels can be changed dynamically using built-in operators. While this capability is powerful, it necessitates additional run-time mechanisms and is not strictly required in our problem domain. The primary distinction is that execution levels are integrated into an AOP interpreter, whereas our approach relies on stacking instrumentation that can be applied unrestrictedly to any language. Other research has explored the combination of instrumentation-based analysis [129, 126, 45]; however, these studies primarily concentrate on ensuring non-interference. To the best of our knowledge, instrumentation stacking has not previously been proposed to achieve multi-layered analysis. This contribution spans the following sections:

- Section 5.4 presents our advice layering approach by providing an analysis for detecting candidates for ahead-of-time evaluation.
- Section 6.2.2 demonstrates the expressiveness of advice layering by providing an analysis capable of virtualizing values of all data types [5].
- Section 6.4 evaluates the performance overhead of advice layering by leveraging our approach to conduct a symbolic execution on Octane.

Provenance metric To study the precision of sound provenance providers, we propose a novel metric that associates a score with each run-time value, quantifying how much is known about its origin. To the best of our knowledge, our metric is the first effort in this direction. This contribution spans the following sections:

- Section 5.5 defines the rationale for the metric and provides an analysis for computing it.
- Section 6.5 experimentally evaluates how well the metric correlates with the extent to which provenance is tracked.

Central analysis of distributed applications To shield analysis implementers from the complexities of developing analyses for distributed systems, we propose centralizing the analysis-specific logic in a separate process. In contrast to traditional methods such as Dapper [112] and Pivot [73], which rely on the analysis of concurrent traces, our approach remains truly online. This contribution spans the following sections:

- Section 7.2 provides an overview of our architecture for centralizing the analysis of distributed applications.
- Section 7.3 offers insights into the implementation of our prototype, called AranRemote.
- Section 7.4 demonstrates the expressiveness of the approach by leveraging our prototype to symbolically execute a collaborative drawing application.

Synchronous non-blocking remote procedure calls We propose a novel protocol for event-driven programs that allows for synchronous invocation of remote procedures in a non-blocking manner. The primary feature of our protocol is its ability to pause the processing of the current event while remaining responsive to loopback remote procedure calls, thereby preventing deadlocks. To the best of our knowledge, such a protocol has not been proposed previously. This contribution spans the following sections:

- Section 7.3.1 explains why existing protocols cannot support our centralized architecture for analyzing distributed applications.
- Section 7.3.2 describes our protocol both as pushdown automata and as JavaScript executables.
- Section 7.3.3 details how our protocol can implement synchronous yet responsive remote references.

Technical contribution Throughout my years as a researcher, I have ensured that my artifacts can be utilized by others to develop analyses targeting real-world JavaScript programs. This commitment is reflected in the extensive validation presented in this dissertation, as well as in the collaborations listed in Section 1.4. This contribution spans the following sections:

- Section 4.2 describes Aran, our instrumentation infrastructure.
- Section 6.1 details Linvail, our implementation of shadow values.
- Section 6.1.5 focuses on VirtualProxy, our library that separates invariant bookkeeping from the target object of proxies.

- Sections 7.3 and 4.2.2 briefly mention Otiluke, our deployment infrastructure for both client-side and server-side JavaScript.
- Section 7.3 also briefly discusses Melf, the implementation of our protocol for performing remote procedure calls in a synchronous yet non-blocking manner.
- Section 7.3 presents AranRemote, our implementation for analyzing distributed applications in a centralized manner.

8.2 Assessment of our evaluation criteria

In Section 2.4, we defined five main criteria for evaluating approaches to implementing dynamic analyses based on point-based instrumentation of source code: applicability, expressiveness, precision, interpretability, *and* practicality. *We will now assess how our approach performs against each of these criteria.*

---

1. Applicability

The range of scenarios to which our approach can be applied.

The value of our approach depends on its ability to support a wide range of scenarios. Its applicability is shaped by the constraints it imposes on the kinds of analyses it can implement, the programming languages it can target, and the runtime environments it can operate in.

Cross-analysis applicability

The range of dynamic program analysis techniques that can be implemented with our approach (p. 29).

In Section 4.1, we outline the primary design decision of our approach: to provide an aspect-oriented interface for expressing analyses. Throughout this dissertation, we demonstrate how our approach can represent a diverse range of provenance-aware analyses, including generic shadow execution (Section 4.1.3), taint analysis (Section 6.2.3), symbolic execution (Section 6.2.1), an API for virtualizing all value types (Section 6.2.2), and distributed symbolic execution (Section 7.4).

Because our modular approach is capable of tracking provenance, we focus on demonstrating its application to provenance-aware analyses. However, we believe that an aspect-oriented interface on a core variant can facilitate the implementation of other analyses listed in Figure 2.2. For instance, profiling of objects and functions can be accomplished by advising calls to functions that implement operations of the meta-object protocol, while control flow analysis can be performed by advising branching and function calls.

■ Our approach is generic and can be utilized to develop a broad range of dynamic program analysis techniques.

Cross-platform applicability

> *The requirements imposed by our approach on the execution environments in which it can operate (p. 35).*

The primary advantage of program instrumentation over runtime instrumentation is its decoupling from the run-time environment. Our approach largely maintains this advantage; only in the case of load-time instrumentation does the runtime need to be configured to modify the import process.

■ Our approach remains independent of the execution environment, except for import hooks when instrumenting code at load time.

Cross-language applicability

> *The requirements imposed by our approach on the programming languages it can target (p. 35).*

Throughout this dissertation, our approach assumes that memory is not directly accessed and that value abstraction is consistently enforced, which is a defining feature of managed languages.

Chapter 3 illustrates that closures are crucial for normalizing the target language into a core variant. Furthermore, closures play an integral role in the advice API presented in Chapter 4. Consequently, our approach is not applicable to languages that lack support for closures as first-class citizens.

In Chapter 5, we relied on virtualizing record values to track provenance through the store. Later, in Chapter 7, we used the same mechanism to implement transparent remote references. While our approach is modular and can still be applied without these components, they form an important part of the value proposition of our approach. As discussed in Section 5.3.2, most mainstream languages feature this mechanism with varying degrees of transparency.

