VRIJE
UNIVERSITEIT
BRUSSEL

Master thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Applied Sciences and Engineering: Computer Science

BoaSpect

Portable and performant interpreter-based instrumentation for JavaScript

Äron Munsters

2021–2022

Promotor: Prof. Dr. Elisa Gonzalez Boix
Advisor: Dr. Angel Luis Scull Pupo

Science and Bio-Engineering Sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen

BoaSpect

Draagbare en performante interpreter-gebaseerde instrumentatie voor JavaScript

Aäron Munsters

2021–2022

Promotor: Prof. Dr. Elisa Gonzalez Boix
Begeleider: Dr. Angel Luis Scull Pupo

Wetenschappen en Bio-ingenieurswetenschappen

Abstract

Although JavaScript is the programming language dominating the web, applications built with it are hard to diagnose due to its dynamic and permissive features. As a consequence, programmers use dynamic analysis tools to facilitate debugging and testing, putting the focus of research on instrumentation platforms which support the implementation and deployment of dynamic analyses.

With current instrumentation platforms for JavaScript, analysis developers face the choice between portability or performance. Source code instrumentation platforms inject the analysis into the target program by rewriting its source code to a variant including the analysis. Thereby it provides a portable approach that can run in any JavaScript engine but suffers from a high-performance overhead and a lack of transparency. In contrast, abstract syntax tree instrumentation platforms wrap the parsed program tree within the engine with instructions that include the dynamic analysis for further execution. This approach offers high performance but suffers from a lack of portability as it is often specific to a JavaScript engine. The choice between portability or performance limits the analysis developer as they would preferably have both.

In this work, we explore our implementation BoaSpect, an instrumentation platform which offers portability to enable execution in different runtimes while offering good performance. BoaSpect is built at the level of bytecode interpretation in the JavaScript interpreter Boa, which leverages on WebAssembly as its compilation target for portability. A benefit of designing BoaSpect at the interpreter level is the opportunity to extend the instrumentation interface with lower-level traps to the analysis developer which is hard, if not impossible, to implement and maintain using source code instrumentation.

To evaluate BoaSpect, we develop analyses using our extended instrumentation interface. More specifically we evaluate its performance using the Sunspider benchmark suite. We observe three key insights. First, our extended interface enables defining analyses which cannot be implemented for source code instrumentation platforms due to source code reflection limitations. Next, when comparing the same input analyses for both BoaSpect and source code instrumentation running on the Boa execution engine, we observe BoaSpect's execution to be 3 to 5 times faster than that of source code instrumentation. Third, when targetting our approach to WebAssembly it has a performance similar to source code instrumentation, while still enabling more analyses and better instrumentation transparency.

Abstract

Hoewel JavaScript de programmeertaal is die het web domineert, zijn toepassingen die ermee gebouwd zijn moeilijk te diagnoseren door de dynamische en toegeefelijke eigenschappen ervan. Als gevolg gebruiken ontwikkelaars dynamische analyse-instrumenten om het debuggen en testen te vergemakkelijken. Hierdoor heeft onderzoek zich toegespitst op instrumentatieplatformen die de implementatie en het gebruik van dynamische analyses ondersteunen.

Met de huidige instrumentatieplatformen voor JavaScript moeten analyse-ontwikkelaars de afweging maken tussen overdraagbaarheid of prestatie. Broncode-instrumentatieplatformen plaatsen de analyse in het doelprogramma door de broncode ervan te herschrijven naar een variant die de analyse bevat. Zo bieden ze een overdraagbare aanpak die in elke JavaScript omgeving kan draaien. Het nadeel is dat deze aanpak lijdt aan een hoge prestatie kost en een tekort aan transparantie. Abstracte syntax boom instrumentatieplatformen daarentegen omhullen de programmaboom binnen in de *engine* met instructies die de dynamische analyse voor verdere uitvoering bevatten. Deze aanpak biedt hoge prestaties, maar lijdt aan een gebrek aan overdraagbaarheid omdat hij vaak specifiek is voor een JavaScript *engine*. De keuze tussen overdraagbaarheid of prestatie bemoeilijkt het voor de analyse-ontwikkelaar aangezien de combinatie van beiden het meest wenselijke is.

In dit werk verkennen we een instrumentatieplatform dat zowel goede prestaties als overdraagbaarheid biedt om de uitvoering in verschillende JavaScript omgevingnen mogelijk te maken. Daartoe stellen we BoaSpect voor, een instrumentatieplatform ontwikkeld op het niveau van *bytecode*-interpretatie in de JavaScript-interpreter Boa, dat gebruik maakt van WebAssembly als compilatiedoel voor verhoogde overdraagbaarheid. Een voordeel van de ontwikkeling van BoaSpect op interpreterniveau is de mogelijkheid om de instrumentatie-*interface* uit te breiden met traps op lager niveau voor de analyse-ontwikkelaar. Deze zijn met de broncode-instrumentatie methode moeilijk, zo niet onmogelijk, te implementeren en te onderhouden.

Om BoaSpect te evalueren, ontwikkelen we analyses met behulp van onze uitgebreide instrumentatie-*interface*. Meer specifiek maken we gebruik van het Sunspider benchmark pakket. We bekomen drie hoofdzakelijke inzichten. Ten eerste, maakt onze uitgebreide *interface* het mogelijk om analyses te definiëren die niet geïmplementeerd kunnen worden met broncode instrumentatie platformen als

gevolg van de beperkingen van broncode reflectie. Ten tweede, zien we dat wanneer we de broncode-instrumentatie techniek vergelijken met BoaSpect, gebruik makend van dezelfde invoeranalyses en beiden draaiend op de Boa *engine*, dat onze aanpak 3 tot 5 keer sneller programma's uitvoert dan de uitvoering met broncode-instrumentatie. Ten derde, wanneer we onze aanpak draaien op WebAssembly, merken we een gelijkaardige prestatie-verhoudingen op als die van de broncode-instrumentatie, terwijl we toch meer analyses en een betere transparantie van de instrumentatie mogelijk maken.

Acknowledgements

First, I would like to express my gratitude to my promotor, Elisa Gonzalez Boix, for her invaluable guidance and support during my work on this thesis. Second, I sincerely thank my advisor, Angel Luis Scull Pupo, for the many hours we shared both on and off-topic. For pushing me further than I could ever imagine myself going on my journey at the university, in doing so further shaping my knowledge and enriching my experience.

To my parents, for their love and care, supporting me in every way they could, I am grateful on so many levels, I could not have done this without you.

And finally, to my girlfriend, who has been with me since I first set foot on campus, helping me all the way through, sharing many beautiful moments of happiness, I thank you.

Declaration of originality

I hereby declare that this thesis was entirely my own work and that any additional sources of information have been duly cited.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this thesis has not been submitted for a higher degree to any other University or Institution.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

The ubiquity of JavaScript as the dominating language for the web is undeniable. It powers web and mobile applications. JavaScript also serves as the driving mechanism for backend services or even a combination of both in the case where JavaScript is used for full-stack applications [2].

The dynamic nature of JavaScript and its flexibility often results in vulnerable or misbehaving code. For example, JavaScript allows loading new code at runtime through the use of `<script>`-tags in webpages or the use of *eval* to evaluate a string as a program in the same context. Another example of JavaScript's dynamic nature is that the language at runtime allows changing the layout of objects flowing through the program at will [3]. This dynamicity offers a high degree of freedom and flexibility to developers, however, it also comes at a cost. These features pose certain risks to the quality of ever-growing programs, including, but not limited to the introduction of application vulnerabilities, poorly optimizable code to execute or sleeping bugs that wake at runtime [3].

Program analysis helps developers to test and debug programs. We distinguish between two big families of program analysis tools: static and dynamic programming tools.

Static analysis tools analyze the code without execution [4, 5]. These static analysis tools for JavaScript, however, are severely limited in their ability to draw conclusions from programs that rely on these dynamic features. For example, a static analysis tool that detects the presence of dead code (code that is never used at runtime) cannot answer with certainty whether a JavaScript-defined function will be used if a portion of the program will be added at runtime by a `<script>`-tag.

On the other hand, dynamic program analysis tools trace the program execution while using instruments to reason and potentially act on the program behaviour [6]. These instruments measure properties of the program under execution including aspects such as the memory use, the values handled by a program or the input and output values. The focus of this work is to support the development of dynamic program analysis for JavaScript.

1.1 Problem Statement

Automatically augmenting a program with instruments to enable dynamic analysis can be achieved using an instrumentation platform. An instrumentation platform is there to ease the development of an analysis and provide the mechanisms to inject the analysis into the execution of the target program under analysis.

For JavaScript, there exist different families of instrumentation platforms categorized by their implementation approach [7, 8, 9, 10, 11].

On the one hand, JavaScript instrumentation platforms have been developed to run the analysis and instrumentation at the source code level [9, 10]. This technique rewrites the target program source code to a variant in which the instrumentation and analysis code is included. Thus, the analysis and instrumentation are shipped with the target program under analysis to an execution environment. On the other hand, the JavaScript instrumentation can take place at the level of the program interpretation. This entails that the interpreter that executes the target program under analysis additionally performs the instrumentation as instructed by the analysis [11] during evaluation.

As both these approaches for implementing an instrumentation platform share the same goal, ie. enabling analysis developers to deploy a dynamic analysis, we introduce four different properties in Section 2.3 which we will use to compare state of the art (in Section 2.4): portability, performance, transparency and expressiveness.

As we argue later in Section 2.4.3, instrumenattion at the level of source code excels in *portability* property. This instrumentation technique can be deployed in a multitude of execution environments as the instrumentation is included alongside the source code of the target JavaScript program. However, source code instrumentation suffers from its *performance* property. This is concerned with the instrumentation platform performance overhead, affecting the overall time required to execute the target program. This performance overhead is high for source code instrumentation as each event of interest is rewritten to code that calls into the analysis definition, losing opportunities for the execution engine to optimize the input program [11].

Interpreter-based instrumentation performs better in terms of lower performance overhead [11]. Furthermore, interpreter-based instrumentation remains more easily uncovered from the target program under analysis compared to source code instrumentation, which we qualify as the *transparency* property. Reflective JavaScript such as querying the string-representation of a function body can leak instrumentation information when the function is rewritten to include instrumentation and analysis code [12]. Contrary to source code instrumentation, interpreter-based instrumentation operates at the level of the interpreter, making it more difficult to uncover the modifications as the interpreter internals are not exposed to JavaScript.

Finally, interpreter-based instrumentation offers higher *expressiveness* for the analysis developer, with which we mean that it can more easily provide program execution information to the analysis [11]. An example of information that can be challenging to provide to the analysis developer is operations that are taking place within the interpreter, such as whenever the execution engine internally performs type conversion on its values.

So far, these distinct families of instrumentation platform implementation techniques for JavaScript seem to point out that an analysis developer has to decide between either two. This implies the analysis developer has to choose between either *portability* or *performance*, *transparency* and *expressiveness*. Preferably, one would have both *portability* to analyse code which runs in browsers and servers while maintaining *performance*, *transparency* and *expressiveness* to enable the analysis to trace an execution true to the intended target application behaviour.

We observe there is no instrumentation platform that can offer a good solution for portability and performance. We hypothesize that it is possible to have an interpretation-based approach which pays little performance overhead but offers portability.

1.2 Developing a Portable Interpreter-Based Instrumentation Platform

In this work, we explore the implementation of an interpreter-based instrumentation platform that can compile to WebAssembly [13] that combines *portability* with *performance*, *transparency* and *expressiveness*. The goals of our design are threefold.

First, we want high execution speeds by keeping the instrumentation overhead to a minimum. Existing instrumentation platforms for JavaScript are either implemented at the level of source code instrumentation [9, 10] or within an interpreter at the level of the abstract syntax tree before the input program is further optimized for high execution speeds [11]. JavaScript engines, however, typically compile their input programs at runtime to an intermediate representation to then interpret this representation for faster execution. Thus, we envision that an instrumentation platform operating at the level of this intermediate representation interpretation will provide high execution speeds.

Second, we want to provide an instrumentation interface which is familiar to developers but it enables functionalities from the interpreter that are hard to do with source code instrumentation.

Third, we want to enable our instrumentation platform to run on WebAssembly execution engines. WebAssembly is a compilation target designed for portable applications that aim at operating at high speeds in a secure sandboxed environment. One of the hosts that enables the execution of WebAssembly is JavaScript, such that it enables the execution of computationally expensive programs at higher and more predictable performance. Thus we would leverage on the guarantees of WebAssembly to have a fast, open, tamper-proof, transparent and portable interpreter-based instrumentation platform.

1.3 Roadmap

Chapter 2 provides the reader with the needed background on the existing state of the art on instrumentation platforms for JavaScript and the techniques for implementing them. It further provides a comparison of these techniques on different scales, after which we revisit the problem statement as motivation for our work.

Chapter 3 covers our approach, a portable interpreter-based instrumentation platform we name BoaSpect. We cover BoaSpect's architecture, followed by an explanation of its interface and usage. The end of the chapter covers the goals of BoaSpect which we later evaluate.

Chapter 4 discusses the implementation of BoaSpect, including the challenges we had to overcome such as exposing internals from the interpreter to the analysis operating at the level of source code.

Chapter 5 explains how we test JavaScript compatibility for BoaSpect and evaluates it with respect to the properties *transparency*, *performance*, *portability* and *expressiveness*.

In Chapter 6 we conclude this thesis and elaborate on future work.

Chapter 2

State of the Art and Motivation

Within the domain of computability, a well-established but unfulfilling theorem, known as Rice's theorem, states that non-trivial semantical properties of programs are undecidable [14]. This theorem carries the meaning that one cannot define a general algorithm that can decide on the behaviour of programs. Thus algorithms to answer interesting properties such as "Will this program terminate?" or "Will this program throw an error?" cannot be defined.

Regardless of the implications of Rice's theorem, developers use tools to improve the quality, security, performance and overall correctness of code. *Program analysis* is a discipline in computer science to automatically analyse the behaviour of programs against those properties. Program analysis can be done without executing a program, which is known as *static analysis*, or during runtime, known as *dynamic analysis*, or as a combination of both.

To further demonstrate how *static* and *dynamic analysis* work, we use the example JavaScript program shown in Listing 1. The program defines the function `fibonacci` which operates on numbers and the function `main` which is the program entry point. The `main` function prompts for external input *count* after which it sums the result of calling the `fibonacci` function for values over the range $(0, count]$. The program is syntactically correct, but we will show how a *static* and *dynamic analysis* could be done to further improve the program.

A static analysis tool may offer, for example, type inference checks. The function `main` includes two calls to the `fibonacci` function. The actual parameter passed is bound to an argument of type *number* during the first call. During the second call to *main*, however, the actual parameter will be bound to an argument of type *string* as the result of calling `prompt`. From the definition of the `fibonacci` function, however, we can derive that the function expects an argument of type *number.* We derived this information by noticing the operations applied on the formal parameter `n` are restricted to the arithmetic operations `n <= 1`, `n - 1`, and `n - 2`. With this information from the analysis, the developer might reconsider adjusting the second call or adjusting the definition of the *factorial* function to account for the possible type differences. The tool could also run a *variable liveness* analysis and determine that for all possible program paths, the variable `extra` in the body of the *main* function does not affect the program behaviour, as the program assigns values to the variable but does not use the variable in subsequent program statements. This information could inform the developer that there might be missing statements that would make use of this variable or that the variable its existence is redundant.

As dynamic analyses can check properties of the behaviour while running the program, examples of such program properties include memory usage and safe access. Consider a profiler which checks the memory usage through the execution of the program in Listing 1 as a concrete example of a dynamic analysis. For different input values, the analysis will point out that the majority of

```
function fibonacci(n) {
  if (n <= 1)
    return 1
  else
    return fibonacci(n-1) + fibonacci(n-2)
}

function main() {
  let count = parseInt(prompt());
  let total = 0;
  let extra = 0;
  while (count > 0) {
    total += fibonacci(count);
    extra += total + count;
    count -= 1;
  };
  console.log(`Total = ${total}`);
  console.log(`Fib(5) = ${fibonacci(prompt())}`);
}
```

Listing 1: A JavaScript program for which a static analysis could infer incorrect function usage while a dynamic analysis could suggest program optimizations.

execution time and use of memory for the call-stack is spent on the `fibonacci` function rather than the `main` function. Using this information, the developer might decide to reimplement the function or remain cautious with the number of calls to the function if a reimplementation is out of the developer's control. Furthermore, a dynamic program trace profiling function calls will point out that for ever-larger growing inputs, the number of function calls grows exponentially while the output value is determined exclusively by the input value. Based on these insights, the developer may decide to employ *memoization* optimization, in which function results are cached, to lower these computational expenses. For larger code bases, such insights could help developers pinpoint the program points where to put optimization efforts.

2.1 Program Analysis

Static program analysis attempts to uncover the runtime properties of the program while abstaining from concrete evaluation, through means of approximating the program evaluation [4]. The applications of static analysis vary from informing optimizing compilers on possible program transformations that preserve the program semantics but improve the execution performance to analyses that prove the program correctness [5]. Other applications for static analysis reason about the code structure to enrich the developer experience by reporting on program modification suggestions. Such suggestions include, but are not limited to, refactorings which alter the program structurally without altering it semantically [15] or code completions based on the available code [16]. Static analyses are different for dynamic languages such as JavaScript since they will approximate the behaviour as it is not computationally possible to have a sound static analysis.

The dynamic process that a program describes can surpass the approximation capabilities of static analysis [6]. These limitations become ever more apparent for programming languages that are of dynamic nature, that involve dynamic typing, dynamic code generation and evaluation or support of incomplete program definitions that are further defined during their execution. Thus, in contrast to static analysis, a dynamic program analysis allows more in-depth verification or enforcement of program runtime properties by augmenting the process that stems from the program with an additional process that gathers the runtime-behaviour information of the program during a concrete evaluation. Optionally, during the trace of the execution, the dynamic analysis could alter the program execution if the analysis deems the program execution behaviour incorrect.

Existing dynamic analysis tools offer developers means to instrument programs to reason about runtime information such as code coverage achieved by a program run for a test suite [17, 18], tracing the memory allocation, leakage or corruption of programs [19, 20], enforcing the program to uphold semantical guarantees through security analyses [21, 22], tools such as GDB enable tracing the program execution [23] and tools like VALGRIND allow profiling performance [19] for a multitude of programming languages.

2.1.1 Challenges for Dynamic Analyses of JavaScript

One particular language that gained interest from the research community for program analysis is JavaScript.

JavaScript is an interpreted dynamically typed object-oriented programming language, which has been adopted on several platforms. These platforms include web browsers such as Mozilla's Firefox [24] and Google's Chrome [25] or server backend infrastructures using NodeJS [26]. With the widespread adoption of the language, the core semantics and syntax are standardised by the ECMA association in the ECMAScript specification [27].

As JavaScript is an interpreted language, wherever the code must be executed a JavaScript interpreter is responsible for executing the statements that make up the JavaScript program. These JavaScript interpreters are typically referred to as JavaScript *execution engines*. The execution engines, which closely follow the ECMAScript standard, are then embedded in host environments that extend upon the ECMAScript specification to allow the interaction of JavaScript programs with their host environment. An example of a set of standardised operations that extends on the ECMAScript specification is defined as the WHATWG Standards [28], which defines the JavaScript web standards to interact with entities such as filesystems or interactions with the webpage Document Object Model (DOM).

This thesis focuses on dynamic analysis for JavaScript, for which we briefly discuss the implications the language has on applying program analysis for JavaScript programs. The language is highly dynamic in nature, making it less suitable for static analysis but a good fit for dynamic analysis [29]. Some high-dynamic features that are extensively used include but are not limited to [3]:

- *Dynamic typing*
  Since JavaScript is dynamically typed, the runtime values in the program are bound to a type. This is in contrast to statically typed languages in which it is the variables that are bound to a type. This feature makes it hard for static analyses to reason about the program behaviour as it further increases the search space in which the static analysis tools look to reason about the analysis [30].

- *Dynamic code generation and evaluation*
  JavaScript features dynamic code evaluation through the `eval` function, which is a built-in function that accepts code as a string and interprets this code in the context where the `eval` function is called. Whereas the use of the `eval` function is considered a bad practice [31] and thus is further discouraged to use, dynamically loading or generating new code is possible in other ways such as using the `Function` constructor or the use of `<script>`-tags in webpages that load code from third parties over the network. This feature

makes it impossible for static analyses to reason about the complete program, limiting the knowledge with which reasoning can be performed.