■ Our approach may also be applicable to other managed languages, provided they feature robust value abstraction, first-class closures, and transparent value virtualization.

2. Expressiveness

> *The extent to which our approach supports the specification of dynamic program analysis in a clear and concise manner.*

Applicability is meaningless without expressiveness; if our approach lacks robust abstractions, it will fail to address any actual specification issues experienced by analysis implementers. The primary goal of this dissertation is to provide these abstractions while preserving applicability.

Cohesion

> *The extent to which our approach exposes an interface with a small number of clear and well-defined entry points.*

Our approach requires analysis implementers to express analysis-specific logic as advice. To control the complexity of this advice, we proposed in Chapter 3 to perform weaving on a core variant of the target language, thereby shielding the analysis implementer from the non-essential syntactic complexity of the language. We illustrated our approach using JavaScript by defining a core variant named AranLang. AranLang not only contains 57% fewer AST nodes, but these nodes are also easier to reason about. As shown in Section 4.1, this enables the design of a join point model with only 31 points, while still

being capable of shadowing the stack and the scope. We believe that this approach can be applied to other managed programming languages, with varying degrees of success depending on the presence of an expressive core within the language.

While transpilation into a core variant is beneficial, it does not alleviate the low-level and imperative nature of specifying analyses in our approach. For example, our shadow execution analysis from Section 4.1.3 requires pervasive state management, which, although facilitated by local state arguments, remains error-prone.

Chapter 6 shows that the complexity introduced by our modular component for tracking value provenance depends on its integration with the analysis. When relying on advice extension (cf. Section 5.4.2), the analysis logic must be directly incorporated into a generic advice for enforcing a frontier. This approach exposes the analysis implementer to more entry points, such as promotion and demotion operations. Furthermore, it makes the analysis code more error-prone (even though static typing provides some safeguards), as bugs related to frontier violations can be easily introduced. In contrast, integration via advice layering (cf. Section 5.4.3) offers a simpler and safer interface, albeit at the cost of additional performance overhead.

■ Our approach offers a cohesive interface by performing instrumentation on a core variant of the language; however, it remains imperative, necessitates state management, and composition by extension can be error-prone.

Support for extrinsic information

The extent to which our approach supports maintaining information extrinsic to the baseline program execution.

In Chapter 5, we enhance the expressiveness of our approach for building provenance-aware analyses by presenting a modular component for tracking value provenance. Despite its apparent simplicity, value provenance is an important form of extrinsic information, playing a crucial role in prominent dynamic program analysis techniques such as taint analysis and symbolic execution. Our approach does not attempt to maintain other information extrinsic to program execution, such as formula-specific state for dynamic model checking.

■ Our approach features a modular component for providing value provenance but does not maintain other extrinsic information.

Support for distribution

The extent to which our approach supports the holistic analysis of distributed applications.

In Chapter 7, we further enhance expressiveness for analyzing distributed applications by centralizing analysis-specific logic within a separate process. Reasoning about distributed systems necessitates an analysis that is itself distributed to comprehend the shared state of the application. However, requiring analysis implementers to develop a distributed system of advice for each new analysis is a significant burden. This component of our approach radically addresses the problem, as the user advice is executed in a single process.

While our approach enables Linvail to be deployed on distributed applications, it does not facilitate tracking the provenance of data as it is passed from one process to another. Currently, this process is manual and requires the analysis implementer to develop an ad-hoc mechanism, similar to the one in our distributed origin analysis from Listing 7.13, to restore the provenance of values as they are created from data originating in another process.

■ Our approach drastically alleviates the complexity of analyzing distributed applications by centralizing the analysis logic in a dedicated process; however, it does not facilitate inter-process provenance tracking.

3. Precision

The extent to which our approach supports generating precise insights about the runtime behavior of the program under analysis.

The value of dynamic program analysis is contingent upon its ability to provide a narrow but precise view of program behavior. This strength must be preserved, which entails that our approach should be both *transparent* and *accurate*.

Transparency

The extent to which our approach supports non-interference with the execution of the program under analysis (p. 32).

There are two primary ways in which analyses can interfere with the execution of the target program and cause discrepancies: by directly altering its semantics (a.k.a. semantic overhead) or by introducing performance overhead. We first discuss semantic overhead, which has two main sources:

- Semantic discrepancies can be caused by the inability of the core variant to accurately express the entire semantics of the target language. Sometimes, it may be worthwhile to compromise transparency to simplify the core variant and enhance expressiveness. For instance, in AranLang, global variables are declared early in scripts, which causes some discrepancies but greatly simplifies the core variant. In Section 4.3.2, we demonstrate that AranLang is capable of expressing a large subset of JavaScript semantics, achieving a 99.8% success rate on the official conformance test suite of ECMAScript.
- Semantic discrepancies can be caused by the lack of robustness in the mechanism to virtualize compound values built into the target language. Our approach to tracking provenance relies on virtual values and their ability to resist revealing introspection. In Section 6.3.2, we demonstrate that our tracking of provenance is semantics preserving, as it only adds a 0.05% failure rate. This remarkable result is made possible by the underlying robustness of the built-in virtualization mechanism of JavaScript. Most other managed languages feature virtualization mechanisms but are less robust, which may lead to added discrepancies.

While the results regarding semantic overhead were largely positive, the findings related to performance overhead were more varied, revealing a compounding effect between the three main components of our approach.

- Section 4.4.2 discusses the performance overhead of normalization and basic analysis for the Octane benchmark. It appears that normalization alone incurs a slowdown factor of about 100×, which is surprisingly high, while simple analyses have a performance overhead of about 1,000×, which is acceptable in comparison. Notably, these findings were not replicated in the non-performance-oriented experiments of Sections 4.3.3 and 6.5.2, which feature only a 10× performance overhead for normalization. This variance might be attributed to a fine-tuning of V8 to perform well on Octane. Hence, we believe that the performance range for practical examples lies between 10× to 100×.
- Section 6.4.2 addresses the performance overhead of provenance tracking for the Octane benchmark. It appears that bare provenance tracking, analyses based on advice extension, and analyses based on advice layering incur additional multiplicative slowdowns of 10×, 100×, and 1,000×, respectively. These performance overheads may cause discrepancies in time-sensitive applications or render our approach impractical for computation-heavy programs.

- Section 7.4 qualitatively discusses the performance overhead of central analysis execution for a collaborative drawing application. Our evaluations showed that, although the distributed application remained usable under analysis, latency was noticeable, which is concerning given the minimal nature of the application. Accordingly, we believe additional work is required to control the performance overhead of this component so that it can be used in real-world scenarios, not just experiments.

Our approach also incurs memory overhead; however, we did not directly measure it during our experiments. We observed its impact on performance through manual investigations of the time spent in garbage collection. In particular, analyses built with Linvail have been noted to be memory-intensive due to the pervasive presence of handle objects.

■ Our approach is capable of achieving a high degree of semantic transparency, which is contingent upon the expressiveness of the core variant and the robustness of built-in value virtualization. However, performance overhead may be an intrinsic weakness of our approach and is likely to cause discrepancies in time-sensitive applications.

Accuracy

> *The extent to which our approach supports maintaining accurate extrinsic information (p. 31).*

The only extrinsic information provided by our approach is the provenance of values. As demonstrated throughout Chapter 5, this serves as a foundation for maintaining a wide range of extrinsic information related to run-time values, such as symbolic relationships to program input and information sensitivity. In Section 5.1.4, we opted to provide sound information regarding the provenance of run-time values at the expense of completeness. Consequently, our approach may indicate that two values do not share the same provenance when, in fact, they do.

In Section 6.1, we qualitatively discussed the completeness of our approach for JavaScript. We made the design decision not to track the provenance of data in some state locations, such as the boolean flags of object properties. But the main source of inaccuracy arises from Linvail's inability to track the provenance of data within exotic objects such as ES6 collections.

To quantitatively evaluate the completeness of our sound provenance provider, we proposed a novel metric in Section 5.5. This metric assigns a provenance score to individual run-time values based on the amount of information known about their origin. In Section 6.5.3, we demonstrated that our metric is easy to compute and correlates well with the extent to which provenance is tracked.

■ Our approach offers precise and sound tracking of provenance, although it cannot track provenance through exotic objects.

4. Interpretability

> *The extent to which our approach supports producing insights that are both understandable and actionable.*

Program analysis is valuable only when its insights can be effectively interpreted to enhance program comprehension and facilitate maintenance. Because of how generic our approach is, this responsibility primarily lies with the analysis implementer. However, we identified two important aspects of interpretability that remain within the scope of our approach: *source-adjacency* and *selectivity*.

Source-Adjacency

> *The extent to which our approach supports producing insights that can be traced back to the source code (p. 33).*

Transpilation to a core variant has improved the expressiveness of our approach; however, it has also complicated the process of tracing insights back to the source code due to transpilation expansion. For example, the most extensive transpilation expansion involves compiling classes into functions, in part by introducing a registry to support private fields. This departure from source-adjacency can be partially mitigated through source mapping, a well-established technique that associates transpiled AST nodes with their corresponding locations in the source code.

Nonetheless, our approach suggests that the core language be a close subset of the target language. In our JavaScript illustration, this is evidenced in Section 3.3 by the fact that the execution model of AranLang is very similar to that of JavaScript, with only minor technical differences, such as the presence of a global registry to hold intrinsic values. Consequently, building analyses and interpreting their results does not require users to learn a different execution model, as would be necessary if they were to analyze JavaScript programs at the level of WebAssembly, for instance.

■ Our approach maintains high source-adjacency compared to direct source instrumentation, despite the presence of transpilation expansion.

Selectability

> *The support of our approach to selectively analyze programs.*

By adopting an aspect-oriented interface, our approach provides built-in support for partial analysis through selective instrumentation. Our pointcut receives comprehensive information about the current join point, including its nature and location within the original source code. For instance, this information allows users to determine whether to instrument an application on a per-file basis or to target only specific syntactic constructs.

In addition, our implementation of shadow values relies on value promotion, which is able to maintain high accuracy even under partial instrumentation. This property is a by-product of the mediation mechanism required to preserve the transparency of value promotion. In contrast, as discussed in Section 5.3.1, the standard implementation of shadow values based on shadow execution has been known to struggle with partial analysis due to synchronization issues in the shadow memory after control is transferred back to the instrumented code.

■ Our approach effectively supports selective analysis through its aspect-oriented interface and the implementation of shadow values based on value promotion.

5. Practicality

> *The engineering effort required to implement our approach across its range of applicable scenarios.*

The engineering effort required to develop a robust, general-purpose platform for dynamic program analysis should not be underestimated. Therefore, the practicality of our approach hinges on the engineering efforts necessary to implement it across its various application scenarios.

Implementability

The simplicity of implementing our approach for a specific language and platform.

Our experience with implementing Aran has shown that prototyping program transformations is straightforward; however, supporting all corner cases is significantly more challenging. While we developed the first version of Aran within a few weeks, achieving a 99.8% success rate on Test262 took us years and required substantial engineering effort.

Our experience with implementing Linvail and AranRemote has demonstrated that while reflection is powerful and concise, it can also be challenging to reason about and prone to errors. For instance, our implementation of Linvail is significantly shorter than that of Aran; however, debugging Linvail proved to be more challenging due to the need to navigate additional layers of abstraction.

■ Our approach can be relatively challenging to implement; transparent transpilation in a core variant is tedious, and debugging reflective code can be difficult.

Portability

The effort necessary to port an existing implementation of our approach to other languages and platforms (p. 34).

Because it relies on source code instrumentation, our approach is inherently tied to a specific programming language, and porting it to others would likely require a complete rewrite.

On the other hand, relying on source code instrumentation has allowed our approach to remain decoupled from the execution environment. Our implementation demonstrates this portability, with the exception of load-time instrumentation, which requires runtime-specific mechanisms—such as a man-in-the-middle proxy for browsers or dedicated hooks in Node.js.