- *Object layout changes*
  At runtime, JavaScript objects can be extended or deprived of properties at any point in time. This is possible not only with static properties, which are denoted using the dot notation such as "`object.property`", but also using computed properties using the bracket notation such as "`object[compute()]`", addressing the need for code to be evaluated *dynamically* to determine what properties may affect which objects. This is in contrast with statically typed languages that typically require developers to declare the layout of compound data structure definitions before the code is executed, like `struct` definitions for the `C` or `Rust`. This dynamic feature further increases the search space for static analysis tools.

Despite the better fit for a dynamic analysis as it manages to trace a concrete execution overcoming the limits of static analysis, JavaScript still challenges dynamic analysis in several aspects [3]:

- *No crash philosophy*
  Starting and growing as a language within the context of web browsers, the language aims to hide a program *crash* from the web user. Exceptions that do occur are shown at developer consoles but do not interfere with other registered event handlers such that they remain responsive. Furthermore, the language operations remain rather permissive in nonsensical operations such as dividing a number by a string, yielding the value `NaN` (which stands for "not a number") rather than crashing the program. This behavior makes it difficult still for dynamic analyses to tell apart intensional from unintentional program behaviour.

- *Non-determinism*
  As the JavaScript interpreter is embedded in a host, the interaction between the host and the running program introduces the main source of non-determinism. This may come from user interaction with a webpage within browsers to additional network requests that return results that trigger further events.

  Given that dynamic analyses trace a concrete program execution, this high source of non-determinism challenges dynamic analyses to identify the possible order of events such that certain code is executed which may be of particular interest to the analysis at hand.

Yet, a variety of dynamic analysis tools have been built for JavaScript. These dynamic analysis tools range from pinpointing application security issues [22, 32], to lowering the number of code smells in code [33, 34] or addressing dynamic race detection [35]. Other dynamic analysis tools may further uncover code that gives rise to poor performance characteristics by execution engine unfriendly code [36, 37, 38].

2.2 Instrumentation-based solutions for Dynamic Analysis of JavaScript programs

Within the domain of computer science, the collection of program behaviour is referred to as the act of *instrumentation*, where one instruments certain aspects of the process. A metaphor for *instrumentation* from electrical engineering would be the act of inserting "probes" in certain parts of the electronic circuit to listen to signals and connect those probes to *instruments* to analyze those signals.

Similarly in the field of computer science, *instrumentation* is achieved through injecting additional code in the program under instrumentation which will collect information about the running process. The process of injecting a specific dynamic analysis into a process is automated through the use of *dynamic analysis tools*, which abstract from the specific process and provide means for developers to put the focus on the analysis.

While an analysis may be implemented for one specific purpose, we focus in this work on the use of an *instrumentation platform* as a tool which supports analysis developers with the creation of a dynamic analysis. An instrumentation platform abstracts from both a specific analysis and a specific program by supporting the developers in both implementing the analysis and injecting it into a program. For JavaScript programs, several approaches to implementing such an instrumentation platform have been developed [39, 10, 7, 11, 8, 9]. Marchand de Kerchove et al. [40] describe three main techniques for dynamic analysis instrumentation platforms for JavaScript, *runtime*, *source-code* and *meta-circular* instrumentation.

We further describe each of the techniques in a dedicated subsection, while we distinguish the runtime instrumentation technique into *interpreter*-based and *abstract syntax tree*-based instrumentation.

2.2.1 Interpreter-Based Instrumentation

As JavaScript programs are interpreted, their runtime environment can be altered to include instrumentation instructions [41, 42]. A JavaScript interpreter typically features a just-in-time (JIT) compiler to gain performance, which compiles JavaScript to an intermediate representation (IR) language for further execution.

An approach to enable the collection of runtime information would be to modify the interpreter to include additional operations during the execution of the JIT-compiled IR language. This approach, however, comes with certain challenges for the instrumentation developer.

First, it requires a solid understanding of the inner workings of the interpreter to avoid the violation of JavaScript semantics. Second, this modification requires much engineering effort as the JavaScript interpreters are often large and complex software projects as they attempt to remain faithful to the JavaScript semantics defined by the ECMAScript standards. In addition to its faithfulness, the interpreters tend to include several additional software components to optimize the interpretation of the JavaScript programs to maximize the execution performance of the input program for the host, such as the aforementioned JIT compiler. Third, the JavaScript language follows an ever-growing language specification thus requiring the instrumentation platform developers to adopt the instrumentation components to the evolving execution engine. And fourth, an additional disadvantage of implementing instrumentation directly in the interpreter is that this restricts the deployment of the instrumentation exclusively to where the engine can be deployed.

On the other hand, this approach comes with the advantage that the instrumentation executes at the level of the interpreter. This benefits the instrumentation in more easily accessing the available interpretation information.

2.2.2 Abstract syntax Tree (AST) Instrumentation

Abstract syntax tree instrumentation is the technique of performing modifications to the program abstract syntax tree (AST). The AST is a tree-like representation of the input program, which is then passed on to a component in the interpreter that further interprets it. We consider AST instrumentation as a second instrumentation technique which also takes place within the JavaScript interpreter.

A well-known AST-based instrumentation platform for JavaScript is NodeProf [11]. Their implementation is built on top of Graal.JS [43], a JavaScript interpreter designed to run on the GraalVM [44] language runtime. They rely on the GraalVM high-performance optimizations for the AST interpretation by performing partial evaluation on the AST nodes by specializing them for runtime assumptions, which for invalid specializations is deoptimized [45]. NodeProf uses the infrastructure of GraalJS to extend the AST with events that perform the analysis by wrapping the nodes with hooks to the analysis instructions. By doing so NodeProf is leveraging the efforts that further optimize the AST nodes during their execution resulting in high-performance benefits while reasoning about JavaScript statements rather than requiring extending the interpreter and maintaining the original program structure.

This brings the additional benefit that the input program remains more easily separated from the analysis, and as such, we say that the analysis is transparent to the program.

By design of the polyglot runtime of the GraalVM interpreter on top of which GraalJS is built, the analysis developer can express their analysis in either JavaScript or Java, the host language for the JavaScript execution engine. As far as we can determine, NodeProf cannot be used for client-side JavaScript code.

2.2.3 Source Code Instrumentation

Source code instrumentation systematically rewrites the input program its source code to transform every single code location with an event of interest, into a function call that invokes the analysis with the respective properties of that event while maintaining equal semantics of the input program.

Examples of such source code rewriting instrumentation platforms are Jalangi2 [10, 46] and Aran [9, 12]. These platforms perform their transformation by relying on existing AST transformation tools such as Esprima [47], Acorn [48] and Escodegen [49].

From the point of the user of an instrumentation platform user, we would define an analysis specification to be provided to the platform. For example, the dynamic analysis explained in the introduction of this chapter had two goals. The first goal was to model the use of the call stack by tracking the number of function applications per function. The second of its goals was to model the relationship between the function calls, in what arguments are provided to the callee and what results are returned to the caller.

An example of a dynamic analysis as it would be developed for Aran, an instrumentation platform, is provided in Listing 2, which is in particular the implementation of the dynamic analysis motivated for the input program in Listing 1. In Listing 2 the analysis comprises an `advice`, specified as a JavaScript object that describes procedurally what additional operations should take place for what particular program operations, in this case for function applications.

The analysis instructs the instrumentation platform to intercept function applications by providing the `apply` trap function with a signature of four arguments, the function as a first-class value, the `this` context bound to the function at call-time, the arguments and a serial identification. The body of the `apply` trap declares the additional operations that are required to take place. The expected value of the function is called through a reflective `Reflect.apply` call, after which the analysis code performs additional instructions before yielding the result value and control to the instrumentation platform. The rest of the analysis expands the objects `invocations` and `callResults` with the number of function calls and input-output combinations respectively as dictionaries where the keys are the function name and the values are the collected information.

The transformation output for the input program in Listing 1 where the analysis in Listing 2 hooks into function applications is partially depicted for the `fibonacci` function in Listing 3. The rewritten code shows how different operations of the target program are rewritten. The top-level

```
global.invocations = {};
global.callResults = {};

global.ADVICE = {
  apply: (f, t, xs, serial) => {
    const result = Reflect.apply(f, t, xs);

    const fn = f.name;
    invocations[fn] = (invocations[fn] || 0) + 1;
    const callMap = callResults[fn] || {};
    callMap[xs] = new Set([...(callMap[xs] || []), result]);
    callResults[fn] = callMap;
    return result;
  },
};
```

Listing 2: An example Aran-compatible analysis specification for profiling function invocations and tracking caching opportunities.

```
Reflect.set(
  ADVICE.builtins["global"],
  "fibonacci",
  function ($n) {
    if ($n <= 1)
      return 1;
    else
      return (
        ADVICE.apply($fibonacci, void 0, [$n - 1], 12) +
          ADVICE.apply($fibonacci, void 0, [$n - 2], 13)
      );
});
```

Listing 3: The transformed output for the `fibonacci` function in Listing 1 if it were instrumented by source code instrumentation for the analysis in Listing 2.

call to `Reflect.set` performs the function definition in the global scope of the `"fibonacci"` function. The notable change within the function body definition is its else-branch return statement, in which the function applications `fibonacci(n-1)` and `fibonacci(n-2)` from Listing 1 are rewritten call into the advice-specified `apply` trap.

The results retrieved from the analysis are reported in Listing 4, which aligns with the aforementioned dynamic properties of the program. The `invocations` dictionary shows a particular high count of seven million applications of the function *fibonacci*, and the `callResults` dictionary shows that for these seven million calls to the *fibonacci* function the arguments were within the range $[0, 30]$ and would benefit highly from caching the intermediate results to improve the program performance as the bound results per concrete argument did not comprise multiple values.

The difference between source-code instrumentation with an interpreter- or AST-based instrumentation is twofold. On one hand, the inclusion of the instrumentation platform and analysis with the input program requires a *transformation* step in which the input program is transformed into a representation, e.g. AST-based representation, which is then augmented, resulting in a program with both the analysis code and its original code. Note that this step thus can take place ahead of time. This is different from interpreter-based instrumentation as interpreter-based instrumentation performs its instrumentation during the execution of the statements, which does not depend on a transformation step.

```
{
    invocations: { parseInt: 1, fibonacci: 7049137, get: 2, log: 2, main: 1 },
    callResults: {
        parseInt: { '30': Set(1) { 30 } },
        fibonacci: {
            '0': Set(1) { 1 }, '1': Set(1) { 1 }, '2': Set(1) { 2 },
            '3': Set(1) { 3 }, '4': Set(1) { 5 }, '5': Set(1) { 8 },
            '6': Set(1) { 13 }, '7': Set(1) { 21 }, '8': Set(1) { 34 },
            '9': Set(1) { 55 }, '10': Set(1) { 89 }, '11': Set(1) { 144 },
            '12': Set(1) { 233 }, '13': Set(1) { 377 }, '14': Set(1) { 610 },
            '15': Set(1) { 987 }, '16': Set(1) { 1597 }, '17': Set(1) { 2584 },
            '18': Set(1) { 4181 }, '19': Set(1) { 6765 }, '20': Set(1) { 10946 },
            '21': Set(1) { 17711 }, '22': Set(1) { 28657 }, '23': Set(1) { 46368 },
            '24': Set(1) { 75025 }, '25': Set(1) { 121393 }, '26': Set(1) { 196418 },
            '27': Set(1) { 317811 }, '28': Set(1) { 514229 }, '29': Set(1) { 832040 },
            '30': Set(1) { 1346269 }
        },
        get: {
            '[object console],log,[object console]': Set(1) { [Function: log] }
        },
        log: {
            'Total = 3524576': Set(1) { undefined },
            'Fib(5) = 8': Set(1) { undefined },
        },
        main: { '': Set(1) { undefined } },
    },
}
```

Listing 4: The reported results from instrumenting the program in Listing 2 with the analysis specification from Listing 1. The conclusion can be drawn that most function invocations happen for the function `fibonacci` and the results from calling the function for different values give rise to memoization.

On the other hand, in source code instrumentation, the analysis and instrumentation platform both execute in the same environment. This makes it difficult to keep analysis transparent, and it thus may compromise security. This requires careful separation of these environments for source code instrumentation, while interpreter-based instrumentation can reason in different environment layers by design. This environment distinction becomes even more apparent if the interpreter is designed in a different language.

2.2.4 Meta Circular Evaluation Instrumentation

A final technique we cover is meta-circular evaluation instrumentation, in which a JavaScript interpreter developed in JavaScript is extended with means to instrument the input program. Marchand de Kerchove et al. [7] developed such a meta-circular interpreter, Narcissus. It has been used to provide a meta-circular instrumentation platform [40], which has then been adopted for prototyping analyses such as Linvail [9] or Multiple Facets for Dynamic Information Flow [50]. Other such meta-circular interpreters are Js.js [51] which is the Firefox JavaScript interpreter that has been compiled from C++ to JavaScript using the Emscriptem compiler and the Photon prototype interpreter [8].

The advantage of this technique is that it enables full control of the runtime with the ease of transparency while not requiring the alteration of the JavaScript engine. The disadvantage of Meta Circular Evaluation Instrumentation is the high incurred performance overhead [8].

2.3 Instrumentation Techniques – One Size Does Not Fit All

The goal of an instrumentation platform is to provide instrumentation means to an analysis developer to enable the instrumentation of the input program. With the multitude of techniques, however, the notion of the "best-suited technique" depends on different factors. In what follows we identified four desired characteristics, which we cover as instrumentation platform properties, to analyse current instrumentation platform techniques.

Transparency
By default, the input program should, during its execution, remain unaware of the fact that it is being instrumented [52, 53]. In other words, this property requires that in the case that the input values and timing of events for a JavaScript program execution were constant, no execution differences should appear in the presence of the analysis injected by the instrumentation platform.

The transparency property may directly affect the correctness of the conclusions drawn from the analysis, as the input program might misbehave if the analysis unintentionally changes the input program semantics, which in turn would affect the conclusions drawn by the analysis. Such an example of an incorrect conclusion would be for a malicious input program to prevent taking the malicious path in its code if it were to suspect that it is instrumented, effectively rendering a security analysis incorrect. Another example would be an analysis that attempts to report bugs in the program. If in some manner the input program would behave differently at runtime, the bug report might report on false positives, ie. reporting on bugs that would not be able to occur if the program were uninstrumented, or false negatives, ie. the report would fail to uncover a bug as the instrumentation prevents the bug from occurring. Such an instrumentation platform would harm the value of the analysis for the developers that try to improve their program. This type of bug, in which the act of inspecting a bug influences the occurrence of the bug is known as a *Heisenbug* [54].

Note, however, that an analysis may intentionally alter the execution of the input program. For example, a taint analysis may halt program execution when there is an unauthorized flow of information to a sink [3].

Performance
As mentioned before, dynamic analysis incurs performance overhead since execution is being instrumented. The choice of the instrumentation platform implementation technique will affect the overhead there is to pay for instrumenting the target program. Such overhead should not impact the user experience for the instrumentation platform depending on the setting in which the analysis takes place.

Given the fact that JavaScript often tends to impact the responsiveness of a front-end user interface, or that its event-based execution model depends on the timing of events, a slowdown of the input program may impact the behaviour of the program, potentially causing the aforementioned *heisenbugs*. This in return could hamper the information that the analysis strives to collect. For example, with JavaScript's event-driven nature, a variety of applications exist that rely on the timing of events, which has been shown to lead to applications that contain race conditions [55]. Dynamic analyses for JavaScript may then attempt to uncover the sequence of events that leads to this race condition which would violate the domain constraints [35, 56, 57].

Note that in this work we only focus on the performance of the instrumentation platform itself, not on the analysis built on top.

Portability
Given the diversity of JavaScript environments, it is important that analyses are portable across concrete execution engine implementations. Examples of JavaScript environments range from different browsers to mobile applications or server back-end infrastructure. For cross-environment instrumentation needs, a technique that is tied to a single environment is then less *portable*, making the technique less desirable.

Expressiveness
The set of operations that the instrumentation platform exposes to the analysis developer impacts which kind of analysis one can write on it. Whereas the analysis is in control of *what* information is collected at runtime, it is the instrumentation platform that is in control of the kind of operations from the execution runtime that is exposed to the analysis developer. Depending on the implementation technique by the instrumentation platform, for example on the JavaScript source code layer or through an interface provided by the JavaScript runtime host, the information on the runtime behaviour offered to the analysis developer will vary.

2.4 Comparing Different Instrumentation Techniques

In this section, we compare state of the art in instrumentation platforms with respect to the different properties discussed in Section 2.3 qualitatively per property.

```
1 const dynamic = prompt(); // non-deterministic string input
2 const object = {
3     [dynamic]: () => {"function body";},
4 };
5 assert(object[dynamic].toString() === '() => {"function body";}');
6 assert(object[dynamic].name === dynamic);
```

Listing 5: A JavaScript program where the assertions rely on a dynamically bound function name and the function source code.

2.4.1 Transparency

Transparency is most easily achieved through techniques that perform their instrumentation on a level of abstraction lower than the source code that is instrumented. For example, source-to-source code transformation requires more in-depth engineering to ensure the presence of the instrumentation and analysis remain hidden whenever the input program performs reflective operations.

Let us show this through a concrete JavaScript program using 2 reflective operations shown in Listing 5. The method `Function.prototype.toString` ensures that all user-defined functions inherit this method which yields its caller a textual representation of the bound function. The property `Function.prototype.name` yields the function name, a property that can be assigned dynamically to the function object.

Overall, the program in Listing 5 retrieves a string from a non-deterministic source on line 1, after which it creates an object literal in which it binds the retrieved string dynamically to an anonymous function on line 2. The next two assertions on lines 5 and 6 take place, one in which a call to the anonymous function method `toString` should yield the string how it has been defined, the other on that its property `name` should yield the dynamically bound name.

For the function properties asserted by Listing 5 and other cases, Christophe et al. [9] explain why these assertions would be violated for their source code transformation instrumentation platform Aran [12]. For the method `Function.toString` the function body is further transformed by the instrumentation platform to include hooks that call the traps defined in the analysis advice. This source code transformation results in the input program being able to expose the function body with a single `Function.toString` call. For the `Function.prototype.name` property Christophe et al. [9] describe that for Aran [12] this property is determined by a static analysis, thus being unable to precompute this value statically before the program is running. Furthermore, Sun et al. [11] continue on these limitations stating that for source code instrumentation an additional transparency-breaking concern is that the stack layout, accessible within JavaScript through `arguments.callee.name`, is polluted with advice code.

Transparency can be achieved for the other instrumentation techniques as they operate on a lower level of abstraction. For AST interpretation it relies on the parsed program tree, for (meta-circular) interpreter-based instrumentation the platforms rely on the (virtualised) interpreter. These differences allow these other techniques to more easily avoid detection in terms of the input program and its reflective capabilities.

2.4.2 Performance

The performance of an instrumentation platform is directly related to the opportunity for optimization by the host execution environment, as the operations defined by the input program should now be augmented with operations that enable the runtime analysis to monitor the execution of those operations. The instrumentation platform decides where and how these *monitoring* operations are added, which in turn affects the performance penalty to pay.

For source code instrumentation, the performance penalty comes from the additional operations that are added during the translation phase that communicate with the analysis through the use of callbacks. By the design of the source code transformation approach, the instrumentation platform adds the analysis on the level of source code and hooks into the traps through callbacks weaved in the input program. The performance penalty for this technique comes not only from the additional function calls, but in turn, it results in the loss of potential optimizations performed by the JavaScript execution engine. The opportunity for the JIT-compiled code to remain efficient may further break as the additional instrumentation operations may obstruct constructs that give rise to efficient IR code. For example, taking back the example shown in Listing 1, the "+" operator in the `fibonacci` function defined in might be optimized to directly execute an assembly `ADD` instruction on two numbers as its operands. If the analysis decides to instruct the instrumentation platform to instrument binary operations such as the use of the "+" operator, as shown in Listing 3, it could further hinder the opportunity to inline the addition as an assembly `ADD` instruction due to the pollution of binary instrumentation hooks. In addition to the performance penalty of the additional operations, for source code transformation it is so that the instrumentation code remains present during the whole lifetime of the JavaScript program. Analyses implemented on top of source code instrumentation platforms for JavaScript such as Aran and Linvail have reported slowdown factors up to three orders of magnitude [9, 36].

For the AST instrumentation, these JavaScript optimizations can remain largely in place as the bulk of the instrumentation platform operates within the host environment. In the case of NodeProf, it is possible to gain performance thanks to the underlying GraalVM [11]. In addition to this, the NodeProf authors enable the analysis developer to maintain their analysis developed in Java, which allows for further optimizations. Sun et al. [11] elaborate on NodeProf performing 2–3 orders of magnitude better than source code instrumentation platforms for JavaScript-based analyses and another order of magnitude faster for Java-based analyses, both monitored after a warmup period in which they let their engine reach a steady state.

For meta-circular instrumentation Marchand de Kerchove et al. [40] show that the Js.js interpreter reports slowdown factors up to 4 orders of magnitude.

2.4.3 Portability

The portability property, which concerns how easily the instrumented application can adapt to a variety of host execution environments, is determined by the nature of the instrumentation platform host language.

As mentioned before, JavaScript input programs tend to run in different environments, which makes it a desirable property for its instrumented counterpart to be able to run in the same environments too. The instrumentation technique, however, comes with its own environment requirements. For example, with interpreter-based instrumentation, the necessary adjustments must be made to the JavaScript execution engine to make the technique possible, the same goes for abstract syntax tree instrumentation. This would require modifications in the web browser, possibly through a modular component system or by shipping the instrumentation platform directly with the browser.

For the source code and meta-circular instrumentation techniques, however, it holds that they, by their design, expect the host environment to be a JavaScript execution engine. These approaches make it so that these techniques are by design portable.

	Meta-circular	Source-code	AST	Interpreter
Transparency	✓	✗	✓	✓
Performance	slower	slow	fast	faster
Portability	✓	✓	✗	✗
Expressiveness	✓	✗	✓	✓
Related work	Narcissus [7], Photon [8]	Aran [9], Jalangi [10]	NodeProf [11]	*Analysis specific* [41, 42]

Table 2.1: A comparison table for the different instrumentation platform properties, per row, for different instrumentation platform techniques, per column.

2.4.4 Expressiveness

The expressiveness property quantifies the ability of the analysis developer to implement their analysis by relying on what the instrumentation platform exposes through its API. The choice for the instrumentation platform in what it exposes through its API is limited by the extent to which it has access to collect the program runtime information which is inherently tied to the level of abstraction on which the instrumentation platform is implemented.

For example, source code instrumentation platforms are required to provide solutions to program properties that are not accessible by the default JavaScript runtime. For Aran and Jalangi either the analysis author or the instrumentation platform is required to virtualise program properties such as the environments for shadow execution or localising analysis state per statement through additional runtime dictionaries [9, 11]. On the contrary, NodeProf showcases they can more easily localise analysis state at certain program points of interest [11] for the additional analysis developer convenience. Interpreter-based instrumentation and AST instrumentation can provide more access to all runtime information, including access to built-in operations of the interpreter [3].

Conclusion

Table 2.1 summarizes the instrumentation platform properties with their benefits and disadvantages that we covered in Section 2.4. Given this table, we observe that it seems that all instrumentation techniques fall short on one or more of these properties and that there is no "one size fits all" approach. On one hand, *AST* and *interpreter* instrumentation impose little overhead, are well-suited for transparency and yield high expressiveness, but they suffer from their lack of portability making them less appealing for instrumentation that should cover a variety of platforms. On the other, *Source-code* instrumentation excels in its portability, but this seems to come mostly at the cost of suffering in all other aspects. *Meta-circular* instrumentation successfully combines portability with transparency and expressiveness, but it severely suffers from performance.

In this work, we investigate a novel instrumentation platform that aims to combine these properties. The most challenging tradeoff between the platforms is between performance and portability. Given the nature and uses of JavaScript, however, these are both essential for JavaScript programs. Its interpreted nature allows for the language to remain platform agnostic, requiring its tools that should run across different devices to adopt this flexible nature too. As explained in Section 2.3, changes in the execution performance can impact the semantical execution of the programs, up to the point where it can affect the responsiveness of applications for JavaScript programs that are responsible for rendering a user interface, as such, performance is also critical.

Chapter 3

BoaSpect: A portable interpreter-based instrumentation platform for JavaScript

This chapter introduces the reader to our approach with which we aim to implement an instrumentation platform that offers *portability*, *performance*, *transparency* and *expressiveness*. We motivate our approach in Section 3.1. In Section 3.2 we discuss the architecture overview of our implementation while explaining how an analysis developer would use our instrumentation platform to implement an analysis. In Section 3.3 we further motivate our research design considerations and our implementation design choices. Section 3.4 elaborates on the research questions we aim to answer in the rest of this work.

3.1 Approach

In this work, we explore a novel approach to implementing an instrumentation platform for JavaScript that is interpreter-based but overcomes the portability issues discussed in the previous chapter by leveraging on WebAssembly [13].

WebAssembly is a binary instruction format designed for predictable and high-performing code used as a compilation target by languages such as Rust, C and C++. By targeting WebAssembly, we aim for a more portable approach compared to *interpreter* or *AST* instrumentation, which are the most suitable approaches in terms of performance, transparency and expressiveness.

Central to our approach is the choice of a suitable JavaScript interpreter and the means of extending this interpreter with support for instrumentation. A JavaScript interpreter developed in JavaScript is unsuitable as its performance would be equivalent to the *meta-circular* instrumentation approach, which is known to suffer from its performance overhead. Departuring from an AST-based interpreter approach like NodeProf is also not possible as their infrastructure is built on top of the GraalVM which currently does not support JavaScript or WebAssembly as its runtime environment. Table 3.1 provides an overview of existing JavaScript engines that can compile to WebAssembly[1]. From this table, we build our work on the JavaScript engine Boa

[1]Part of the list presented in Table 3.1 comes from `https://wapm.io`, which is a WebAssembly package manager for standalone programs and libraries.

JavaScript Engine	Most relevant setting	Host language
SpiderMonkey	Firefox Web Browser	`C++`
JavaScriptCore	Safari Web Browser	`C++`
QuickJS	Embedding JavaScript	`C`
Duktape	Embedding JavaScript	`C`
Boa	Embedding JavaScript	`Rust`

Table 3.1: A listing of known JavaScript engines with support to compile to WebAssembly.

because it has the potential of being performant and compiles to WebAssembly which we further motivate in Section 3.3. We name our instrumentation platform for developing dynamic analyses built on Boa, BoaSpect. For our approach, it is crucial that the modifications we perform do not introduce any cross-platform-breaking changes to the project. This restriction allows one to run the instrumentation platform within any host environment where WebAssembly can be executed. The presence of a WebAssembly execution engine technically does not restrict our approach exclusively to JavaScript environments, as long as the host environment contains a WebAssembly execution engine.

This approach differs from existing *meta-circular* instrumentation platforms as in our approach the interpretation will not take place within the JavaScript execution engine but on the WebAssembly execution engine. In contrast to existing *AST*- and *interpreter*-based instrumentation platforms, our approach maximizes portability by leveraging on WebAssembly as a supported runtime environment for the interpreter. Furthermore, this approach differs from *source-code* instrumentation in that this approach does not compile the input program into an instrumented variant which can further add to the performance overhead, rather it performs its instrumentation during the evaluation of the program which takes place in a host JavaScript environment.

3.2 BoaSpect's Architecture

The platform architecture of our approach is depicted in Figure 3.1. This figure shows the different components that enable dynamic analysis.

BoaSpect takes as input the JavaScript program to analyse and the analysis source code is also written in JavaScript. More concretely, the analysis is written using the Aran API, namely an advice specification on a number of hooks the analysis requires to function.

BoaSpect, the instrumentation platform is the compilation result of the modified Boa engine. The compilation of the instrumentation platform can target either WebAssembly to further enhance the portability or it may target a system-level executable to execute directly on the CPU.

Next, we further explain the technical details of our instrumentation approach, from the perspective of how an analysis developer would interact with BoaSpect and describe the fundamental aspects of its API. In Chapter 4 we describe the most relevant details of BoaSpect from the perspective of implementing the instrumentation platform.

3.2.1 Analysis API Design

BoaSpect allows the implementation of analyses at the language operation level, similar to other dynamic analysis tools such as Jalangi and Aran. It offers analysis developers a set of *trap functions* that allow them to control the execution of the input program. The behaviour of

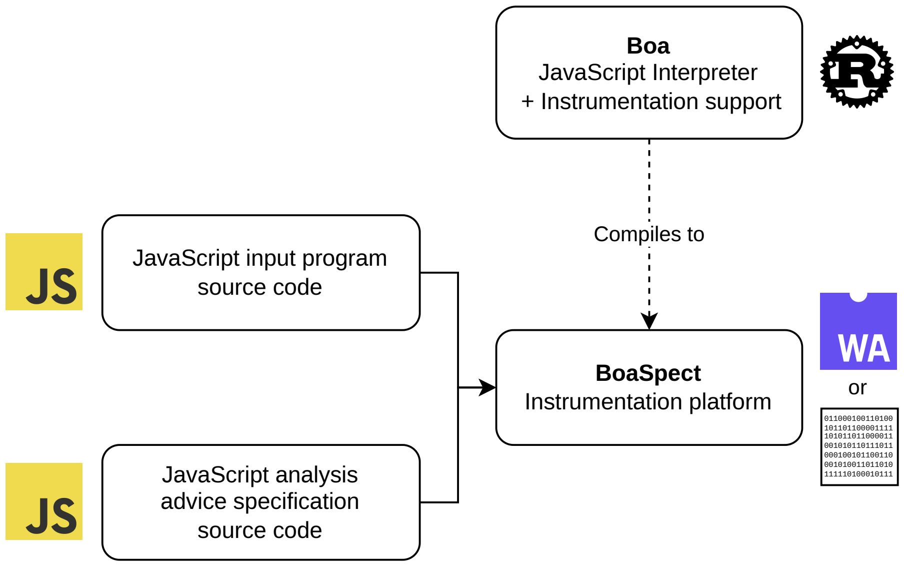

Boa
JavaScript Interpreter
+ Instrumentation support
Compiles to
JS
JavaScript input program source code
JS
JavaScript analysis advice specification source code
BoaSpect
Instrumentation platform
WA
or

Figure 3.1: A high-level overview of the architecture of our approach in which we extend the JavaScript engine Boa to account for support of instrumentation. The platform accepts two JavaScript programs. One is the JavaScript input program which is the application which is subject to a dynamic analysis. The other input program is an analysis advice specification which is compatible with the Aran API implementing the hooks to which the interpreter will call back to complete the instrumentation.

the dynamic analysis is expressed by overriding those functions. The collection of all these *trap functions* is combined as a set of methods of an object which makes up a single *advice* specification for the instrumentation platform.

We illustrate how developers can use the platform by means of a concrete analysis example displayed in Figure 3.2. The goal of the analysis in Figure 3.2 can be considered as the basis for a memory profiling analysis, by modelling the program call stack. To this end, we need to intercept all function calls that take place in the input program. The analysis implementation handles a JavaScript `Array` `callstack` as a stack data structure to push a call frame at the end of `callstack` when the input program performs a function call and pops this call frame off `callstack` when the input program returns from a function call. For this simplified example, we model the information per call frame to track the function itself.

The implementation of such an analysis for BoaSpect would be implemented as shown in Figure 3.2 in the file "advice.js". This file allows BoaSpect, upon calling the top-level anonymous function, to retrieve an *advice object* which comprises the aforementioned *advice* specification. An analysis developer familiar with the API of BoaSpect would be aware that only the implementation of the trap `apply` suffices for implementing the aforementioned analysis goal. The API of the trap `apply` is that the caller will be the instrumentation platform whenever a function call takes place for the input program. The arguments provided to this trap function are the called function, the *this* context bound to the function call and the arguments passed at call-time. The return value of the trap `apply` determines what value the intercepted call will evaluate to.

In the implementation of the trap `apply` in "advice.js" (lines 6–11), we can see that a function `preApply` is called (line 7), afterwards the outcome result of the intercepted function call is computed (line 8), next the function `postApply` is called (line 9) and the result is returned to the instrumentation platform. The definitions of `preApply` and `postApply` (lines 3–4) respectively grow and shrink the modeled callstack (line 2).

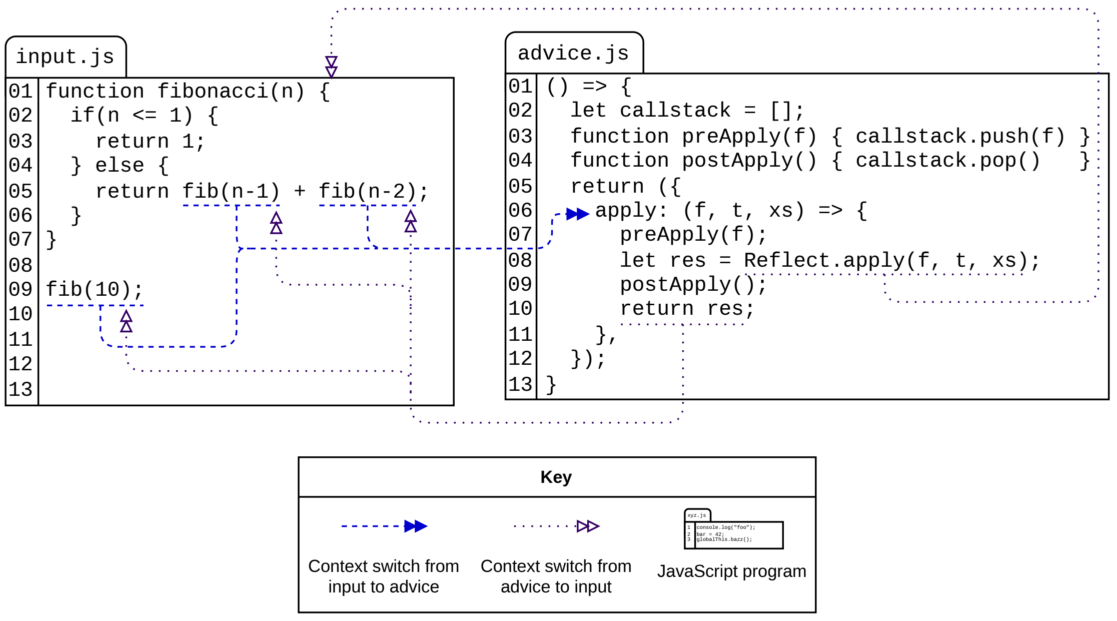
The input program "input.js" defines a recursive function `fibonacci` (`input.js`, lines 1–7) which calls itself twice inside the body. The final statement of the input program is a call to the `fibonacci` function (`input.js`, line 9).

The figure further illustrates, with arrows, the jumps from the program under analysis to the advice specification and back to the input program. This emph jumps that perform the *context switches* are how the instrumentation platform inserts the traps. As it is important that the instrumentation platform correctly performs the context switch from the input program to the advice, it is equally important that the instrumentation platform correctly returns from the advice to the input program, ie. to respect the *continuation.* These are generally possible in two manners. The first is the use of reflective means to continue the evaluation of the input program such as the implementation of `Reflect.apply` for apply traps. The second manner to yield control to the instrumentation platform is to terminate the evaluation of the *trap function* through the use of a `return` statement or the end of the function body.

With this example, we now understand how the implementation of only the `apply` trap suffices to implement our analysis. More generally, with this example, we touched upon some core concepts which we now cover in more detail.

The design of BoaSpect's advice API enables the analysis developer to take control over the operation within the body of the *trap function.* During the evaluation, upon reaching an operation of interest the instrumentation platform now hooks into the corresponding trap. As the evaluation now takes place within the body of the *trap function*, it is up until the point where the body terminates that control flow is yielded back to the interpreter to continue with the evaluation of the input program. Within the body of such a *trap function*, the analysis can perform both *introspection* and *intercession* on the input program.

The act of *introspection* entails the ability to "look inside" the program. This is possible


input.js
advice.js
Key
Context switch from input to advice
Context switch from advice to input
JavaScript program

```
function fibonacci(n) {
  if(n <= 1) {
    return 1;
  } else {
    return fib(n-1) + fib(n-2);
  }
}

fib(10);
```

```
() => {
  let callstack = [];
  function preApply(f) { callstack.push(f) }
  function postApply() { callstack.pop()   }
  return ({
    apply: (f, t, xs) => {
      preApply(f);
      let res = Reflect.apply(f, t, xs);
      postApply();
      return res;
    },
  });
}
```

Figure 3.2: An example input program "input.js" is subject to a dynamic analysis specified by the "advice.js" file. The blue dashed arrows show the control flow jumps the instrumentation platform needs to perform from the input program to the advice. The purple dotted arrows show the control flow jumps the instrumentation platform needs to perform from the advice to the input program.

through the ability to read from the values passed on to the *trap function*. For the example analysis implementation in Figure 3.2 this concerns the passed arguments to the *call* trap which can be kept in the analysis for further inspection.

The act of *intercession* entails the ability to alter the behaviour of the program. This is achieved through the design of the API, in which the return values of the *trap function* are often used as the result of the operation that is replaced with a call to the *trap*. Other possibilities are for the analysis to write additional properties to the values passed to the *trap functions* through *call-by-reference*, such as an object which can then be augmented with additional analysis properties. Not only are the *trap functions* capable of computing the expected outcome for the operations that are instrumented, the functions are *required* to perform these operations, as the control in this manner is passed from the interpreter to the analysis.

To enable a transparent analysis, the semantics of the input program should remain unaltered, making it a requirement to be able to compute the correct return value. Computing the correct return value requires a reflective framework or reflective means by the language that enable the analysis to "return back" to the level of the input program. For the example analysis implementation in Figure 3.2 the act of intercession by the analysis is possible through modifying the passed arguments to the *call* trap or returning values different from the excepted function call result. Transparency for the example analysis implementation in Figure 3.2 is achieved through returning the expected value through a call to `Reflect.apply`.

It is by wrapping the computation of these reflective operations with additional *before* or *after* qualifiers that the analysis can further introduce aspects such as *preconditions* or *postconditions*, depending on the requirements of the analysis. These *preconditions* and *postconditions* would represent requirements that must hold whenever the operations of the input program are performed.

As we aim for BoaSpect to be compatible with the API as specified by Aran, we thus more specifically mean to accept *advice* objects as analysis specifications. These advice objects can then be interpreted by BoaSpect to know which operations are of interest for the analysis. This compatibility requires BoaSpect to conform with both the naming convention decided by Aran and the signature of the corresponding *trap functions*. However, as the advice API of Aran as of July 2022 specifies a set of 26 *trap functions* the analysis developer can implement, we target a subset of those that serve as an entry point for defining a set of analyses to further evaluate BoaSpect as an instrumentation platform for this work. The subset of Aran-compatible *trap functions* that BoaSpect supports is summarized in Table 3.2 and further elaborated upon below. All traps enable intercession both by providing the values directly from the input program and returning them directly to the input program. The details of each *trap function* are further elaborated upon below.

`call`: The `call` trap defines the operations that should take place whenever a function is called at the level of the input program. The signature of this function follows the ECMAScript specification of the `Reflect.apply` method, which accepts three arguments. The actual parameters passed to the trap function are, from left to right, the *function* itself, the *this* value which refers to the context bound to the function, and an array containing the arguments passed to the called function. As this trap function should return the result of the input program call itself, an analysis which does not alter the return values from the function call could rely on the implementation of `Reflect.apply` to correctly yield expected value.

`get`: The `get` trap is invoked whenever a property of an object at the level of the input program is being accessed. The signature of the function consists of the *object* as the first argument and the *property* as the second. The type of the second argument would be

Trap name	Trap signature	JavaScript example	Instrumentation
call	(function, this, arguments)	`fibonacci(1)`	`call(fibonacci, this, [1])`
get	(object, key)	`human.name`	`get(human, "name")`
set	(object, key, value)	`human.name = "John"`	`set(human, "name", "John")`
write	(value, variable)	`name = "John";`	`write("John", "name")`
read	(value, variable)	`name;`	`read("John", "name")`
binary	(operator, left, right)	`1 + 2;`	`binary("+", 1, 2)`
unary	(operator, operand)	`!true;`	`unary("!", true)`
primitive	(value)	`1;`	`primitive(1)`
toPrimitive	(value, hint)	`-human;`	`toPrimitive(human, "number")`

Table 3.2: An overview of the *trap functions* supported by BoaSpect which can be defined by the analysis developer. The instrumentation column showcases the semantics of the call performed by the instrumentation platform applied to the sample statement in the JavaScript example column. The bottom row states the *trap function* `toPrimitive` which is not supported by Aran.

either a string for static property access using the dot notation or the dynamic value that is passed for dynamic property access using dynamic property access. In a transparent analysis implementation, the return value of this trap function will result in the value read by the accessing operation, thus the analysis programmer should rely on the use of the implementation of `Reflect.get` to correctly yield expected value.