■ Our approach is closely tied to a specific programming language; however, it remains highly portable across runtimes and hardware.

Summary

Table 8.1 summarizes our assessment of the evaluation criteria from Section 2.4. We can characterize our approach as follows:

- Our approach is very generic and applicable to a wide range of scenarios in the dynamic analysis of managed languages. However, this flexibility requires manual effort to customize it into concrete analysis techniques.
- By conducting program instrumentation at the source code level, our approach is tightly coupled to the targeted programming language but remains portable across different runtimes and hardware.
- The first area for improvement in our approach is to control performance overhead, making it more suitable for analyzing time-sensitive applications.
- The second area for improvement in our approach is to enhance its expressiveness, as it currently requires manual effort and state management to implement concrete analysis techniques.

Applicability

Cross-Analysis	5/5	⊕ Highly generic
Cross-Language	3/5	⊕ Might be applicable to other managed languages ⊖ Requires value abstraction ⊖ Requires first-class closures ⊖ Requires value virtualization
Cross-Platform	4/5	⊕ No platform-specific assumptions ⊖ Requires import hook (for load-time instrum.)

Expressiveness

Cohesion	2/5	⊕ Simple join point model (via normalization) ⊖ Imperative with state management ⊖ Contingent on core variant ⊖ Advice extension is complex and error-prone Advice layering is simpler and safer
Support for Extrinsic Information	3/5	⊕ Automated intra-process provenance tracking ⊖ No support for dynamic model checking
Support for Distribution	4/5	⊕ Seamless analysis lifting to distribution ⊖ Manual inter-process provenance tracking

Precision

Transparency (Semantics)	4/5	⊕ Low semantic overhead ⊖ Contingent on expressive core variant ⊖ Contingent on robust value abstraction ⊖ Contingent on transparent value virtualization
Transparency (Performance)	1/5	⊖ High performance overhead ⊖ Compounding effect: normalization ⊖ Compounding effect: value virtualization ⊖ Compounding effect: synchronous communication ⊖ Compounding effect: advice layering
Accuracy	4/5	⊕ Precise and sound provenance tracking ⊖ Cannot track provenance through exotic objects

Interpretability

Source-Adjacency	4/5	⊕ Insights remain close to the source code ⊖ Transpilation expansion
Selectivity	5/5	⊕ Complete support for selective analysis

Practicality

Implementability	2/5	⊖ Transparent normalization is tedious ⊖ Reflective code is difficult to reason about
Portability (Language)	1/5	⊖ Highly coupled with target language
Portability (Platform)	5/5	⊕ No platform-specific assumptions

Table 8.1: Summary of the assessment of our evaluation criteria

8.3 Limitations and future work

We conclude this dissertation by discussing the technical limitations of our approach and how to address them, as well as by exploring interesting avenues for future work.

Limitations and areas for improvement

In this section, we discuss the most prominent areas for improvement, whether they arise from implementation shortcomings, technical limitations, or deeper issues.

Semantic transparency Both our experiments deploying our approach onto Test262 yielded very positive results, with a failure rate of only 0.2%. However, these semantic discrepancies should be taken seriously, as dynamic analysis tools are expected to be precise. While we could further improve semantic transparency, achieving full transparency is likely to compromise the simplicity of the core variant, thereby reducing the reusability potential of the approach. Another direction would be to implement a warning system to ensure that the absence of warnings guarantees full semantic transparency.

Performance overhead The main pain point of our approach is performance overhead: normalization alone incurs a slowdown factor of approximately 100× in the Octane benchmark, while Aran analyses and Linvail analyses incur performance overheads of 1000× and 10,000×, respectively. Although we observed consistently lower slowdown factors of about 10× in other experiments, there is no denying that performance overhead remains a weak point of our approach, impacting both its transparency and applicability. In the future, it will be valuable to investigate methods to reduce the slowdown factor, particularly for bare normalization, which is surprisingly high.

Provenance tracking within exotic objects The capability of our approach to track the provenance of immutable run-time values depends on the existence of a mechanism within the language to virtualize record values. While mediating access to properties of regular objects is well supported across mainstream languages, intercession on the hidden fields of so-called "exotic objects" is far less common. This limitation means that provenance cannot be tracked across important value locations, such as the hidden resolution field within promises. In the future, it would be worthwhile to evaluate the extent of changes required by language implementers to extend intercession to hidden property fields. Alternatively, other tracking mechanisms could be explored, although they would likely encounter either the two-body problem or synchronization issues.

Inter-process provenance tracking Tracking the provenance of values across processes offers clear advantages. For instance, a sound dynamic taint analysis can be made much more precise by avoiding over-tainting values originating from data transferred between processes. However, currently, inter-process provenance must be manually restored by the analysis implementer. The challenge of incorporating such a mechanism into our approach for centralizing the analysis lies in its ties with both the provenance provider and the inter-process communication mechanism, whether it involves high-level procedure calls or low-level messaging. In the future, it would be beneficial to investigate how our approach for centralized analysis could facilitate inter-process provenance tracking while remaining generic.

Directions for future work

We now discuss promising avenues for future work that could build upon our research.

Leverage AranLang to statically reason about JavaScript This dissertation primarily emphasizes dynamic program analysis. However, we believe that static program analysis techniques could also benefit from targeting AranLang rather than attempting to tackle the full complexity of JavaScript directly. Future investigations could explore the relative ease of formally reasoning about AranLang programs in comparison to JavaScript programs, and how this approach stands against existing methodologies such as CIL [81], which target low-level languages.

Domain-specific language for implementing analyses The main weakness of our approach is the manual effort required to leverage our framework to build user-facing tools. In particular, advice must often be written in an imperative style to manage state, which is error-prone. To alleviate this issue while remaining generic and applicable to a wide range of analyses, an interesting approach would be to develop a more declarative way to express analyses, for instance, by providing a domain-specific language. This could enhance cross-analysis reusability to the extent that analyses are developed on a per-application basis, transforming our approach into ready-to-use infrastructure for dynamic program analysis.