`set`: Similarly to the setup for the `get` trap, the `set` trap is called whenever the input program modifies the value of an object property. The signature consists, next to the object and the property also of the value being set to the property. The analysis should include a call to the implementation of `Reflect.get` for correctly falling back to interpretation at the level of the input program.

`write`: The `write` trap is called whenever the input program writes a value to a variable. The signature consists of the value being set and the variable being written to, as a string. The return value of the trap function decides what value is assigned to the variable.

`read`: The `read` trap is invoked when the input program reads a variable. Similar to the `write` trap, the signature of the `read` trap consists of the value being read and the corresponding variable represented as a string. The return value of the trap function decides what value is read as a result of the read statement.

`binary`: The `binary` trap is called whenever the input program performs a binary operation on two operands. The set of JavaScript binary operators includes, but is not limited to, "+", "-", "/", "*", "%", "**", "&", "|", "^", "<<", ">>", ">>>", "==", "!=", "===", "!==", ">", ">=", "<", "<=", "in", and "instanceof". Since the operators are no first-class values in JavaScript, the signature consists of a string representation of the operator and the operands. As no reflective operation exists within JavaScript to perform binary operations, the platform addresses these shortcomings by providing means to perform the input program level binary operation. Concretely, this entails that the platform provides an interface for the analysis developer to hook into for computing the result of the operation that is intercepted at the level of the input program. The design of the interaction between the advice and the instrumentation platform with this trap enables the analysis developer to perform intercession as they are in control of the resulting value from the binary operation through the return value of the trap function.

`unary`: Similar to the `binary` trap, the `unary` trap is called whenever the input program performs a unary operator on an operand. The signature consists of the string representation of the operator and the operand. Similar to the requirements of trapping the binary operations, the platform hosts the means of performing the unary operation at the level of the input program.

`primitive`: The `primitive` trap is called whenever the input program creates a primitive value. This is one of the following values: `undefined`, `null`, `true`, `false`, `NaN`, a number, positive or negative infinity or a literal such as a string or a BigInt. The signature consists of the value being created. The return value of the trap enables the analysis to wrap the creation of primitive values before they are available to the input program.

`toPrimitive`: The `toPrimitive` trap is called whenever the input program tries to convert an object to a primitive value through the ECMAScript operation ToPrimitive. This operation is a JavaScript abstract operation, which is an operation happening at the level of the interpreter. The signature of this operation is based on the signature of the operation ToPrimitive. The first argument is the value that is being converted to a primitive value, and the second is a hint at the prefered type of the conversion, being one of “string”, “number” or “default”. The return value of the trap function will determine the evaluation of the *ToPrimitive* operation.

3.2.2 Supporting the Trap `toPrimitive`

We also included a novel *trap function* `toPrimitive` that BoaSpect supports which is not supported by Aran. In this section we further explain by means of a use case how the *trap function* `toPrimitive` can be useful.

The trap function is called whenever the execution of the input program triggers the ECMAScript operation ToPrimitive. This operation is one of ECMAScript’s abstract operations, therefore, it cannot be invoked at the level of source code but are exclusive for use within the interpreter.

The abstract operation `toPrimitive` is called by the interpreter when it needs to convert a value to a primitive value. The signature of the abstract operation, and our corresponding trap function, consists of two arguments. The first is the value that is being converted, the second is a *hint* to which primtive type the first argument should be converted, which is either `"string"`, `"number"` or `"default"`.

For example the JavaScript code “`42 - "0"`” will call *ToPrimitive* with the value `"0"` and the hint `"number"` in an attempt to convert `"0"` to a number to proceed with the subtraction.

Despite that only the interpreter can call the abstract operation *ToPrimitive*, JavaScript developers can assign a function to the property `Symbol.toPrimitive` for an object, which *ToPrimitive* will call to convert that object to a primitive value. The bound function should then follow the same signature as the abstract operation *ToPrimitive* and the return value should either be a primitive value, enabling developers to *override* this conversion with a custom implementation.

We argue that the overriding implementation of *ToPrimitive* should not perform any side effects. Not only does the default behaviour of *ToPrimitive* not perform any side effects, but overall the input program developer cannot tell when calls to *ToPrimitive*. Calls to *ToPrimitive* can be performed also by the host environment, such as web browsers that attempt to display a value on the screen will call *ToPrimitive* with the hint `"string"`.

Implementing `Symbol.toPrimitive` as a function with side effects, however, might indicate a bad coding practice or a malicious program. To this end, Brown et al. [58] demonstrate how an

```
1  /* File: node/lib/buffer.js */
2  function fill(val, start, end) {
3    // ...
4    // bound checks
5    if (start < 0 || end > this.length)
6      throw new RangeError("Out of range index");
7    if (end <= start)
8      return this;
9    // calls binding code
10   binding.fill(this, val, start)
11 }
```

Listing 6: The JavaScript implementation of the Buffer library of Node version 6.5.

```
1  /* File: node/src/node_buffer.cc */
2  void Fill(const FunctionCallbackInfo<Value>& args) {
3    size_t start = args[2]->Uint32Value();
4    size_t end = args[3]->Uint32Value();
5    size_t fill_length = end - start;
6    // ...
7    CHECK(fill_length + start <= ts_obj_length);
8      if (Buffer::HasInstance(args[1])) {
9      SPREAD_ARG(args[1], fill_obj);
10     str_length = fill_obj_length;
11     memcpy(ts_obj_data + start, fill_obj_data,
12         MIN(str_length, fill_length));
13     // ...
14   }
15 }
```

Listing 7: The C++ implementation of the Buffer library of Node version 6.5.

attacker implements `Symbol.toPrimitive` to exploit a vulnerability present in the program listen in Listing 6.

The program in Listing 6 shows the JavaScript code of the Buffer library that is shipped with NodeJS 6.5.0 (released in August of 2016). The Buffer library in JavaScript enables handling allocated buffers in memory and Listing 6 lists the implementation of the method `fill`. This method enables to copy the contents of the first argument to the buffer, starting at the index `start` until the index of `end`. The code performs bound checks on the indices (lines 5 and 7) and then calls the bound `fill` method on the host to proceed with the memory copy operation (line 10).

The C++ implementation to perform the memory copy in the host is presented in Listing 7. This `Fill` function retrieves the indices and converts them to unsigned integers which will call *ToPrimitive* (lines 3–4). The implementation then computes the length of when will be copied (line 5), checks the write is within the allocated size (line 7) and then proceeds with the memory copy (lines 8–13).

An example malicious input program is presented in Listing 8. Here, after buffer allocation (line 1) a call to `fill` takes place with a `"payload"`, the `start`-index and the end index `1`. The call to `fill` (Listing 8, line 12) will first perform the JavaScript bound checks in Listing 7. The bound checks will call *ToPrimitive* to convert the `start`-value to a number, which ends up calling the user-defined behaviour of *ToPrimitive* implemented in Listing 8. The first bound check will update a local counter and yield a benign value of `0` (Listing 8 lines 6–7). The second bound check and successive calls to *ToPrimitive* for the `start` value will yield -1. Within the host, the

```
/* File: attack.js */
const buf = Buffer.alloc(1);
let ctr = 0;
const start = {
  [Symbol.toPrimitive]: (_hint) => {
    if (ctr == 0) // evade the check in lib/buffer.js
      return ctr++;
    else // perform attack
      return -1;
  }
};
buf.fill("payload", start, 1);
```

Listing 8: The malicious input program that bypasses both JavaScript bound-checks in Listing 6 and the host bound-checks in Listing 7.

negative value for an unsigned integer results in a very large value for `start`. This addition with the `fill_length` will become less than the `ts_obj_length` which makes the check pass, resulting in the malicious program deciding with the negative value where in memory will be written [58].

To prevent such ill-formed behaviour, we developed a dynamic analysis in which we monitor the program behaviour with respect to calls to *ToPrimitive* and attempt to prevent such behaviour. The analysis is shown in Listing 9. The analysis validates two conditions whenever *ToPrimitive* is called.

First, it validates that the layout of the object at hand does not change during the call to *ToPrimitive*. We do this by comparing the *hash* of the object before and after the computation of the intercepted *ToPrimitive* (lines 13–16).

Second, we validate that the combination of the same hint and the same object *hash* yields the same result. For the first occurrence per combination, we just store and then return the result, as we have no prior results to compare this occurrence to (lines 19–24). For hint-*hash* combinations that have occurred before, we compare if the current outcome matches prior ones (lines 26–28). If it does not, we yield the expected prior outcome and output a warning (lines 29–30).

This analysis successfully prevents the attack of the input program in Listing 8 as the second validation returns the result of the first computation, which initially was `1`.

3.3 Research Design Considerations

We now discuss design choices and considerations of our approach.

Overall, we designed BoaSpect such that it allows the analysis developer to specify an analysis compatible with the Aran API. The choice for compatibility with the Aran API is twofold. First, this API has proven to serve as a sufficient basis in terms of expressiveness for implementing a variety of analyses, thus allowing developed analyses on Aran to run on our approach with little to no required modification. Second, this allows us to port existing analyses from Aran which we then use for further testing and benchmarking to validate and evaluate our approach.

Our choice for selecting Boa to build our work on is threefold. First, the implementation language of Boa is Rust. Rust servers as a systems programming language executing at high speed with a minimal runtime, yet offering a high-level language design that aims to be memory safe and guarantees thread safety. Second, Boa is actively maintained by a community of developers that from the start aim to make Boa WebAssembly compatible, serving as a decent entry-level interpreter to extend with the support of instrumentation. Efforts are put in place to make the

```
(BoaHooks) => {
  //     value    ->  hint    ->   hash -> result
  // Map<object, Map<string, <Map<string, primitive>>>>
  let primitive_states = new Map();
  let hash = JSON.stringify /* better would be deep-conversion */;
  return {
    toPrimitive: ($value, $hint) => {
      console.log($hint);
      if (typeof $value !== "object")
        return BoaHooks.toPrimitive($value, $hint);

      // 1. verify against side-effects on object properties before and after the call
      const object_hash = hash($value);
      const res = BoaHooks.toPrimitive($value, $hint);
      if (object_hash !== hash($value))
        console.warn(`ANOMALY DETECTED - case 1`);

      // 2. verify behaviour of call to to_primitive remains constant given object properties
      if (!primitive_states.has($value) ||
          !primitive_states.get($value).has($hint) ||
          !primitive_states.get($value).get($hint).has(object_hash)) {
        // ... code to install first time result for object layout left out ...
        return res;
      }

      const previousResult = primitive_states.get($value).get($hint).get(object_hash);
      if (previousResult === res)
        return res;
      console.warn(`ANOMALY DETECTED - case 2`);
      return previousResult;
    },
  }
}
```

Listing 9: The analysis implementation using the `toPrimitive` trap to uncover unconventional or malicious use of overriding the behaviour of the abstract operation *ToPrimitive*.

engine fully ECMAScript compatible. To serve as an indication of the ECMAScript conformance, the latest Boa version as of August 2022, version 0.15, has conformance of 62.26% with the Test262 test suite. In comparison, both V8 and Spidermonkey achieve 100% conformance[2]. The Test262 test suite serves as a conformance test suite with the ECMAScript standards. Third, we chose Boa as the engine aims to tackle the 'real-world' approach of including optimization components such as a JIT-compiler to appeal as a choice of an interpreter in terms of execution performance. This makes the engine a suitable approach to gaining insights in terms of how optimizations could further coexist with instrumentation.

For our approach, we stand by the design choice of keeping the language of the input program and the analysis the same, facilitating the need for investigating the program behaviour of a JavaScript program through instrumenting it with platform support that understands a JavaScript analysis. Not only does this enable the analysis developer to more easily understand the analysis if they were familiar with the input programs under analysis, but it also enables the programmer to reason within the same programming paradigm the base-level language provides. This prevents overcomplication as the meta-properties of the interpreter are not required to be known by the analysis developer before a single safe analysis can be developed. This approach forces the burden on the instrumentation platform developer of providing clear interfaces to the analysis developer for meta-operations at the interpreter level within the same language abstraction as the input program. This is more desirable than forcing the burden onto the analysis developer of becoming familiar with the internals of the interpreter design that is typically hidden from JavaScript, or the language enforcements at the abstraction level of the interpreter. This approach is inspired by the work on reflective language design in Smalltalk and Lisp.

As explained before in Section 2.2.2, NodeProf extends the GraalJS JavaScript execution engine that is built using the truffle language framework to run on top of the GraalVM. To this end, we investigated if it was possible to make NodeProf portable to determine if GraalVM can run within a JavaScript environment. From our findings we could not find an approach in time to make this possible, thus we decided to look for an alternative approach. While technically it should be possible to compile the GraalVM project using the GraalVM native image component as a standalone binary, its compilation pipeline does not yet target WebAssembly.

3.3.1 Safety

From the perspective of the instrumentation platform designer, its implementation could affect the security of the platform in which the interpreter is hosted if it were improperly implemented. With our approach the improvements regarding safety are twofold.

First, the WebAssembly runtime environment serves as an isolated sandboxing environment. An approach to implement an *interpreter*-based instrumentation platform could have been to further modify an existing JavaScript interpreter such as V8 or SpiderMonkey aimed at running it on top of the CPU directly. The V8 engine contains over 1.35 million lines of code as of version 10.6.95, which should give a rough estimation of the size and related complexity of these execution engines that serve the dominating web browsers. The required modifications, more so for a developer unfamiliar with one of the projects, could introduce severe bugs or insecurities that could serve as entry points for exploitation if these engines were run in a sensitive environment. For extensions, to V8 or SpiderMonkey an additional attack surface would be the potential vulnerabilities the instrumentation platform might introduce, for Boa running in WebAssembly however, an additional layer of security is achieved by the safety guarantees of the WebAssembly runtime.

[2]http://kangax.github.io/compat-table/esintl/

Second, the host language serves as an additional guarantee for a specific set of vulnerabilities, and memory leaks. Current engines such as V8 and SpiderMonkey are implemented in `C++`, a language which requires manual memory management. While this may increase the performance, it comes at the cost of often being the source of bugs and vulnerabilities. Boa its implementation language is Rust, which enforces memory safety through a verification step by the borrow checker during compilation, further minimizing the attack vector.

3.4 Research Questions

To the best of our knowledge, we are the first to provide an *interpreter*-level instrumentation platform for JavaScript. In implementing this architecture, there are a number of challenges that need to be solved which we formulate in a set of research questions.

RQ1. *How can one extend a JavaScript interpreter to provide an instrumentation interface on which others can build an analysis?*
This research question aims to explore whether it is feasible to take an existing JavaScript interpreter and identify the appropriate places where to control the execution of the application and call the trap functions defined for the Aran instrumentation interface.

RQ2. *What is the execution overhead for interpreter-based instrumentation using the Boa interpreter for JavaScript?*
Since the instrumentation adds overhead to the runtime execution, we aim to quantify how much is this for analyses ranging from an empty analysis that does not instrument the application to analyses that trace every supported trap function. The quantification of the overhead of the instrumentation platform is investigated by this research question.

RQ3. *Does the overhead of portable interpreter-based instrumentation outweigh the overhead of source-code instrumentation?*
Not only it is important to determine the overhead of the platform, but also how it compares with respect to related work. In this case, we aim to compare to a source code instrumentation because it offers a portable instrumentation approach that BoaSpect aims to offer too. We formulate this research question to address this matter.

RQ4. *What benefits arise from interpreter-based instrumentation in terms of expressiveness compared to a source code-based solution?*
In this work, we start by providing an API to analysis developers compatible with the API of the Aran platform. This research question aims to investigate if an interpreter-based instrumentation backend can not only support the same analysis as an existing source-code-based one but enable new ones. Thus we aim to study the impact and consequences in terms of expressiveness most notably compared to *Aran*.

Conclusion

This chapter introduced our approach to implement an instrumentation platform that combines *portability* with *performance*, *transparency* and *expressiveness*. To this end, we implement our instrumentation platform at the level of the interpreter while providing the option to compile our implementation to WebAssembly. We further motivated how our extension to the instrumentation interface with the `toPrimitive` allows for implementing analyses that uncover potentially ill-formed applications.

Chapter 4

BoaSpect's implementation

This chapter covers the implementation of BoaSpect to include instrumentation support at the level of the interpreter as described in Section 3.1. We start by giving an architectural overview of Boa in Section 4.1 to give some insights into its components. Afterwards, we cover the changes applied to Boa's execution engine to implement BoaSpect in Section 4.2.

4.1 Architectural Overview of Boa

Boa is an open-source implementation of a JavaScript execution engine [1]. The project is developed as a Rust library for embedding the JavaScript engine in Rust applications. Additionally, the authors of Boa provide a command-line interface (CLI) for users to interact with Boa as a standalone JavaScript interpreter accessible from a command line.

Figure 4.1 presents Boa's high-level pipeline for evaluating a JavaScript program, together with an example input program and how it changes throughout the pipeline. The pipeline follows the common interpreter design which approximately consists of a *lexer*, *parser*, *bytecode compiler* and *bytecode interpreter* [59]. The *lexer* tokenizes the JavaScript source code into a sequence of tokens. For the example program "`foo("bar"); 1+2;`" listed in Figure 4.1 it is shown that this is tokenized into a sequence of 9 tokens. The *parser* parses the produced tokens into an abstract syntax tree. The example shows the output abstract syntax tree of 9 nodes, of which the root "*program*" represents the entry point of the program. The *bytecode compiler* compiles the abstract syntax tree into Boa-specific bytecode. The resulting bytecode is finally passed onto the *bytecode interpreter* which interprets it until the interpreter either terminates execution and returns the result of the computation or prematurely throws a JavaScript exception.

4.1.1 Running JavaScript Programs Using Boa

To evaluate a JavaScript program when including Boa as a library, the developer must firstly instantiate a Rust `Context`, which is Boa's implementation of the ECMAScript Execution Context. By relying on the `default` function implemented for the `Context` type as it implements the `Default` trait[1], one can instantiate a new context which will initialise among other aspects the global environment. Then, the developer can call the `eval` function on this `Context` to evaluate a JavaScript program.

A simplified implementation of the `eval` function is presented in Listing 10, which is the implementation of the pipeline shown in Figure 4.1. The code distinctly lists the different pipeline

[1]https://doc.rust-lang.org/std/default/trait.Default.html

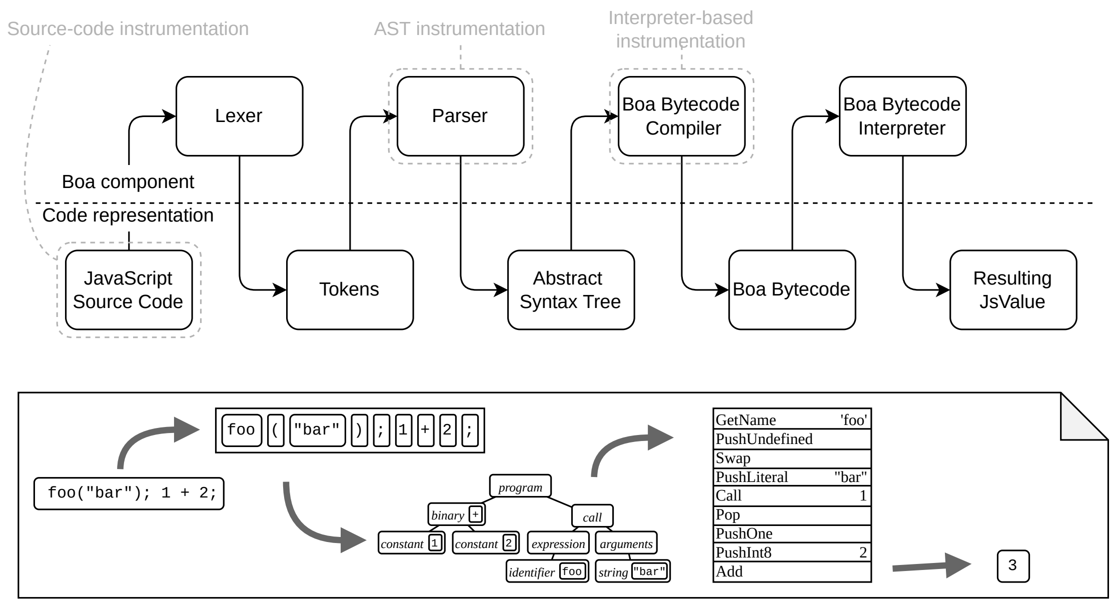


Figure 4.1: The Boa interpreter pipeline [1]. The pipeline showcases the process from input JavaScript source code to an output JavaScript value which is the result of its evaluation. The grey highlighted areas further showcase the possible points of introducing the additional operations to make instrumentation possible.

```rust
impl Context {
    pub fn eval<S>(&mut self, src: S) -> JsResult<JsValue>
    where
        S: AsRef<[u8]>,
    {
        let parsing_result = Parser::new(src.as_ref())
            .parse_all(self)
            .map_err(|e| e.to_string());

        let statement_list = match parsing_result {
            Ok(statement_list) => statement_list,
            Err(e) => return self.throw_syntax_error(e),
        };

        let code_block = self.compile(&statement_list)?;
        let result = self.execute(code_block);

        result
    }
}
```

Listing 10: The function `eval` which corresponds with the evaluation pipeline laid out in Figure 4.1 which interprets the JavaScript source code as a string reference and yields the evaluation result as a Rust `JsValue`.

stages earlier discussed as one procedure. The JavaScript source code for execution is passed to `eval` as a string reference on lines 2–3 which is parsed into an abstract syntax tree on lines 6–8. For an unsuccessful tree-parse, the engine will throw a syntax error on lines 10–13. For a successful tree-parse, the output tree is compiled into the bytecode representation on line 15. This bytecode is then interpreted on line 16, after which the result is returned to the caller of `eval`.