Exploring unsound provenance tracking We made the design decision to always promote values of unknown origin into new handles. This results in an approximation of provenancial equality that is sound but incomplete. In the future, it will be interesting to investigate how different promotion heuristics affect the precision of the corollary provenancial equality.

Formally defining the frontier We formally defined the concept of a frontier with respect to a set of internal values and a system for locating values within the run-time state. Currently, the presence of a frontier within a program is evaluated empirically through run-time checks. However, in the future, it will be worthwhile to further explore this formalism by precisely defining a frontier in these terms and investigating how to formally prove that a given program implements the specified frontier.

Further establishing our provenance metric We proposed a metric for measuring the precision of provenance providers independently from consuming analyses. Although we observed a strong correlation between the extent of frontier designs and recorded scores, the definition of our metric lacks intrinsic meaning for consuming analyses. Our metric could be further legitimized by evaluating how it correlates with the precision of representative provenance-aware analyses.

8.4 Closing remarks

This was long overdue, and I want to reiterate my thanks to my family, my promoters (Coen and Wolfgang), and my friends for their unwavering support. I also want to reiterate my thanks to the members of the jury (Michael, Tom, Ann, Elisa, and Jens) for taking the time to read through this long and sometimes dense text.

I am taking the opportunity to go "meta" and share some of my reflections on my research journey (so far). Hopefully, this will be of some help to fellow researchers. *Disclaimer*: I fully acknowledge that what follows is unsolicited advice, coming from a guy who took more than thirteen years to graduate.

Let me address the question: do I regret doing a PhD? The response is a resounding "no". Could I have graduated sooner? The response is a resounding "yes". And there lies the problem. While doing a long PhD was actually fun, as I could satisfy my curiosity to my heart's content, I have to consider the opportunity cost. What would life have had in store for me had I graduated sooner? Probably something more fulfilling and impactful than what I have achieved so far.

So I would advise fellow researchers not to lose track of time and to just dive into whatever problems stand in front of them. At the time, I know, it seems like this is the most important thing in the world. And this is probably what is required to be a good researcher. But one should not keep sinking time into any given problem.

Let me illustrate. When I started reviewing the related work for this text, as little as eight years ago, I built a web crawler to construct a citation graph by scraping Google Scholar. It seemed like a good idea at the time, but I spent way too much time on this tool trying to bypass captchas. I ended up getting the IP address of the VUB banned from Google Scholar for a couple of hours, only to not even include the result in the present text.

My point is: being a good researcher is about persevering to solve problems, which requires a sense of importance. But one should be mindful of the sunk cost fallacy. The value of one's work is not directly related to the amount of effort spent on it. My Google Scholar scraping tool certainly wasn't. So the flip side of being a good researcher is knowing when to capitulate. Research is about exploration: one cannot know in advance which efforts will be fruitful. We only have a finite amount of time, energy, and brain power, and they should be allocated wisely.

Related to the sunk cost fallacy, I want to address a more pernicious concern. Research is difficult. Sometimes one spends an entire day working hard and has nothing to show for it: no progress, no dopamine hits, *nothing*. Then one's brain starts wondering: could I have done something more enjoyable for the same effect? Well, if the alternative to research is taking a walk through a park, that is all fine and well. After a couple of walks, perhaps one will start re-learning that life is meaningless, and will resume research to extinguish existential anguish. That is how research should happen. One should learn to be okay with periods when little progress is being made.

The problem is that there are entire industries (social media, gaming, and content platforms) that profit from grabbing one's attention. It is not that these industries are evil per se, but their incentives are not aligned with human flourishing. In contrast to a walk in a park, these services can distract one for much longer. As one's dopamine circuits are hijacked while consuming these services, they start feeling guilty, of course. Only when guilt reaches a certain threshold does one resume some research; only to fall back into distraction when guilt diminishes.

For me, breaking this pernicious guilt-based cycle came from a combination of two things. First, I had far less time to allocate to my research (three kids and a full-time job will do that to you). Those long periods of hard work with little progress simply no longer occurred. Second, I became more gentle with myself. As long as I did not spend any time on gaming or YouTube, I did not allow myself to feel any guilt related to my research.

Be gentle with yourself, allocate your time and energy wisely, and enjoy the one life you have!

Bibliography

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.

[2] et all Alfred V. Aho Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. 1986. ISBN: 0-201-10088-6.

[3] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. "Understanding Asynchronous Interactions in Full-stack JavaScript". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE16)*. 2016.

[4] Frances E Allen. "Control flow analysis". In: *ACM Sigplan Notices*. Vol. 5. 7. ACM. 1970, pp. 1–19.

[5] Thomas H. Austin, Tim Disney, and Cormac Flanagan. "Virtual values for language extension". In: *Proceedings of the 26th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA11)*. 2011, pp. 921–938.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008.

[7] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. "Magpie: Online Modelling and Performance-aware Systems." In: *HotOS*. 2003, pp. 85–90.

[8] Kent Beck and Erich Gamma. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[9] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[10] Eric Bodden and Klaus Havelund. "Racer: effective race detection using aspectj". In: *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 155–166.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. "Directed greybox fuzzing". In: *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2017, pp. 2329–2344.

[12] Shawn A Bohner and Robert S Arnold. *Software change impact analysis*. Vol. 6. IEEE Computer Society Press Los Alamitos, 1996.

[13] Derek Bruening and Saman Amarasinghe. "Efficient, transparent, and comprehensive runtime code manipulation". PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering …, 2004.

[14] Prashanth P Bungale and Chi-Keung Luk. "PinOS: a programmable framework for whole-system dynamic instrumentation". In: *Proceedings of the 3rd international conference on Virtual execution environments*. ACM. 2007, pp. 137–147.

[15] Brian Burg. *jsprobes: cross-platform browser instrumentation using JavaScript*. 2012. URL: `https://brrian.tumblr.com/post/10571624125/jsprobes`.

[16] Yinzhi Cao, Zhichun Li, Vaibhav Rastogi, and Yan Chen. "Virtual browser: a web-level sandbox to secure third-party JavaScript without sacrificing functionality". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 654–656.

[17] Roberto Capizzi, Antonio Longo, VN Venkatakrishnan, and A Prasad Sistla. "Preventing information leaks through shadow executions". In: *2008 Annual Computer Security Applications Conference (ACSAC)*. IEEE. 2008, pp. 322–331.

[18] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. "Understanding data lifetime via whole system simulation". In: *USENIX Security Symposium*. 2004, pp. 321–336.

[19] Tom Christiansen, Larry Wall, Jon Orwant, et al. *Programming Perl: Unmatched power for text processing and scripting.* " O'Reilly Media, Inc.", 2012.

[20] Laurent Christophe, Wolfgang De Meuter, Elisa Gonzalez Boix, and Coen De Roover. "Linvail: A General-Purpose Platform for Shadow Execution of JavaScript". English. In: *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER).* 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER2016) ; Conference date: 14-03-2016 Through 18-03-2016. IEEE CS, Mar. 2016, pp. 260–270. ISBN: 978-1-5090-1855-0.

[21] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. "Aran: JavaScript Instrumentation for Heavyweight Dynamic Analysis". English. In: *Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop (BENEVOL).* Vol. 3941. Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop. Belgium-Netherlands Software Evolution Workshop 2024, Benevol 2024 ; Conference date: 21-11-2024 Through 22-11-2024. CEUR Workshop Proceedings, Nov. 2024, pp. 97–114. URL: https://benevol2024.github.io.

[22] Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. "Orchestrating Dynamic Analyses of Distributed Processes for Full-Stack JavaScript Programs". English. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming (GPCE).* Ed. by Eric Van Wyk and Tiark Rompf. International Conference on Generative Programming: Concepts & Experience, GPCE ; Conference date: 05-11-2018 Through 06-11-2018. ACM, Nov. 2018, pp. 107–118. ISBN: 978-1-4503-6045-6. DOI: 10.1145/3278122.3278135.

[23] Koen Claessen and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Acm sigplan notices* 46.4 (2011), pp. 53–64.

[24] Edmund M Clarke and E Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Workshop on Logic of Programs.* Springer. 1981, pp. 52–71.

[25] James Clause, Wanchun Li, and Alessandro Orso. "Dytan: a generic dynamic taint analysis framework". In: *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM. 2007, pp. 196–206.

[26] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. "A systematic survey of program comprehension through dynamic analysis". In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702.

[27] Dorothy E Denning. "A lattice model of secure information flow". In: *Communications of the ACM* 19.5 (1976), pp. 236–243.

[28] Dorothy E Denning and Peter J Denning. "Certification of programs for secure information flow". In: *Communications of the ACM* 20.7 (1977), pp. 504–513.

[29] Will Drewry and Tavis Ormandy. "Flayer: exposing application internals". In: (2007).

[30] Chuck Easttom. *Advanced JavaScript.* Jones & Bartlett Learning, 2010.

[31] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. "A survey on automated dynamic malware-analysis techniques and tools". In: *ACM computing surveys (CSUR)* 44.2 (2012), p. 6.

[32] Aryaz Eghbali and Michael Pradel. "DynaPyt: a dynamic analysis framework for Python". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 2022, pp. 760–771.

[33] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones". In: *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014), p. 5.

[34] Patrick Eugster. "Uniform proxies for Java". In: *Proceedings of the 21st International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA06).* 2006, pp. 139–152.

[35] Mattias Felleisen and Daniel P Friedman. "A calculus for assignments in higher-order languages". In: *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM. 1987, p. 314.

[36] Alan R Feuer. "si| an interpreter for the C language". In: *Proceedings of the 1985 Usenix Summer Conference, Portland, OR*. 1985.

[37] Robert E Filman, Daniel P Friedman, and Dennis Koga. "Aspect-oriented Programming is quantification and implicit invocation". In: *Aspects Oriented Software Development*. 2001.

[38] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. "X-trace: A pervasive network tracing framework". In: *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association. 2007, pp. 20–20.

[39] Xiaoqin Fu and Haipeng Cai. "A dynamic taint analyzer for distributed systems". In: *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2019, pp. 1115–1119.

[40] Dimitra Giannakopoulou and Klaus Havelund. "Automata-based verification of temporal properties on running programs". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 412–416.

[41] GitHub Staff. *Octoverse: AI leads Python to top language as the number of global developers surges*. https://github.blog/news-insights/octoverse/octoverse-2024/. GitHub Blog, accessed 2025-09-02. Oct. 2024.

[42] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI05)*. 2005, pp. 213–223.

[43] Patrice Godefroid, Michael Y Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing". In: *Communications of the ACM* 55.3 (2012), pp. 40–44.

[44] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. Citeseer. 2008, pp. 151–166.

[45] Michael Gorbovitski, Yanhong A Liu, Scott D Stoller, and Tom Rothamel. "Composing transformations for instrumentation and optimization". In: *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*. 2012, pp. 53–62.

[46] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. "The essence of JavaScript". In: *European conference on Object-oriented programming*. Springer. 2010, pp. 126–150.

[47] Carl A Gunter. *Semantics of programming languages: structures and techniques*. MIT press, 1992.

[48] Mark Harman and Robert Hierons. "An overview of program slicing". In: *software focus* 2.3 (2001), pp. 85–92.

[49] Klaus Havelund and Grigore Roşu. "Efficient monitoring of safety properties". In: *International Journal on Software Tools for Technology Transfer* 6.2 (2004), pp. 158–173.

[50] Klaus Havelund and Grigore Roşu. "Synthesizing monitors for safety properties". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2002, pp. 342–356.

[51] Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. "Reflection for the Masses". In: *Workshop on Self-sustaining Systems*. Springer. 2008, pp. 87–122.

[52] William E Howden. "Theoretical and empirical studies of program testing". In: *IEEE Transactions on Software Engineering* 4 (1978), pp. 293–298.

[53] Hajime Inoue, Frank Adelstein, Matthew Donovan, and Stephen Brueckner. "Automatically bridging the semantic gap using C interpreter". In: *Proc. of the 2011 Annual Symposium on Information Assurance*. Citeseer. 2011, pp. 51–58.