4.1.2 Boa's Bytecode Interpreter

Because our approach primarily relies on changes to the bytecode interpreter, it is important to cover how the bytecode interpreter works. The interpreter executes on the bytecode as a stack-based virtual machine [60]. This entails that the interpretation pushes intermediate results of the computation on a stack while executing the bytecode. The values handled by this stack are an instance of the Rust `JsValue` type. This `JsValue` type corresponds with a JavaScript runtime value which is an instance of `Null`, `Undefined`, `Boolean`, `String`, `Rational`, `Integer`, `BigInt`, `Object`, or `Symbol`.

The bytecode interpreter receives its bytecode from the bytecode compiler as a `CodeBlock`. A `CodeBlock` is essentially a continuous buffer filled with opcodes and operands. As Figure 4.1 illustrates for the example program "`foo("bar"); 1 + 2`", its output is compiled into a `CodeBlock` consisting of 9 opcodes, four of which have an operand. The order of execution of the bytecode for the example program is the following:

Stack state	Opcode	Instruction behavior
[]	`Getname 'foo'`	Lookup the value bound to 'foo' in the current environment, push result on stack
[`foo`]	`PushUndefined`	Push `undefined` as a placeholder for if the callee yields no return value
[`foo`, $undefined$]	`Swap`	Swap top of stack and second element on stack such that caller is on top of stack
[$undefined$, `foo`]	`PushLiteral "bar"`	Push `"bar"` on stack as an argument in preparation for function call
[$undefined$, `foo`, `"bar"`]	`Call 1`	Perform function call for "1" argument, this will pop the arguments and callee, perform function call and push result on stack
[$undefined$, `"bazz"`]	`Pop`	Discard result of function call as it is unused
[$undefined$]	`PushOne`	Push the JavaScript value `1` on the stack
[$undefined$, 1]	`PushInt8 2`	Push the JavaScript value `2` on the stack
[$undefined$, 1, 2]	`Add`	Add two top-most arguments of the stack, push result
[$undefined$, 3]		end of `CodeBlock`, return to caller

Listing 11 shows the implementatio of `execute_instruction`, which executes individual opcodes. This function reads the current opcode by dereferencing its current program counter on the current call frame on lines 4–6. After the current opcode has been read, `execute_instruction` increments the program counter by one in preparation for executing the next opcode. The next opcode is executed after handling the current instruction, which is performed by repeatedly calling `execute_instruction` until the program counter reaches the end of the code block. The function `execute_instruction` further matches the read opcode against the known enumeration of opcodes on lines 11–20. As shown in the example program in Figure 4.1 the JavaScript "+" compiles to the `Opcode::Add` opcode. The implementation to execute `Opcode::Add` is visible on lines 14–17. This implementation pops both the left and right operand from the stack on lines 14–15. Next, the Rust `add` function is called for the left operand, passing both a reference to the right operand and the current context on line 16. The result of the call to `add` is pushed on the stack

```rust
impl Context {
    fn execute_instruction(&mut self) -> JsResult<ShouldExit> {
        let opcode: Opcode = {
            let opcode = self.vm.frame().code.code[self.vm.frame().pc]
                .try_into()
                .expect("could not convert code at PC to opcode");
            self.vm.frame_mut().pc += 1;
            opcode
        };

        match opcode {
            // ... other opcodes
            Opcode::Add => {
                let rhs = self.vm.pop();
                let lhs = self.vm.pop();
                let value = lhs.add(&rhs, self)?;
                self.vm.push(value)
            };
            // ... other opcodes
        }

        Ok(ShouldExit::False)
    }
}
```

Listing 11: A portion of the Rust source code is responsible for interpreting the opcodes that are a result of the Boa bytecode compiler. The code illustrates how the opcode `Opcode::Add` further instructs the VM to pop the operands and call the function `add` on the operands, which will further call the abstract operation `ApplyStringOrNumericBinaryOperator` as defined by the ECMAScript standard.

on line 17.

Note that the rust call to the `JsValue::add` function further implements the Rust implementation of appropriately adding both values. This `JsValue::add` function implements the abstract operation `ApplyStringOrNumericBinaryOperator` as it is defined in the semantics of the ECMAScript standard [27]. This abstract operation further implements the polymorphic behaviour of JavaScript's "+" operator. This implementation is a recurring pattern, in which the bytecode interpretation eventually performs computations on JavaScript values as they are defined by the ECMAScript.

4.2 Modifying Boa to Include BoaSpect

In the previous section, we described step-by-step how Boa's bytecode interpreter works. This section describes the modifications and extensions we made to Boa, such as modifying the bytecode interpreter in order to support the implementation of dynamic analyses. In addition to the aforementioned features an instrumentation platform should have, *portability* with *performance*, *transparency* and *expressiveness*, there are some requirements that need to be addressed to make BoaSpect usable.

We will thus explain the faced challenges to enabling interpreter-based instrumentation and our solutions to those challenges to implement BoaSpect with the desired platform properties.

The implementation of Boa's *engine* component spans 325 rust files accumulating to a total of 105 thousand lines of code, excluding the CLI- or WebAssembly-interface, but including the scanner, lexer and the *bytecode interpreter*. To give the reader a sense of the required changes to enable the instrumentation platform within Boa, the implementation of BoaSpect changed 14 of the 325 files, accumulating to a change of over 1.2 thousand lines of code (adding new or modifying existing lines). These changes do not include code to test or benchmark BoaSpect.

The changes performed in Boa restrict themselves to the *bytecode interpreter*, meaning we did not change or extend in any form the code in the *lexer*, *parser* or the *bytecode compiler*. This implies that BoaSpect does not rewrite JavaScript source code, the parsed abstract syntax tree or the generated bytecode, rather it changes the manner in which the bytecode is interpreted by the *bytecode compiler*.

The rest of this section discusses the requirements of the instrumentation platform following the motivation discussed in Section 3.1. Each of the following requirements is further discussed in the subsections below from the perspective of the modifications and extensions applied to the engine.

- The platform has to provide means to accept an analysis advice separate from the input program.

- Once the instrumentation platform received the advice, it should be aware of the implemented trap functions which it should correctly hook into the respective operation. We further refer to these "points of interest" as *joinpoints*, a term we borrow from the domain of aspect-oriented programming. The current implementation of BoaSpect considers the presence of a trap function as the advice targetting all operations corresponding to this trap. We leave the means to define fine grain *pointcuts* future work, which is not a limitation to our approach but an extension we currently did not implement.

- The instrumentation platform should correctly separate the two contexts, input program and analysis, such that no analysis code would be instrumented. We further refer to these contexts as the *base context* for the input program and the *meta context* for the analysis.

- The analysis advice itself may further rely on reflective tools provided by the platform enabling the analysis to correctly fall back to the level of the input program.

The rest of this section covers the requirements and implementation aspects of BoaSpect. We cover the advice initialization in Section 4.2.1, ensuring a correct context separation between the input program and analysis in Section 4.2.2, ensuring the hooks to the *trap functions* are inserted during evaluation in Section 4.2.3 and providing the analysis with the required reflective operations in Section 4.2.4.

4.2.1 Accepting and Installing an Advice

As mentioned in Section 3.2, the first step in instrumenting the input program is to determine the *joinpoints* for the analysis. It is the first requirement for BoaSpect, similar to other JavaScript instrumentation platforms such as Aran and NodeProf, to provide an interface to accept an advice separate from the input program.

As mentioned in Section 4.1, the `Context` is the entry point to instantiate the execution engine and execute JavaScript programs. We extend the `Context` type with the function `install_advice`, such that it can be called to install the advice in BoaSpect before the evaluation of the input program takes place, as is motivated in Section 3.2. In doing so we allow the analysis to *hook* into the fresh environment to change the environment at their will. This enables the analysis, among other things, to set up a certain set of environment properties or modify pre-existing prototypes provided by the default runtime. To elaborate, we cover the changes made to the Boa CLI to account for an advice specification.

The CLI is extended to accept an optional `--advice` flag followed by an argument to indicate the program should run under the presence of an analysis. The value of the argument specifies the file path to the advice itself. The extension of the CLI is listed in Listing 12, it first instantiates a new `Context` (line 5) after which the optional advice is installed (lines 8–11). Eventually, the rest of the input program evaluation takes place (line 16). The evaluation of the input program for this listing is left out for the sake of brevity, however, its statements are to read the input program file contents and call the `eval` function presented in Listing 10 for this program. This example showcases the required changes needed in order to extend Boa with BoaSpect. The changes required are a single call to `context.install_advice` with a reference to the advice.

The implementation of `context.install_advice`, however, require some more detail. To install the advice, the interpreter firstly evaluates the analysis specification (line 26), ensuring it is callable (lines 27–29) after which it retrieves the traps (line 32) which are then installed together with the advice itself in the `Context` structure (lines 33–35). By encapsulating analysis state in the scope of the immediately invoked function as a closure, the analysis developer can further ensure the analysis *transparency*.

As demonstrated in Listing 12, it is the execution `Context` value that holds a reference to the advice throughout the evaluation. We chose to expand this value with a reference to the advice as this value remains accessible throughout the entire evaluation code base, thus making it accessible whenever hooking into the traps is required.

4.2.2 Distinguishing the Execution Context

One key distinction to be made throughout the evaluation is the execution of the input program from the execution of the analysis program. Whenever the interpreter evaluates an operation where instrumentation is applicable, it should *jump* into the defined trap function, evaluate its body with the instrumented information and afterwards return to the continuation of the operation. If the interpreter fails to distinguish the evaluation of *analysis code* from the *input*

```rust
//  # File: CLI.rs
pub fn main() -> Result<(), std::io::Error> {
    let args = Opt::parse();

    let mut context = Context::default();

    if let Some(path) = &args.advice {
        let advice_buffer = read(path)?;
        context.install_advice(advice_buffer);
    };

    /* ... rest of interpreter ... */

    Ok(())
}

//  # File: Context.rs
impl Context {
    /* ... more implementations for the Context type ... */

    pub fn install_advice<S>(&mut self, advice_buff: S) -> Result<(), JsValue>
    where
        S: AsRef<[u8]>,
    {
        self.instrumentation_conf.set_mode_meta();
        let advice_callback = self.eval(advice_buff)?;
        let advice_callback = advice_callback
            .as_callable()
            .ok_or(self.construct_error("Advice not callable"))?;
        let meta_hooks = Hooks::default(self);
        let advice = advice_callback.call(&JsValue::Null, &[meta_hooks], self)?;
        let traps = Traps::from(&advice, self);
        self.instrumentation_conf.install_traps(traps);
        self.instrumentation_conf.install_advice(advice);
        self.instrumentation_conf.set_mode_base();
        Ok(())
    }

    /* ... more implementations for the Context type ... */
}
```

Listing 12: Parts of the Boa interpreter command line interface that were extended for instrumentation support. The highlighted lines show the inclusion of BoaSpect for installing an advice. The `main` function illustrates the additional support for reading an analysis specifications. The `install_advice` function for a `Context` illustrates the evaluation of the advice specification after which the traps are extracted and references to both are kept in the `Context`.

code it would be the case that the instrumentation platform starts instrumenting the analysis itself. Doing so, unintentionally according to the analysis, would yield incorrect and nonsensical results. Furthermore instrumenting the analysis itself within the same context would lead to unbound recursion if one of the operations that the advice targets is used in the advice itself. Andreasen et al. [3] note to be cautious for unbound recursion similarly but for source code instrumentation:

> " One of the key challenges in source code instrumentation is that the injected code could use a library that is itself instrumented by the instrumentor. This could lead to unbounded recursive function calls when the instrumented program is executed. "

Doing so for Jalangi, Sen et al. [10] state that libraries used by the analysis should be loaded in a private namespace accessible exclusively to the analysis to prevent this unbound recursion from occurring.

For BoaSpect, so far this distinction has not yet been clarified from a technical point of view. Our approach requires the analysis developer to provide a set of traps as JavaScript functions, bound to the advice as is a JavaScript object. During the execution of the input program, the instrumentation platform *calls* into the trap function whenever the respective operation would be evaluated at the level of the input program. Thus, it is the *function call* to a trap function that could – and should – inform the interpreter about a change in context from *base* to *meta* or vice versa while the interpreter "walks" through the evaluation of both the input program and the advice. For the *function call* to be able to inform the interpreter if the call changes the context, it requires the function that is being called to store some form of information about the context of the function body. This is to say, the execution of the trap functions their body should not be instrumented. BoaSpect names this function property that is stored within a function its "evaluation mode", which is either `EvaluationMode::Base` reflecting the level of the input program or `EvaluationMode::Meta` reflecting the level of the advice. Thus, during function creation, a function should alongside its lexical scope and body also store the evaluation mode. In turn, this requires the function to be informed in which context it is created, being either `EvaluationMode::Base` or `EvaluationMode::Meta`. This information is passed on from the interpreter depending on the current evaluation mode of the interpreter itself, meaning that functions that are created within the `Meta` context are informed to evaluate in `Meta` context too and functions created in the `Base` context should evaluate their body in `Base` context too. Indeed, from the point of the analysis, this corresponds with the requirement that the analysis code is not subject to being analysed itself by the same analysis.

Note the back-and-forth communication required between the interpreter and first-class function values: the interpreter informs a function at creation time in which context it is created, and the function informs the interpreter at call time in which context it should continue its evaluation of the body. Thus we came up with the proposed solution, throughout the evaluation of the program the interpreter is required to keep a certain flag that tracks its execution state which is either for the *analysis* or the *base*. This flag is stored in the creation of functions and allows for the interpreter to decide whether or not to change its flag to *analysis* or *base* upon calling such a function.

To elaborate further on this, Listing 13 shows the Boa implementation for JavaScript function creations and JavaScript function invocations. The listing highlights the required modifications to ensure the interpreter can distinguish both layers of its execution. The Rust function `create_function_object` (line 1) partially shows how the code of the function and the environment are stored in a Rust structure that represents a runtime function object. The modifications on this function show how an additional `evaluation_mode` value is stored in the Rust structure, which is retrieved from the interpreter at the time of creating the function through the context its instrumentation configuration (line 9). The Rust function `call` (line 16) is called whenever JavaScript objects are used as a call target. This function illustrates how the `evaluation_mode` is retrieved from the function (lines 27, 31) which then determines the interpreter state upon entering the body of the function (lines 43–44) which is restored after the call of the function (line 49).

The curious reader might remark that one crucial aspect of this approach of distinguishing the contexts has not been covered: if the interpreter informs the function at creation time of the context to store and the function informs the interpreter at call-time of the context to restore, where does it start? The answer to this lies in the entry point of both programs. As the interpreter starts with the evaluation of the analysis as presented in Listing 12, it sets its evaluation mode to `Meta` (line 25) before evaluating the advice such that the outer function that yields the advice object is informed about the `Meta` mode and all subsequent function values that originate from it are within this mode too. After the evaluation of the advice, the instrumentation platform restores its evaluation mode to `Base` (line 35) after which regular evaluation proceeds.

Note that this approach does not prevent the analysis author from leaking values from the analysis into the input program. It does, however, ensure that values that do leak the analysis layer will not be instrumented as they have reached the input program layer, which is beneficial for analyses that might want to wrap values used by the input program with values that include analysis code which should still not be instrumented.

4.2.3 Hooking in the Operations

In addition to being capable of distinguishing the two contexts, the interpreter is still required to decide when to hook into the operations and call one of the corresponding *trap functions* This requires BoaSpect to alter the interpretation of the program to make an additional call.

In the implementation of `execute_instruction`, partially shown in Listing 11, we implemented the required changes to call the `binary` *trap* instead of evaluating the base program binary expression. Rather than matching for the opcode (line 12) which will match with the included matching arm (line 14), we want to preliminary test if the execution engine is in the `EvaluationMode::Base` state (ie. at the evaluation level of the input program) and verify if the opcode matches one of the opcodes of interest. If the bytecode interpreter would then match its current instruction with the *Opcode::Add* opcode, it would decide to alter its default evaluation. The evaluation would now perform a set of different steps of which the order is important, a recurring pattern we found for the implementation of *hooking* into operations.

1. Set the evaluation mode to `EvaluationMode::Meta` as the operations that follow could – but should not – trigger additional instrumentation operations.

2. Gather all required information which will be provided to the *trap function*, reifying the values that are accessible to the interpreter but not available at the level of JavaScript.

3. Call into the *trap function*, awaiting the result. The subsequent interpretation might further trigger compilation and opcode interpretation, but the interpreter state at the level of `EvaluationMode::Meta` will prevent the advice code from being instrumented.

4. Upon receiving a result, terminate the advice-level interpretation by restoring the interpreter evaluation mode to `EvaluationMode::Base`.

5. Yield the result to the operation that was instrumented and prevent the default interpretation of this operation to take place, ensuring further interpretation of the input program can take place successfully.

```
pub(crate) fn create_function_object(
    code: Gc<CodeBlock>,
    context: &mut Context,
) -> JsObject {
    /* ... function object creation preparation (prototype, signature) ... */
    let function = Function::Ordinary {
        code,
        environments: context.realm.environments.clone(),
        evaluation_mode: context.instrumentation_conf.mode(),
    };
    /* ... function additional properties (eg. constructor) ... */
    function
}

impl JsObject {
    pub(crate) fn call(
        &self,
        this: &JsValue,
        args: &[JsValue],
        context: &mut Context,
    ) -> JsResult<JsValue> {
        let body = match self {
            /* other function types left out: eg. built-ins */
            Function::Ordinary {
                code,
                environments,
                evaluation_mode,
            } => FunctionBody::Ordinary {
                code: code.clone(),
                environments: environments.clone(),
                evaluation_mode: evaluation_mode.clone(),
            }
        };

        match body {
            /* other function types left out: eg. built-ins */
            FunctionBody::Ordinary {
                code,
                mut environments,
                evaluation_mode,
            } => {
                /* ... call preparation (pushing function, call frame and args on the stack) ... */
                let outer_evaluation_mode = context.instrumentation_conf.mode();
                context.instrumentation_conf.set_mode(evaluation_mode);

                let result = context.run();
                context.vm.pop_frame().expect("must have frame");

                context.instrumentation_conf.set_mode(outer_evaluation_mode);

                context.realm.environments.pop();
                Ok(result)
            }
        }
    }
}
```

Listing 13: Parts of the Boa interpreter for the creation and usage of function objects. The highlighted lines illustrate the storage and retrieval of the interpreter state which enables the interpreter to determine at call-time for a function if it should be instrumented or not.