[54] Yves Jaradin, Fred Spiessens, and Peter Van Roy. *Capability confinement by membranes*. Tech. rep. Technical Report Research Report RR2005-03, Dep. of Comp. Science and Eng …, 2005.

[55] Kangkook Jee, Vasileios P Kemerlis, Angelos D Keromytis, and Georgios Portokalidis. "ShadowReplica: efficient parallelization of dynamic data flow tracking". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 235–246.

[56] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[57] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. "Dta++: dynamic taint analysis with targeted control-flow propagation." In: *NDSS*. 2011.

[58] Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen. "Platform-independent dynamic taint analysis for javascript". In: *IEEE Transactions on Software Engineering* 46.12 (2018), pp. 1364–1379.

[59] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. "A framework for constructing javascript virtual machines with customized datatype representations". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1238–1247.

[60] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-oriented programming". In: *ECOOP'97 – Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer. 1997, pp. 220–242.

[61] Gary A Kildall. "A unified approach to global program optimization". In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM. 1973, pp. 194–206.

[62] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. "Implicit flows: Can't live with 'em, can't live without 'em". In: *Information Systems Security: 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings 4*. Springer. 2008, pp. 56–70.

[63] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[64] Donald E Knuth. "An empirical study of FORTRAN programs". In: *Software: Practice and experience* 1.2 (1971), pp. 105–133.

[65] Bogdan Korel and Janusz Laski. "Dynamic program slicing". In: *Information processing letters* 29.3 (1988), pp. 155–163.

[66] Peter J Landin. "The mechanical evaluation of expressions". In: *The computer journal* 6.4 (1964), pp. 308–320.

[67] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. "Challenges for static analysis of java reflection-literature review and empirical study". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 507–518.

[68] Daniel Lehmann and Michael Pradel. "Wasabi: A framework for dynamically analyzing webassembly". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 1045–1058.

[69] Henry M Levy. *Capability-based computer systems*. Digital Press, 2014.

[70] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. "A survey of code-based change impact analysis techniques". In: *Software Testing, Verification and Reliability* 23.8 (2013), pp. 613–646.

[71] Blake Loring, Duncan Mitchell, and Johannes Kinder. "ExpoSE: practical symbolic execution of standalone JavaScript". In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. 2017, pp. 196–199.

[72] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI05)*. 2005, pp. 190–200.

[73] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot tracing: Dynamic causal monitoring for distributed systems". In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), pp. 1–28.

[74] Pattie Maes. "Concepts and experiments in computational reflection". In: *ACM Sigplan Notices* 22.12 (1987), pp. 147–155.

[75] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. "DiSL: a domain-specific language for bytecode instrumentation". In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM. 2012, pp. 239–250.

[76] John McCumber. *Assessing and managing security risk in IT systems: A structured methodology*. Auerbach Publications, 2004.

[77] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall Englewood Cliffs, 1997.

[78] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. "Safe active content in sanitized JavaScript". In: *Google, Inc., Tech. Rep* (2008).

[79] Mark Samuel Miller and Jonathan S Shapiro. "Robust composition: towards a unified approach to access control and concurrency control". PhD thesis. Johns Hopkins University, Baltimore, Maryland, USA, 2006.

[80] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[81] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. "CIL: Intermediate language and tools for analysis and transformation of C programs". In: *International Conference on Compiler Construction*. Springer. 2002, pp. 213–228.

[82] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. "Tierless programming and reasoning for {software-defined} networks". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 519–531.

[83] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI07)*. 2007, pp. 89–100.

[84] James Newsome and Dawn Song. "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software". In: (2005).

[85] Srinivas Nidhra and Jagruthi Dondeti. "Black box and white box testing techniques-a literature review". In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2 (2012), pp. 29–50.

[86] Hyukwoo Park, Seonghyun Kim, and Boram Bae. "Dynamic code compression for JavaScript engine". In: *Software: Practice and Experience* 53.5 (2023), pp. 1196–1217.

[87] Harish Patil and Charles N Fischer. "Efficient Run-time Monitoring Using Shadow Processing." In: *AADEBUG*. Vol. 95. 1995, pp. 1–14.

[88] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. "Towards tierless web development without tierless languages". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2014, pp. 69–81.

[89] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[90] Thomas M Pigoski. *Practical software maintenance: best practices for managing your software investment*. Wiley Publishing, 1996.

[91] Gordon D Plotkin. "A structural approach to operational semantics". In: (1981).

[92] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE. 1977, pp. 46–57.

[93] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. "Python: The full monty". In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 217–232.

[94] Angel Luis Scull Pupo. "Language-Based Security for Web Applications". PhD thesis. Vrije Universiteit Brussel, 2021.

[95] Axel Rauschmayer. *Speaking JavaScript: an in-depth guide for programmers*. " O'Reilly Media, Inc.", 2014.

[96] Jie Ren, Ling Gao, and Zheng Wang. "JavaScript Performance Tuning as a Crowdsourced Service". In: *IEEE Transactions on Mobile Computing* 23.5 (2023), pp. 6116–6132.

[97] Henry Gordon Rice. "Classes of recursively enumerable sets and their decision problems". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[98] Andrei Sabelfeld and Andrew C Myers. "Language-based information-flow security". In: *IEEE Journal on selected areas in communications* 21.1 (2003), pp. 5–19.

[99] José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. "Symbolic execution for JavaScript". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. 2018, pp. 1–14.

[100] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. "A symbolic execution framework for javascript". In: *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE. 2010, pp. 513–528.

[101] Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. "Practical Information Flow Control for Web Applications". English. In: *Proceedings of the 18th International Conference on Runtime Verification (RV)*. Vol. 11237. 18th International Conference on Runtime Verification, RV ; Conference date: 11-11-2018 Through 13-11-2018. Springer, Nov. 2018, pp. 372–388. ISBN: 978-3-030-03768-0. DOI: 10.1007/978-3-030-03769-7_21.

[102] Peter Seibel. *Coders at work: Reflections on the craft of programming*. Apress, 2009.

[103] Koushik Sen. "Concolic testing". In: *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*. 2007, pp. 571–572.

[104] Koushik Sen and Gul Agha. "CUTE and jCUTE: concolic unit testing and explicit path model-checking tools". In: *Proceedings of the 18th International ConferenceComputer Aided Verification (CAV06)*. 2006, pp. 419–423.

[105] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. "Jalangi: a selective record-replay and dynamic analysis framework for JavaScript". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. 2013, pp. 488–498.

[106] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: a concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 30. 5. ACM. 2005, pp. 263–272.

[107] Manuel Serrano. "JavaScript AOT compilation". In: *ACM SIGPLAN Notices* 53.8 (2018), pp. 50–63.

[108] Manuel Serrano. "Of JavaScript AOT compilation performance". In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–30.

[109] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. "EROS: a fast capability system". In: *Proceedings of the seventeenth ACM symposium on Operating systems principles*. 1999, pp. 170–185.

[110] Masaaki Shimasaki, Shigeru Fukaya, Katsuo Ikeda, and Takeshi Kiyono. "An analysis of Pascal programs in compiler writing". In: *Software: Practice and Experience* 10.2 (1980), pp. 149–157.

[111] Olin Shivers. "Control-flow analysis of higher-order languages". PhD thesis. Citeseer, 1991.

[112] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. *Dapper, a large-scale distributed systems tracing infrastructure*. Tech. rep. Technical report, Google, Inc, 2010.

[113] Brian Cantwell Smith. "Reflection and semantics in Lisp". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1984, pp. 23–35.

[114] Amitabh Srivastava and Alan Eustace. *ATOM: A system for building customized program analysis tools*. Vol. 29. 6. ACM, 1994.

[115] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. "Precise client-side protection against DOM-based Cross-Site scripting". In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 655–670.

[116] Abel Stuker, Aäron Munsters, Angel Luis Scull Pupo, Laurent Christophe, and Elisa Gonzalez Boix. "JASMaint: Portable Multi-language Taint Analysis for the Web". English. In: *Proceedings of the 22nd International Conference on Managed Programming Languages and Runtimes (MPLR)*. Belgium-Netherlands Software Evolution Workshop 2024, Benevol 2024 ; Conference date: 21-11-2024 Through 22-11-2024. ACM, Oct. 2025. URL: https://conf.researchr.org/home/icfp-splash-2025/mplr-2025.

[117] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. "Efficient dynamic analysis for Node.js". In: *Proceedings of the 27th International Conference on Compiler Construction*. 2018, pp. 196–206.

[118] Éric Tanter. "Execution levels for aspect-oriented programming". In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*. 2010, pp. 37–48.

[119] Eric Tanter, Philippe Moret, Walter Binder, and Danilo Ansaloni. "Composition of dynamic analysis aspects". In: *Proceedings of the ninth international conference on Generative programming and component engineering*. 2010, pp. 113–122.

[120] David Thomas, Andrew Hunt, Chad Fowler, et al. *Programming Ruby: the pragmatic programmers' guide*. Raleigh, NC: Pragmatic Bookshelf, 2005.

[121] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. "eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems". In: *Journal of Computer Languages* 51 (2019), pp. 261–279.

[122] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. "Soot: A Java bytecode optimization framework". In: *CASCON First Decade High Impact Papers*. 2010, pp. 214–224.

[123] Tom Van Cutsem and Mark S Miller. "Proxies: design principles for robust object-oriented intercession APIs". In: *Proceedings of the 6th Symposium on Dynamic Languages (DLS10)*. 2010, pp. 59–72.

[124] Tom Van Cutsem and Mark S Miller. "Trustworthy proxies". In: *ECOOP 2013–Object-Oriented Programming*. Springer, 2013, pp. 154–178.

[125] David Van Horn and Matthew Might. "Abstracting abstract machines". In: *ACM Sigplan Notices* 45.9 (2010), pp. 51–62.

[126] Jonne Van Wijngaarden, Eelco Visser, et al. "Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems". In: (2003).

[127] Maarten Vandercammen. "Inter-process concolic testing of full-stack JavaScript web applications". In: (2023).

[128] Maarten Vandercammen, Laurent Christophe, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. "Prioritising Server Side Reachability via Inter-process Concolic Testing". English. In: *The Art, Science, and Engineering of Programming* 5.2 (Oct. 2020). ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2021/5/5.

[129] Eelco Visser. "A survey of rewriting strategies in program transformation systems". In: *Electronic Notes in Theoretical Computer Science* 57 (2001), pp. 109–143.

[130] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. "Cross site scripting prevention with dynamic data tainting and static analysis". In: *Proceedings of the Network and Distributed System Security Symposium (NDSS07)*. 2007.

[131] Thorsten Von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. "J-Kernel: A capability-based operating system for Java". In: *Secure Internet programming: security issues for mobile and distributed objects* (1999), pp. 369–393.

[132] Shiyi Wei and Barbara G Ryder. "Practical blended taint analysis for JavaScript". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, pp. 336–346.

[133] Mark Weiser. "Program slicing". In: *Proceedings of the 5th international conference on Software engineering*. IEEE Press. 1981, pp. 439–449.

[134] Christian Wimmer and Thomas Würthinger. "Truffle: a self-optimizing runtime system". In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 2012, pp. 13–14.

[135] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[136] Babak Yadegari and Saumya Debray. "Bit-level taint analysis". In: *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2014, pp. 255–264.

[137] Qian Yang, J Jenny Li, and David M Weiss. "A survey of coverage-based testing tools". In: *The Computer Journal* 52.5 (2009), pp. 589–597.

[138] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. "Panorama: capturing system-wide information flow for malware detection and analysis". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 116–127.

[139] Qin Zhao, Derek Bruening, and Saman Amarasinghe. "Umbra: Efficient and scalable memory shadowing". In: *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 2010, pp. 22–31.

[140] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. “lprof: A non-intrusive request flow profiler for distributed systems”. In: *OSDI*. Vol. 14. 2014, pp. 629–644.