In the case of binary instrumentation, this would require us to *(1)* set the evaluation mode to `Meta`, *(2)* pop the right operand `rhs` and the left operand `lhs` of the VM stack, *(3)* call into the binary *trap function* after which we can *(4)* set the evaluation mode back to `Base` and finaly *(5)* push the result on the stack rather than pushing the result of "`lhs.add(&rhs, self)?`" on the stack as is the case in Listing 11 (lines 17–18).

Listing 14 shows the modifications made to the bytecode interpreter for supporting the instrumentation of primitive values. This shows the preconditions that take place for instrumenting the primitive values (lines 5–17) after which the *(1)* evaluation mode is updated (line 18), *(2)* the value is retrieved that would be pushed on the stack but (lines 19–21), *(3)* the call into the primitive *trap function* takes place (line 22) after which *(4)* the evaluation mode is restored (line 26) and finaly *(5)* the result is pushed on the stack rather as the outcome of this operation (line 27).

This pattern of changing the interpretation context, calling into the trap and returning to the continuation generalizes for all the *trap functions* instrumented by BoaSpect for Boa. The differences between the traps mostly are with the manner in which the information available for the interpreter but relevant for the trap function is reified to make it available for the advice. The most notable alternative implementation is that for trapping function applications, as motivated by our approach outlined in Section 4.2.2. The implementation of hooking into the abstract operation ToPrimitive operation is similarly implemented by including the aforementioned pattern in the Rust counterpart implementation of `JsValue::to_primitive`.

4.2.4 Interpreter Operation Reification

After discussing the implementation details for the instrumentation platform to accept an advice specification, we covered how the instrumentation platform distinguishes the two contexts and how the interpreter jumps from the base context to the meta context through calls to the *trap functions*. A crucial implementation aspect, however, is to allow the analysis developer to make the context leap from the analysis back to the input program. This is to say, implicitly the analysis can return to the context of the input program with a `return` statement.

However, by the design of the API, the analysis is required to yield the result of the computation it intercepts from the input program. Through the computation, the analysis must communicate back to the interpreter that the computation thereof requires further instrumentation. By the approach of BoaSpect, function bodies defined by the input program are instrumented as they are evaluated, thus a call to `Reflect.apply` will inform the interpreter of the context switch. However, a problem arises for certain operations, such as binary or unary operations when the result must be computed at the level of the base context. The signature of binary trap functions consists of the operator, left operand and right operand: `(op, l, r)`. If the binary trap function receives the following bound arguments `("+", 1, 2)`, it might implement the computation of the result as a switch-statement based on the operator and compute the result as such "`case "+": return l + r;`". While the result of the computation will remain correct, the conclusion of the analysis might be incomplete, we explain now why.

The trap functions supported by BoaSpect include the trap function `toPrimitive`. Recall from Section 3.2.2 that this trap supports instrumentation of the abstract operation ToPrimitive used by the interpreter to convert values to primitives.

As such, the operation "`l + r`" may trigger the invocation of the `toPrimitive` function defined in Boa aligning with the ECMAScript specification is one of both operands requires conversion to a primitive JavaScript value. The issue now makes clear what happens at the level of the interpreter when the advice would rely on its self-defined computation "`case "+": return l + r;`". As the interpreter evaluates the advice, it would encounter the `Add` *opcode* in the advice code.

```rust
fn execute_instruction(&mut self) -> JsResult<ShouldExit> {
    let opcode: Opcode =
        self.vm.frame().code.code[self.vm.frame().pc];

    if let EvaluationMode::BaseEvaluation = self.instrumentation_conf.mode() {
        match opcode {
            // Primitive instrumentation
            Opcode::PushUndefined
            | Opcode::PushNull  | Opcode::PushTrue   | Opcode::PushFalse
            | Opcode::PushZero  | Opcode::PushOne    | Opcode::PushInt8
            | Opcode::PushInt16| Opcode::PushInt32 | Opcode::PushRational
            | Opcode::PushNaN   | Opcode::PushPositiveInfinity
            | Opcode::PushNegativeInfinity | Opcode::PushLiteral => {
                if let Some(traps) = &mut self.instrumentation_conf.traps {
                    let traps = traps.clone();
                    if let Some(ref trap) = traps.primitive_trap {
                        if let Some(advice) = self.instrumentation_conf.advice() {
                            self.instrumentation_conf.set_mode_meta();
                            self.vm.frame_mut().pc -= 1;
                            let _ = self.execute_instruction();
                            let value = self.vm.pop();
                            let result = self.call(trap, &advice, &[value]);

                            match result {
                                Ok(result) => {
                                    self.instrumentation_conf.set_mode_base();
                                    self.vm.push(result);
                                    return Ok(ShouldExit::False);
                                }
                                Err(v) => {
                                    panic!("Instrumentation: Uncaught {}", v.display());
                                }
                            }
                        }
                    }
                }
            }
            /* ... match against other opcodes for instrumentation ... */
        }
    }

    match opcode {
        /* ... match against opcodes for regular execution ... */
    }
}
```

Listing 14: A portion of the code present in the virtual machine component of Boa that interprets the compiled bytecode. The highlighted lines show the inclusion of BoaSpect for instrumenting primitive expressions. This code intercepts regular execution to collect the primitive value which it passes to the trap function for further analysis. After the result is returned from the trap function the result is pushed on the stack, replacing the primitive expression.

This opcode would result in the interpreter performing a Rust call into the abstract operation `ApplyStringOrNumericBinaryOperator`. By the definition of the standard, the abstract operation `ApplyStringOrNumericBinaryOperator` will in turn call the abstract operation ToPrimitive on its operands. As these Rust calls all take place within the `Meta` evaluation mode, the abstract operation ToPrimitive will *not* be instrumented. However, semantically, this operation should have been instrumented as the call to ToPrimitive is unavoidable by the input program performing the binary add-operation.

This issue illustrates the need for the advice to be capable of informing the instrumentation platform to perform the operations at the level of the input program. We provide the analysis access to an object containing *hooks* into the operations that were instrumented. By calling these hooks, the interpreter is informed to change its evaluation mode to the level of the input program, compute the result of the operation that was intercepted, change the evaluation mode back to the level of the analysis and proceed with the *continuation* of the analysis. This object with the hooks into the interpreter is provided to the top-level anonymous function that upon evaluation yields the advice object. For example, the "advice.js" definition would accept an additional `BoaHooks` argument on line 1 in Figure 3.2. This is also visible in the function `install_advice` shown in Listing 12. After evaluating the analysis function that yields the advice (lines 26–29), the object with hooks is constructed (line 30) after which it is passed on to the analysis function (line31).

4.3 Targetting WebAssembly

To compile BoaSpect to WebAssembly, we rely on the *Wasm-pack* toolchain, which is developed to compile Rust codebases to WebAssembly [61]. This toolchain enables us to write a new Rust library in which we include BoaSpect as our sole dependency, write a single Rust `evaluate_with_advice` function that accepts both an input program and input analysis as a string and returns the evaluation result.

We then instruct *Wasm-pack* to export the `evaluate_with_advice` function from the compiled WebAssembly module. Doing so, the output WebAssembly module for BoaSpect resulted in a 3-megabyte file accompanied by the JavaScript bindings that support initializing the WebAssembly module memory and passing the input programs as JavaScript strings. This WebAssembly module can now be used within a WebAssembly-supported environment to execute and instrument JavaScript programs.

Conclusion

This chapter discussed the implementation of Boa and the changes and extensions applied to implement BoaSpect. The changes that enable BoaSpect to serve as an instrumentation platform were covered next. This ranged from extending the interpreter to accept an analysis specification, to appropriately hooking into the trap functions. Furthermore, we covered how we ensure that the analysis does not instrument itself, and we explained the requirement that enables the analysis to compute the expected result from the operation that was intercepted.

Chapter 5

Evaluation

In this chapter, our goal is to evaluate BoaSpect. We elaborate on our experimental setup in Section 5.1. In Section 5.2 we evaluate how our modifications to Boa comply with the compatibility with ECMAScript. Recall from Chapter 2 that one can evaluate an instrumentation platform with respect to four different properties, which we do so in subsequent sections: *transparency* in Section 5.3, *performance* in Section 5.4, *portability* in Section 5.5, and *expressiveness* in Section 5.6. In Section 5.7 we answer the research questions constructed in Section 3.4 with the conclusions drawn during the evaluation of BoaSpect.

5.1 Experimental Setup

We performed our evaluation on a server hosting 256GB of ram and an Intel® Xeon® E5-2637 v3 CPU @ 3.50GHz. The software available on the server during our evaluation was Ubuntu 20.04.4 as the operating system with Node v15.14.0. The utilised version of Boa, both with and without modifications to implement BoaSpect, was at version 0.15.

We cover the different sets of input programs and input analyses used to perform experiments for the rest of the evaluation next.

5.1.1 Input Programs

We first cover the different sets of input programs.

Aran/test
We use the test suite developed by the authors of Aran[1] to test for the correctness of JavaScript operations as they are being instrumented. The test suite consists of a total of 62 input programs. Each input program is built as a unit test, asserting the correctness of a specific JavaScript operation in isolation. For example, an input program may test if the outcome of a binary operation matches the expected result. As a whole, this set of programs is meant to ensure that Aran does not violate the input program semantics, rather than asserting that the analysis works as intended.

V8/webkit
We use the test suite V8/webkit[2] to assess the compatibility of Boa and BoaSpect with

[1]https://github.com/lachrist/aran/
[2]https://github.com/v8/v8/

respect to the ECMAScript semantics as means of regression tests. The test suite consists of a total of 384 input programs in which each program tests a JavaScript feature as a unit, accumulating to a total of 3.410 assertion statements. These test files aim to assert that the semantics of the interpreter is correct from the point of view of the input program instead of the interpreter state. We firstly investigated the option to make use of the test suite test262, a test suite consisting of over 58 thousand tests covering the whole ECMAScript specification[3]. However, we noticed this test suite not only ensures the evaluation of the input program is correct but further asserts properties on the interpreter state, such as how many objects are allocated at a program point. Because of the changes BoaSpect adds to Boa, the interpreter state is different upon including support for instrumentation as we alter properties of the interpreter, such that tests would fail. The *V8/webkit* suite follows a more permissive approach in asserting only the outcome and behaviour of the input program, which is in our case more adequate for evaluating BoaSpect.

Sunspider suite (version 1.0.2)
We use the Sunspider suite[4] as a benchmark suite to assess the performance of our solution. The suite is designed to assess the performance and optimization capabilities of a JavaScript interpreter. This benchmark suite originated in 2007 and has been kept up to date over different iterations until 2011 to version 1.0.2. It consists of a total of 26 input programs, benchmarking a variety of JavaScript features such as manipulating `Date`s, `String`s to performing compute-intensive arithmetic operations for the evaluation of functional or object-oriented programs.

We investigated the use of other existing JavaScript benchmark suites such as Kraken[5] (developed by Mozilla), Octane[6] (developed by Google) and JetStream[7] (developed by Webkit). However, those benchmark suites are either specialized for a certain interpreter or proved lower compatibility with Boa compared to the Sunspider suite. Furthermore, the setup requirements of the alternative benchmark suites were unsupported out-of-the-box, while the Sunspider suite mostly works from the start.

5.1.2 Input Analyses

We use a set of input analyses for the evaluation of BoaSpect. The analyses are the subset of analyses used to evaluate Aran [9] which are compatible with the limited number of trap fucntions BoaSpect currently supports. The input analyses are the following.

"empty.js"
This analysis implementation yields an advice in which no trap function is specified. Its definition is "`(_BoaHooks) => ({})`". In terms of runtime behaviour, it serves as an indication of the performance overhead added by the instrumentation platform to the interpreter when the instrumentation platform does not trap any operation.

"forward.js"
This analysis implementation yields an advice in which every trap function is defined to return the computation of the expected result of the instrumented operation. In terms of runtime behaviour, it serves as an indication of the overhead of an analysis which traps all operations of a target program but it does not add additional analysis code.

[3]https://github.com/tc39/test262
[4]https://github.com/WebKit/WebKit
[5]https://wiki.mozilla.org/Kraken
[6]https://developers.google.com/octane/
[7]https://www.browserbench.org/JetStream/

	Uninstrumented input program			
Input program suite	Total	Success	Crash	Timeout
Sunspider suite (v1.0.2)	26	20	2	4
Aran/test	62	52	10	0
V8/webkit	384	335	42	7
Combined	472	407	54	11

Table 5.1: The considered input program suites to validate Boa's compatibility with the ECMAScript language. *Total* indicates the number of input programs per suite, *Success* the number of programs that Boa can execute successfully, *Crash* the number of programs which Boa does not support and *Timeout* for the number of programs that did not terminate within our devoted time budget per program.

For example, the definition of the `apply` trap function for this analysis implementation is "`apply: (f, t, args) => Reflect.apply(f, t, args)`". This will result in maximum instrumentation effort as the platform will instrument every available operation that is targeted by the advice which for this example is every available trap function aside from `toPrimitive`. The instrumentation of `toPrimitive`-operations is left out as it would result in an unfair comparison with the Aran platform as it is not available for Aran. We perform a separate evaluation for this trap with the analysis implementation in "toPrimUncover.js", explained below.

"logging.js"
This analysis is similar to the "forward.js" analysis in that it implements every trap function, but in addition outputs to a buffer what operation took place.

For example, the definition of the `apply` trap function in this analysis then is "`apply: (f, t, args) => { buffer.write("apply"); return Reflect.apply(f, t, args) }`".

"profiling.js"
This analysis intercepts all the function applications and models the depth of the call stack.

"toPrimUncover.js"
This analysis implementation yields an advice in which exclusively the trap function `toPrimitive` is specified. The analysis traces all *ToPrimitive* operations, such that objects that are converted to a primitive value do not change their internal structure before and after the operation. The implementation of this analysis and its motivation are discussed in Section 3.2.2.

5.2 Compatibility

An essential part of our instrumentation platform is compatibility with the ECMAScript language. If our instrumentation platform does not properly support a certain language feature, it would not be possible to perform dynamic analysis on programs that use said feature. As such, it is our goal that the extensions to Boa should comply with the ECMAScript specification as long as the analysis is transparent.

From the perspective of the implementation, we aim to comply with the ECMAScript specification to remain transparent as long as the analysis remains transparent. How we achieve this in the implementation is threefold. First, we achieve this through the careful insertion of the intercepting operations. We ensure that whenever a trap function is called, the default behaviour is prevented and the result from the trap function is returned while respecting the expectations of the continuation for the interpreter. These expectations include for example the stack layout according to the calling convention or the register state for the virtual machine interpreting the opcodes. Second, we achieve correctness by ensuring that the provided hooks from the platform to the analysis to compute the intercepted operation outcome are correct by providing the identical ECMAScript implementation of what is intercepted. Third, we report that the set of 991 unit tests that come with the Boa codebase all pass after the extension of BoaSpect.

Next, we cover compatibility from the perspective of the execution of input programs. The input programs used are the combination of the programs from *Aran/test*, *V8/webkit*, and *Sunspider suite (version 1.0.2)*. To evaluate per input program if BoaSpect remains compatible, we first validate whether Boa in the first place is compatible. To this end, we verify per input program if the program terminates under 20 minutes on Boa, the execution engine without any instrumentation. If the test input program crashes (ie. one of the program assertions fails or execution fails), we label it *Crash*. If its time budget to execute exceeds 20 minutes we label it *Timeout*, otherwise we label the program *Success*. For Boa, from the 472 input programs combined from the suites 407 are labelled *Success*. A total of 54 programs reported a *Crash* and 11 programs reported *Timeout*. Table 5.1 elaborates on the test results for Boa per input suite.

Given these input programs, we further validate that the instrumentation platform BoaSpect remains compatible with ECMAScript. To this end, we conduct an experiment in which we compare the outcome of the uninstrumented input programs with the output of their instrumented versions after applying the set of transparent analyses {"empty.js", "forward.js", "toPrimUncover.js"}. Our criteria for an "equal outcome" is determined at the level of the interpreter. This entails that we assert the final value for all evaluations of an input program (instrumented and uninstrumented) is the same according to the ECMAScript abstract operation *IsStrictlyEqual*. This abstract operation corresponds with the equality operator "`===`". Furthermore, if it were the case that intermediate values were not equal, its incorrectness would propagate to the program end the considered input programs are augmented with assertions that would early throw a program error for unexpected intermediate program values.

We report that for a combined total of 407 input programs for which the uninstrumented evaluation was executed successfully (ie. Boa reporting the label *Success*), all instrumented variants report an equal program outcome except for one. The sole failed input program is "test-Literal.js" from the *Aran/test* suite, which fails due to the final statement outcome being the regular expression literal "`/abc/g`". Two evaluations of the literal "`/abc/g`" can not be equal according to the definition of the *IsStrictlyEqual* operation. This is thus a false positive and we can conclude that for the set of considered input programs our platform preserves the input program semantics under transparent analysis.

Note that the programs that reported *Crash* or *Timeout* for Boa were not further evaluated against BoaSpect's ECMAScript compatibility since the applied instrumentation would not help to fix these tests or improve the execution speed.

5.3 Evaluating Transparency

Recall from Section 2.3 that transparency is how easily the platform presence can be uncovered by the input program. From our assertions that BoaSpect does respects the ECMAScript semantics in Section 5.2 no test suite reports unexpected behaviour which would point out BoaSpect leaks presence information.

We will discuss two examples in which source code instrumentation leaks information but BoaSpect does not because it uses an interpreter-based instrumentation approach. Source-

```
1 function f (x) {
2   arguments[0] = "bar";
3   assert(x === "bar");
4   x = "qux";
5   assert(arguments[0] === "qux");
6 }
7 f("foo");
```

Listing 15: An example input program for which the assertions fail for the Aran instrumentation platform but do not fail for the BoaSpect instrumentation platform.

code instrumentation techniques suffer from lack of transparency because of reflective JavaScript operations such as `Function.toString` and `Function.prototype.name` may uncover the source-code transformed input program at runtime.

First, we investigate the behaviour of BoaSpect compared to Aran for the program presented in Listing 5. This program asserts the expected output for a computed function name and the function body. However, the implementation of Boa fails for both assertions as the computed name of a function evaluates to `undefined` and the evaluation of `Function.toString` yields a simplified string representation of the program rather than the full function body as a string. From the design of BoaSpect, however, we know that the assertion for `Function.toString` would not be violated as BoaSpect's instrumentation technique does not perform any function body rewrites. Similarly, we know that the assertion for `Function.prototype.name` does not require any static analysis to determine the assigned function names which would fail for computed name properties, which is the approach taken by Aran.

We now discuss a second input program example from Aran which we can run on BoaSpect [12]. The input program is shown in Listing 15 and does asserts the reflective *arguments* object in function bodies behaves as expected. This program defines the function `f` in which the body modifies the *arguments* object, which is a reification of the sequence of actual arguments passed to the function at call time. Ensuring transparency of the analysis means that modifying the *arguments* object (line 2) should ensure argument access through reading the argument value from the environment should yield the modified value (line 3) or, vice-versa, assigning a new value to an argument through an assignment operator such as "=" (line 4) should be reflected in the *arguments* object (line 5).

When writing the analysis in a source-code instrumentation platform like Aran, this bidirectional link between the call-time arguments and the *arguments* object is broken as the instrumentation platform transforms functions into parameterless versions for further use by the platform. In our work, this link is preserved as we keep the representation of functions the way they are initially defined. Furthermore BoaSpect implements the *trap functions* `read` and `write`, which instrument reading or writing environment variables, to read from the available environment from the interpreter. We ran the "forwards.js" analysis on the test input program in Listing 15 both on Aran and on BoaSpect, and report the instrumentation for Aran fails while the instrumentation for BoaSpect passes.

We believe we are not leaking information that could break transparency for BoaSpect. The only feature which can compromise transparency would be through the `Function.prototype.caller` property, which yields the function caller at the point in time for a given function call. This would leak stack information, enabling the input program to further inspect the presence of *trap functions* on the call stack. The use of this property, however, is deprecated in the current ECMAScript version since this availability of operations makes it more difficult to optimize the execution of the input program [62]. In addition to this, Boa does not include support for `Function.prototype.caller`, thus we can safely assume this does not break the transparency.

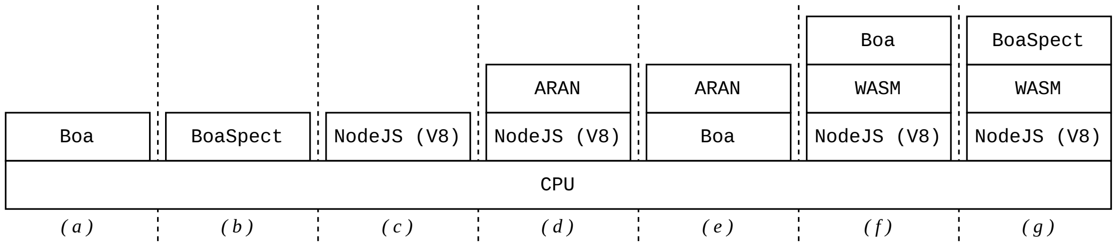


Figure 5.1: The different benchmark experiment setups in terms of the execution environment for the input programs.

5.4 Evaluating Performance

We conduct a set of experiments to evaluate the overhead of BoaSpect and compare it to source code-based instrumentation solutions. These experiments entail running a set of benchmarks to assess the performance overhead of instrumentation applied to the Sunspider suite. The considered analyses for the performance experiments are “empty.js”, “forward.js”, “logging.js”, and ‘profiling.js”.

Figure 5.1 depicts the different execution configurations on which we performed the benchmarks. To evaluate the overhead of BoaSpect, our experiment setups include the baseline execution for Boa *(a)* and the execution of BoaSpect *(b)*. To compare BoaSpect’s overhead in comparison to a source-code-based instrumentation platform, we use Aran. For comparison to Aran running on top of a real-world engine, we further include the baseline execution for Node *(c)* and the instrumented version of Aran on top of Node *(d)*. The V8 engine, however, has an advantage in terms of performance compared to Boa, as its execution performance has been optimized over the last decade. As such, comparing the slowdown of Aran on Node to BoaSpect may be heavily influenced by the V8 structure. Thus, we also include a setup in which we run Aran on top of Boa *(e)*. As our approach offers portability through running BoaSpect on WebAssembly, we also evaluate the execution of Boa on WASM running on Node *(f)* and as its instrumented counterpart we include BoaSpect on WASM running on Node *(g)*. This enables us to compute the slowdown of BoaSpect when running it on WebAssembly.

Per experiment, we evaluate each input program of the benchmark suite for a total of 35 runs either once for regular evaluation or per analysis for instrumented evaluation. The first five runs of the 35 are not taken into account for the rest of our evaluation, as we consider these the *warm-up* runs to allow the execution engine to reach a steady state.

For further performance evaluation, we leave out input programs that either crash or do not terminate within our time budget of 30 seconds. Of the 26 input programs that the Sunspider suite offers, two crash on Boa, one crashes for Aran, one does not terminate for Boa or Aran within our time budget and five do not terminate for BoaSpect within our time budget, leaving a total of 17 input programs we keep for further performance evaluation. These results are summarized in Table 5.2. We do note, however, that none of the programs left out reports a crash for BoaSpect which does not crash for Boa, which otherwise would indicate our instrumentation platform did not behave as expected.

In the bar charts plotted in subsequent subsections, each bar corresponds with the median of the 30 runs of a given experiment. The y-axis is plotted on a logarithmic scale for the sake of readability, the x-axis represents the different input programs from the Sunspider benchmark suite.

Input program left out	Setup	Reason
date-format-tofte.js	Aran on Node	Crash
3d-raytrace.js	Aran on Node, Boa	Timeout
crypto-aes.js	Boa	Crash
date-format-xparb.js	Boa	Crash
regexp-dna.js	Boa	Timeout
string-tagcloud.js	Boa	Timeout
access-nsieve.js	Boa	Timeout
bitops-nsieve-bits.js	BoaSpect	Timeout
string-unpack-code.js	BoaSpect	Timeout

Table 5.2: The set of files from the Sunspider benchmark suite that are left out from the evaluation of BoaSpect, including the interpreter which does not support the file and the reason why the file is not included.

5.4.1 Absolute Time to Execute: Node Against Boa

In this section, we assess the performance difference between Aran on Node and BoaSpect in terms of time to execute the Sunspider suite for the “forward.js” analysis. Recall from Section 5.1.2 that the “forward.js” analysis instruments every operation without performing analysis code, making it suitable for comparing two instrumentation techniques. To this end, we employ experiment setup *(b)* and *(d)*. Figure 5.2 depicts the resuts of such experiment.

We observe a significant difference between the two: BoaSpect is always outperformed by Aran on Node. The closest BoaSpect comes relative to the performance of Aran on Node is for the input program “math-partial-sums”, for which Boa is about 11 times slower than Node. The largest relative performance gap between the two is for the input program “3d-morph”, for which Boa is about 867 times slower. The average relative slowdown for this figure is a factor of 150.

To assess the influence of the base execution engine on the performance of the instrumentation platform, we conduct a second experiment comparing the performance of the execution engines without instrumentation. Figure 5.3 shows similar significant performance differences compared to Figure 5.2, but this case for uninstrumented execution. The smallest relative difference is for the input program “math-partial-sums”, for which Boa is about 32 times slower than Node. The largest relative difference is for the input program “3d-morph”, for which Boa is about 11301 times slower. The average relative slowdown for this figure is a factor of 1390.

Our observations indicate significant differences between the baseline performance of both engines. This comes little as a surprise, given the maturity of both engines. Boa is a young project from 2018 where most of the engineering effort goes to comply with the ECMAScript standards. Node and the underlying V8 engine, however, are industrial products with large teams of engineers working on optimisations. To be able to have a fair comparison with Aran, we will conduct the rest of the experiments on the same execution engine.

5.4.2 Slowdown for Instrumentation with Boa as the Engine

In this section, we assess the performance of Aran and BoaSpect when they both run on Boa. To this end, we compute the slowdown for an input program and an analysis where the baseline is the uninstrumented version of the input program. In Figure 5.4 these are grouped per analysis per program, and are from left to right the following: *Aran on Boa (e)* relative to *Boa (a)* to compute the slowdown for Aran when executing on Boa, *BoaSpect (b)* relative to *Boa (a)* to compute the slowdown of BoaSpect, and *BoaSpect on WebAssembly (g)* relative to *Boa on*

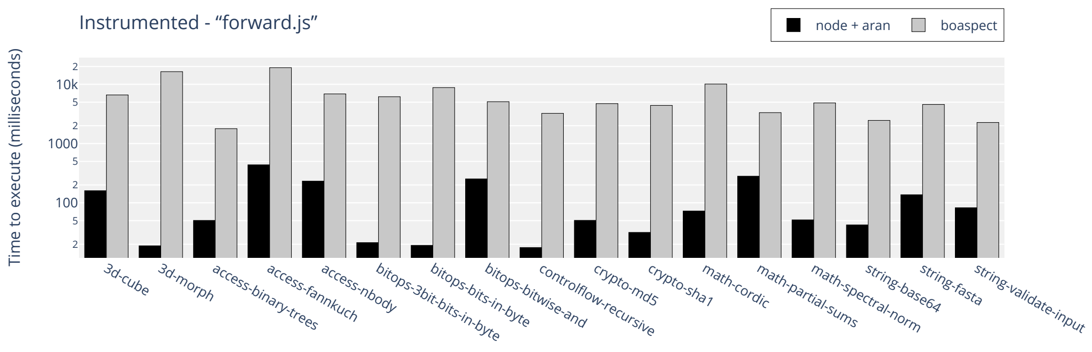


Figure 5.2: A bar chart plotting the absolute time in milliseconds it takes to instrument input programs of the Sunspider suite for the “forward.js” analysis specification. This chart compares the experiment setups for running Aran on top of Node and our instrumentation platform BoaSpect.

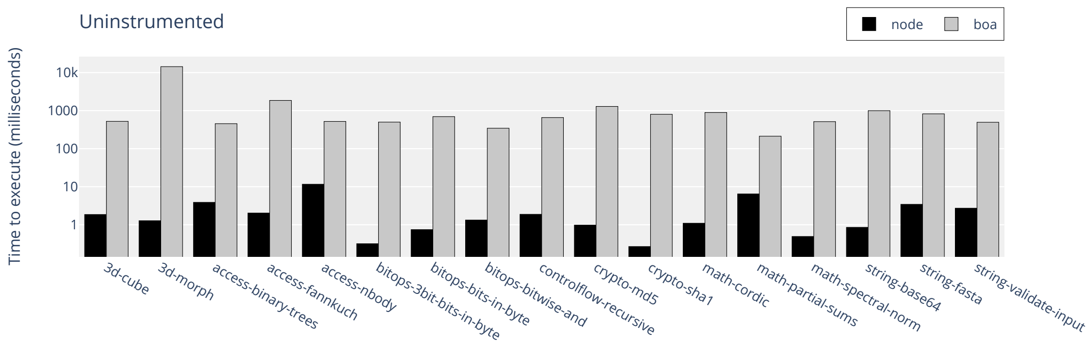


Figure 5.3: A bar chart plotting the absolute time in milliseconds it takes to execute input programs of the Sunspider suite. This chart compares the experiment setups for the Node execution engine and the Boa execution engine.

WebAssembly (f) to compute the slowdown of the portable approach of BoaSpect[8].

From the plots in Figure 5.4 we draw the following observations:

- **Aran is outperformed by BoaSpect when ran on Boa**
  We observe that regardless of the input program or the analysis specification, BoaSpect outperforms Aran. While BoaSpect highest slowdown report is 44.03 for the analysis "logging.js" and the input program "bitops-bitwise-and.js", Aran reports its highest slowdown factor of 261.73 for the same analysis but for the input program "math-partial-sums.js". For the "empty.js" analysis Aran is on average 3.12 times slower than BoaSpect, for the analysis "forward.js" Aran is on average 3.7 times slower, for the analysis "logging.js" this factor is 3.06 and for "profiling.js" this is 5.85.

- **BoaSpect's slowdown remains similar when running on WebAssembly**
  As we motivate our implementation to be portable by running it on WebAssembly, ideally the properties of our approach should remain unaffected for targetting WebAssembly. We conclude from the slowdown reports that the slowdown difference between Aran and BoaSpect are far more significant than the differences between running BoaSpect on a CPU or on WebAssembly.

  Interestingly, however, it is not the case that the slowdown factors are always lower for one comparison or the other. Depending on the input program and the analysis specification, there is some variation.

- **The minimal overhead of BoaSpect's presence is lower than that of Aran**
  The evaluation of BoaSpect with the "empty.js" analysis showcases that the presence of the instrumentation platform has a larger impact on Aran than it has for BoaSpect. The Sunspider benchmark suite is slowed down by a factor in the range $[1, 1.54]$ for BoaSpect, for Aran, this factor is in the range $[1, 23.47]$.

- **The minimal overhead of BoaSpect for instrumenting every supported trap is lower than that of Aran**
  The evaluation of BoaSpect with the "forward.js" analysis showcases that the presence of the instrumentation platform where every trap is instrumented has a larger impact for Aran than it has for BoaSpect. These differences are, however, not near as explicit compared to the differences for the "empty.js" analysis. The Sunspider benchmark suite is slowed down by a factor in the range $[1.13, 15.60]$, for Aran, this factor is in the range $[1.21, 125.90]$.

- **The type of analysis plays a key role in the performance overhead**
  Our results also confirm our hypothesis that the choice of dynamic analysis heavily influences the overall application slowdown. This is due to how one analysis might require the use of all available traps and perform a set of heavy computations while another might decide to implement a single trap followed by a lightweight computation. This difference can be derived from Figure 5.4, by investigating the difference in general trend for the "logging.js" analysis and the "profiling.js" analysis. The former reports slowdowns in the range $[1.34, 261.73]$ while the latter reports slowdowns in the range $[1.03, 77.15]$. Furthermore, the performance overhead for the "profiling.js" analysis targets function applications. This exclusive trap selection also affects which application suffers more from a slowdown, as the slowdown trend between "forward.js" and "logging.js" is altogether different from the slowdown trend between "forward.js" and "profiling.js".

[8]Note that there are missing bars for the WebAssembly variant evaluation of BoaSpect for input program "controlflow-recursive.js". This is because BoaSpect crashed due to the stack exceeding the WebAssembly allowed memory limit.

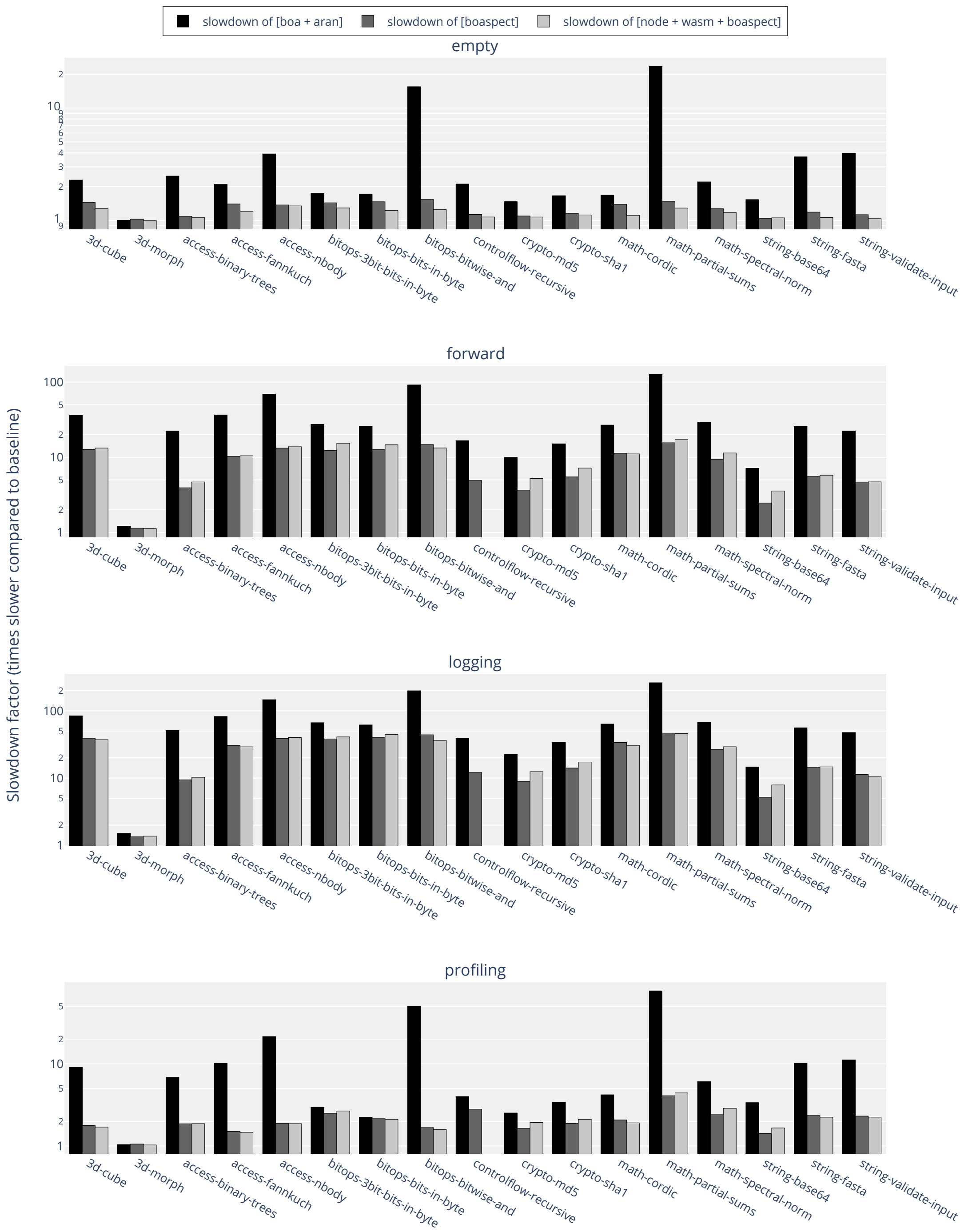


Figure 5.4: A set of bar charts plotting the slowdown factors for executing input programs of the Sunspider suite for different analyses. The charts compare the slowdown factors for running Aran on Boa, running BoaSpect, and running BoaSpect on WebAssembly in Node.

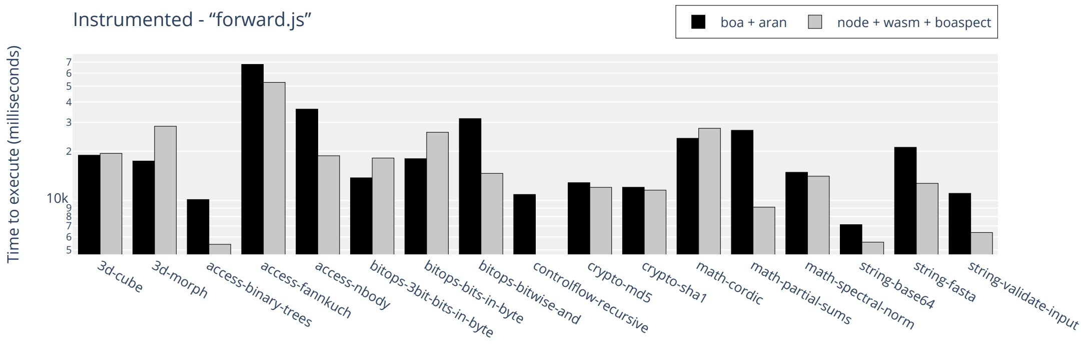


Figure 5.5: A bar chart plotting the absolute time in milliseconds it takes to execute input programs of the Sunspider suite. This chart compares the experiment setups for running Aran on top of Boa and running BoaSpect on WebAssembly on top of Node.

With these insights we conclude that the minimal overhead for instrumenting every supported operation lies in the range of a slowdown factor of $[1.13, 15.60]$, with the highest report for a profiling analysis reaching 44.03.

Moreover, the slowdown of Aran (on Boa) compared to the slowdown of BoaSpect indicates that implementing an instrumentation platform at the level of the interpreter proves to be more beneficial in terms of performance overhead. Our results indicate the interpreter-level instrumentation approach may slow the application 3–5 times less down compared to source-code-level instrumentation.

5.5 Evaluating Portability

As mentioned in Section 2.3, *portability* is concerned with how easily one can adapt the instrumentation to a variety of hosts. Section 5.4 discusses our benchmark setup to run the Sunspider suite on BoaSpect, running on WebAssembly which confirms our implementation is portable and accessible in other host environments.

In terms of portability, we aim to qualitatively assert how our *portable approach* compares to the source code instrumentation which is portable by default. Specifically, we aim to answer the question "Does BoaSpect outperform Aran if BoaSpect runs on WebAssembly?". This is to say if an analysis developer had to choose between the portable Aran approach or the portable BoaSpect approach, in terms of time to execute. To investigate this, we plot the absolute time it takes to execute the "forward.js" analysis for Aran on Boa *(e)* and BoaSpect on WebAssembly on top of Node *(g)* in Figure 5.5.

Interestingly here the performance results are relatively close and there is no overall best performer. The better-performing instrumentation platform technique depends highly on the input program. Thus, in terms of performance with respect to portability, we conclude BoaSpect's WebAssembly-compatible approach and Aran's implementation are equally competing.

```
ToPrimitive ( input [ , preferredType ] )
    1. If Type(input) is Object, then
        a. Let exoticToPrim be ? GetMethod(input, Symbol.toPrimitive).
        b. If exoticToPrim is not undefined, then
            i. If preferredType is not present, let hint be "default".
            ii. Else if preferredType is string, let hint be "string".
            iii. Else,
                1. Assert: preferredType is number.
                2. Let hint be "number".
            iv. Let result be ? Call(exoticToPrim, input, hint).
            v. If Type(result) is not Object, return result.
            vi. Throw a TypeError exception.
        c. If preferredType is not present, let preferredType be number.
        d. Return ? OrdinaryToPrimitive(input, preferredType).
    2. Return input.

OrdinaryToPrimitive ( O, hint )
    1. If hint is string, then
        a. Let methodNames be "toString", "valueOf".
    2. Else,
        a. Let methodNames be "valueOf", "toString".
    3. For each element name of methodNames, do
        a. Let method be ? Get(O, name).
        b. If IsCallable(method) is true, then
            i. Let result be ? Call(method, O).
            ii. If Type(result) is not Object, return result.
    4. Throw a TypeError exception.
```

Listing 16: The ECMAScript specifications of the abstract operations *ToPrimitive* and *OrdinaryToPrimitive*. The specifications define how a non-primitive JavaScript value is converted to a primitive value at the level of interpretation. These operations are internally performed by an interpreter and are not available at the level of the source language.

5.6 Evaluating Expressiveness

Recall from Section 2.3 that expressiveness is the property concerning the capability of the platform to provide the analysis developer with means of instrumenting given program properties.

We mention in Section 2.4 that the *expressiveness* of an instrumentation platform is typically higher at the interpreter-level than the source-code-level. As explained in Chapter 4, BoaSpect's offer to analsis developers in an API based on the one of Aran including an *trap* `toPrimitive`. In what follows we discuss how developers need to work around the lack of this trap when using an API like Aran.

The lack of an abstract operation *ToPrimitive* in an instrumentation platform means that there is no trap to query whenever the act of converting non-primitive values to primitive values through calls to *ToPrimitive* takes place. The lack of this trap forces developers to derive the information directly in the analysis, implying that the analysis developer will have to model *ToPrimitive* and its dependency *OrdinaryToPrimitive* themselves.

One could reify this information at the source code level. The reification of whenever the abstract operation *ToPrimitive* is performed, would require us firstly to model the operation *ToPrimitive* and its dependancy *OrdinaryToPrimitive*, another abstract operation, as JavaScript functions. This reification is required as our goal is not only to intercept these operations but also to provide the means to compute the expected outcome to ensure the instrumentation platform can remain transparent with respect to the input program semantics. The definitions of both abstract operations as they are defined by the ECMAScript standards are given in Listing 16.

To correctly implement that the instrumentation platform intercepts whenever *ToPrimitive* is called, we would need to also model the operations that rely on *ToPrimitive* and ensure these now call our definition of *ToPrimitive* or the *trap* `toPrimitive` depending whether the operation at hand is subject to instrumentation. According to the ECMAScript specification for both operations shown in Listing 16, the abstract operation *ToPrimitive* is a dependency of 11 other operations[9]. If these additional 11 operations were available to Aran to instrument, such as the "+" operator currently is, it would be a matter of rewriting the corresponding operation to ensure they call into our trap definition of *ToPrimitive*. However, this is not the case as the specification of the JavaScript-level "+' operator does not use *ToPrimitive* directly but makes use of the abstract operation *ApplyStringOrNumericBinaryOperator*, which in turn makes use of *ToPrimitive*. This implies that we are required to reimplement the "+" operator, and also the additional infrastructure of operations other than *ToPrimitive* on which the implementation of "+" depends to correctly model and thus reify the "+" operator. Notice the dependency that grows both ways for modelling the *ToPrimitive* for instrumentation, on one hand, we need to model "ApplyStringOrNumericBinaryOperator" and "+" to ensure they call into our *ToPrimitive* definition, but on the other hand we need to provide the dependencies for "ApplyStringOrNumericBinaryOperator" and "+" such that they can be correctly implemented.

Visually, the dependency graph and this growth are depicted in Figure 5.6. The operations that rely on *ToPrimitive* need to be reified all the way "up" in the dependency chain until the point where the source-level operation is available (highlighted in green), as this can then be rewritten through source-code transformation. For this example this concerns the definitions of *ApplyStringOrNumericBinaryOperator* and *EvaluateStringOrNumericBinaryExpression*. But in the same way, we would need to reify the (abstract) operations "down" in the dependency chain until the point where all required infrastructure is available for the abstract operations we aim to intercept for instrumentation. For this example, this concerns the definition of *OrdinaryToPrimitive*, but also all other dependancies of *OrdinaryToPrimitive*, *ApplyStringOrNumericBinaryOperator*, and *EvaluateStringOrNumericBinaryExpression*.

The required efforts for supporting the *trap* `toPrimitive` only grow in size as soon as the aspect of the host environment of the interpreter is included. The abstract operation *ToPrimitive* is one on which the host environment may depend to turn JavaScript values into values of a type which aligns with the format of the host. For example, a web browser that hosts a JavaScript engine may require JavaScript values to be turned into values of type string such that it can display the content of these values on a web page. For these use cases, the host may also rely on the abstract operation *ToPrimitive*. Thus in the case of Aran, this would require the instrumentation platform to intercept these operations with the host too. This is depicted in Figure 5.6 with the additional host layer that calls into the *ToPrimitive* operation.

The growth of this example showcases how the inaccessibility of source-code instrumentation is overcome through implementing parts of the interpreter at the source-code level. For the ever-growing requirements of the analysis in terms of hooking into more operations, the source-code level instrumentation platform will be required to reify more and more aspects of the interpreter. We believe this growing reification of the interpreter aspects turns the source-code instrumentation platform partially into a meta-circular interpreter to overcome the limitations of information otherwise only available to the interpreter, further driving the performance overhead of source-code instrumentation towards the performance of a meta-circular instrumentation platform.

To overcome the aforementioned issues to reify primitives in a source-code-based platform, it could be possible to inject proxies that specify a source-level *ToPrimitive* operation by overriding

[9]The ECMAScript operations that have *ToPrimitive* at least once as one of its dependancies are "ToNumeric", "ToNumber", "ToBigInt", "ToString", "ToPropertyKey", "IsLessThan", "IsLooselyEqual", "ApplyStringOrNumericBinaryOperator", "BigInt", "Date", and "Date.prototype.toJSON".

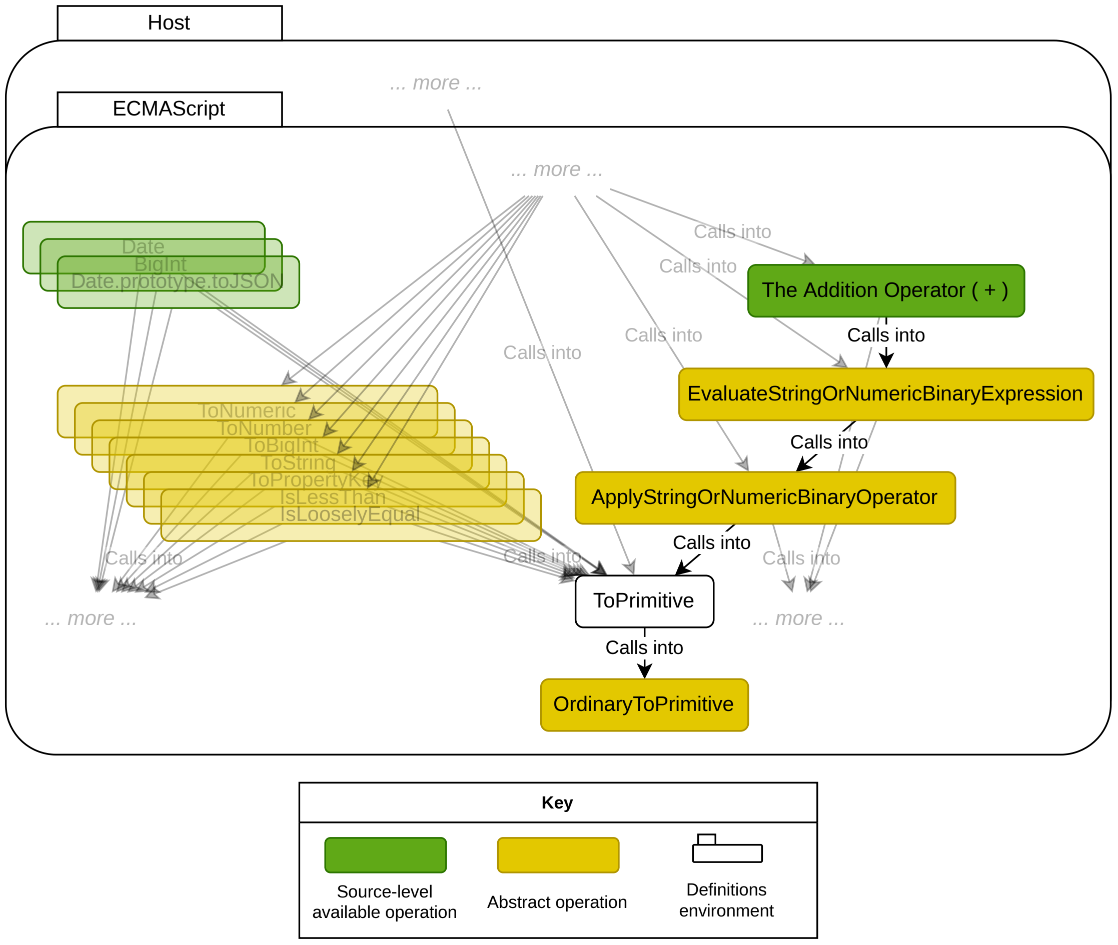


Figure 5.6: A depiction of the abstract operations *ToPrimitive* and *OrdinaryToPrimitive* as they are defined in the ECMAScript specification, including the operations that call into *ToPrimitive* and *OrdinaryToPrimitive*. The schematic also illustrates how additional operations of the surrounding host may hook into the abstract operations such as *ToPrimitive*, bypassing the source-level operations that may yield the same result.

the proxy's `toPrimitive` property as explained in Section 3.2.2, which the interpreter will hook into as specified by instruction *(iv)* in the definition of *ToPrimitive* in Listing 16. This alternative approach, however, would provide a trap function for the abstract operation *OrdinaryToPrimitive* rather than *ToPrimitive* as its call would be overlooked for non-primitive values. Furthermore, this alternative approach would not scale for instrumenting other abstract operations for which there is no source-code level mechanism to reimplement, such as the other illustrated abstract operations in Figure 5.6 other than *OrdinaryToPrimitive*.

This concludes our discussion on how to add support for the *trap* `toPrimitive` to Aran. The source-code instrumentation platform is required to, to some extent, simulate this information at the source-code level, which introduces additional complexity and instrumentation code for computing values that are readily available at the level of the interpreter. When we compare this to the required efforts of adding support for the *trap* `toPrimitive` to BoaSpect, there is a clear difference. As Section 4.2.3 explains, the majority of the efforts boil down to the reification of the available information at the level of the interpreter. This allows for the interpreter-based instrumentation platform to reuse the existing infrastructure of the interpreter, saving on required engineering efforts, additional runtime computations at the level of source-code and memory use compared to instrumentation at the source-code level.

These benefits further apply for *introspection* for additional available information to the interpreter. For including *intercession* support, the required efforts, in general, become more complex. For example, the reification of the environment to inspect the available variables when calling a *trap function* would require reifying the ECMAScript *Execution Context* such that it is available to the analysis. To enable *intercession* on this reified representation, it involves the use of abstract operations such as *PutValue* that may further modify the environments in the execution context.

5.7 Evaluation w.r.t. the Research Questions

In this section, we repeat each research question, followed by our answer.

RQ1. *How can one extend a JavaScript interpreter to provide an instrumentation interface on which others can build an analysis?*
The answer to this research question is by construction in Chapter 4. Here, we showed our approach extends the ECMAScript Execution Context with both a reference to the evaluated advice, and a binary value capturing whether the interpreter is at the level of the source code, *Base*, or the level of the analysis, *Meta*. Furthermore, during the evaluation of corresponding interpreter implementations of ECMAScript operations of interest to the analysis, our approach is to precede the default behaviour to check if the current interpreter state is *Base* and if so, call into the corresponding trap function to proceed with the analysis. We further noted that first-class functions should capture whether their body should be evaluated as *Base* or *Meta* code as they enable the context switch.

RQ2. *What is the execution overhead for interpreter-based instrumentation using the Boa interpreter for JavaScript?*
In the performance evaluation we observed BoaSpect's performance overhead to slow the execution of the program under analysis down up to two orders of magnitude. Instrumenting every supported trap function for BoaSpect with no additional analysis code, evaluated with the "empty.js" analysis, reports a slowdown factor in the range $[1.13, 15.60]$. The overhead of BoaSpect for dynamic analyses that profile all operations and model the call stack, we report slowdown factors in the range $[1.03, 44.03]$.

RQ3. *Does the overhead of portable interpreter-based instrumentation outweigh the overhead of source-code instrumentation?*
To answer this question, we compared the performance of BoaSpect running on WebAssembly with the performance of Aran. The results of this comparison, depicted in Figure 5.5, show that both portable instrumentation platforms take similar amounts of time to execute the input programs. However, when BoaSpect does not run on WebAssembly, we report that on average its slowdown factor is 3–5 times faster than the slowdown factor of Aran. This leads us to the conclusion that including the optional portability incurs an overhead which makes the performance overhead similar to that of Aran.

RQ4. *What benefits arise from interpreter-based instrumentation in terms of expressiveness compared to a source code-based solution?*
To evaluate the benefits of BoaSpect in terms of *expressiveness*, we firstly asserted it is possible to port existing analyses from Aran on BoaSpect, “empty.js”, “forward.js”, “logging.js”, and “profiling.js”. We also developed a new analysis that was not possible to implement for Aran because it cannot access the abstract operation *ToPrimitive* at the level of source code. This shows how instrumentation at the level of the interpreter can access and provide the analysis with information that is unavailable at the source code level.

Conclusion

We evaluated BoaSpect with respect to the properties for which one can compare different instrumentation platforms: *transparency*, *performance*, *portability*, and *expressiveness*.

Regarding the property *transparency*, we showed that BoaSpect avoids transparency issues that arise when one would rewrite the program under analysis as source-code instrumentation does. At the level of source code instrumentation, this affects the result of calling reflective JavaScript operations such as `Function.toString`, `Function.prototype.name`, or the `arguments` object. Our approach hides away instrumentation information from the source code within interpreter internals that is not accessible at the level of source code, thus maintaining a high level of *transparency*.

Using the set of benchmarks from the Sunspider suite we evaluated BoaSpect in terms of the property *performance*. First, we concluded that the V8 execution engine is several orders of magnitude faster than Boa, which is why we restrict further performance evaluation to use Boa as our baseline execution engine. Second, we concluded that Aran is always slower than BoaSpect. For BoaSpect, one can expect the program under analysis to run up to 44.03 times slower. On average BoaSpect’s slowdown reports are 3–5 times faster than the slowdown reports for Aran.

For the property *portability*, we successfully compiled BoaSpect to WebAssembly and evaluated how this affects the instrumentation platform performance overhead for the Sunspider suite. Here, we report that the time to execute an input program under instrumentation on BoaSpect running in WebAssembly runs at a similar performance to running Aran on Boa running directly on our benchmark system. For this, we conclude that our approach leaves the choice for an analysis developer to opt for BoaSpect as a portable approach with the additional benefits from the other properties.

In terms of the property *expressiveness*, we concluded that information exclusively available to the interpreter is either not possible to expose or requires a lot of additional operations to make it available for a source code instrumentation approach. We concluded this by addressing addressed the shortcomings of source code instrumentation while showing how issues reduce to non-issues for BoaSpect.

With our evaluation we conclude that instrumentation at the level of the interpreter is a strong competitor compared to the other instrumentation techniques, providing portability as a tradeoff for performance.

Chapter 6

Conclusion

In this work, we explored a novel approach to implementing an instrumentation platform for JavaScript. For JavaScript, static analysis tools quickly fail to approximate the runtime behaviour of a program as the dynamic features of JavaScript rapidly exceed the approximation capabilities of static analysis. Dynamic analysis tools, however, execute alongside a concrete program run to overcome the limitations of approximating program behaviour [3]. To this end, research has focused on instrumentation platforms for JavaScript to facilitate the implementation and inclusion of dynamic analyses for a given target program.

In Chapter 2, we discussed on the different properties for which one can compare different instrumentation platform implementation techniques: *transparency*, *performance*, *portability*, and *expressiveness*.

As the target programs may operate in a variety of hosts such as client-side browsers or NodeJS [26], and we need to minimize the impact of the analysis code to remain as true to the target program execution as possible for correct analysis results, there is a demand for an instrumentation platform offering all properties. The two most promising techniques so far, source code level instrumentation and interpreter-based AST instrumentation, however, offer the analysis developer a tradeoff between either the focus on *portability* or the focus on *transparency*, *transparency* and *expressiveness*.

On the one hand, an instrumentation platform can be implemented at source code level through source code transformation [9, 10]. This technique injects the analysis and instrumentation operations in the target program at the level of source code, by rewriting the target program into a variant program in which operations of interest are transformed into function calls to the dynamic analysis. This technique offers high *portability* as the instrumented application can run on any JavaScript engine, but suffers from *performance*, *transparency* and *expressiveness* issues as the instrumentation platform operates at the same level as the target application.

Alternatively, instrumentation platforms may be implemented in the interpreter. This technique modifies a JavaScript execution engine such that during the evaluation of operations of interest in the target program, the analysis is informed about the operation at hand. A well-known implementation is NodeProf [11], which wraps the parsed program abstract syntax tree (AST) nodes with additional operations that are further executed by the interpreter. This technique offers high *performance*, *transparency* and *expressiveness* due to the instrumentation code operating at a different level from the target program, but suffers from *portability* as the implementation of the JavaScript engine cannot easily be ported to run in other JavaScript engines.

With the aforementioned implementation techniques for instrumentation platforms for JavaScript, analysis developers face the choice between either *portability* or *performance*, *trans-*

parency and *expressiveness*. Preferably, one would have both *portability* to analyse code which runs in browsers and servers while maintaining *performance*, *transparency* and *expressiveness* to enable the analysis to trace an execution true to the intended target application behaviour.

6.1 Our Approach

We explore an instrumentation platform that offers *portability* to run analyses on different JavaScript engines, while maintaining *performance*, *transparency* and *expressiveness* to enable the analysis to trace an execution true to the intended target application behaviour. We created BoaSpect which fulfils these ideas in the Boa interpreter.

As JavaScript engines typically compile their input programs at runtime to an intermediate representation to then interpret this representation for faster execution, we target our instrumentation platform to operate at the level of this intermediate representation interpretation to maintain high *performance*.

Furthermore, given that BoaSpect operates within the interpreter it is possible to more easily remain *transparent* and provide a more *expressive* interface to the analysis developer.

To achieve *portability*, we chose Boa as our starting engine as it offers the means to compile to WebAssembly [13], a compilation target that JavaScript environments can execute at high speed. Our implementation of BoaSpect then offers the analysis developer the choice to instrument their application running in any WebAssembly compatible JavaScript host.

Our implementation of BoaSpect offers an instrumentation interface for the analysis developer based on the interface provided by source code instrumentation platforms, in particular the interface of Aran. However, we augment our interface with support to trace the execution of the abstract ECMAScript operation *ToPrimitive*, which cannot be provided by source code instrumentation.

A dynamic analysis specifically is just a regular JavaScript program that evaluates to an *advice* object, which implements the instrumentation-compatible trap functions that the instrumentation platform will call into. For example, if the advice implements the trap function `apply`, the instrumentation platform will intercept all function applications and call the `apply` trap with the available information of the intercepted operation.

6.1.1 Interpreter-Based Instrumentation

During our implementation discussion in Chapter 4, we faced several challenges to implement BoaSpect.

Ensuring transparency – To maintain *transparency* one must separate the advice from the target program such that the target program cannot access it. To this end, we include a second entry point for the advice specification as a separate input program and maintain a reference to the advice from within the interpreter, inaccessible to the target program.

As the interpreter is both responsible for evaluating the target program and the analysis, one is required to distinguish evaluating code of the target program from evaluating code of the analysis to avoid running the risk of the analysis instrumenting itself which can lead to unbound recursion. To this end, we augment first-class functions with a flag to tell the interpreter whether their body should be evaluated as code that is subject to instrumentation or not. Transitively, functions created in contexts that are subject to instrumentation will carry the flag that they are subject to instrumentation too, and vice versa functions created in the evaluation context of the analysis will carry the flag that they should be evaluated as code not subject to instrumentation.

Hooking into the operations – During the evaluation of the target program, the instrumentation platform must correctly hook into the operations of interest that take place when evaluating the target program to call the respective trap function with the operation information.

To this end, we identified a recurring pattern to implement the instrumentation code in Boa, in which (1) the execution context is identified at operations of interest, (2) the available information is reified and (3) the corresponding trap function is called.

We assessed how compatible our instrumentation platform with ECMAScript is by testing 407 unit tests from 3 different benchmark suites. Out of the 407 tests, all 407 passed.

6.2 Evaluation

In Chapter 5 we evaluated BoaSpect with respect to the four properties to compare instrumentation platforms.

- **Transparency**
  To evaluate the *transparency* of BoaSpect we evaluated how reflective JavaScript constructs do not leak the presence of BoaSpect and we compare it to a source code instrumentation platform, i.e. Aran.

  In particular, we compare how reflective constructs like `Function.toString`, which evaluates to the function body, are leaking for Aran the transformed function body while for BoaSpect this is not the case as the instrumentation code is hidden from the source code, preventing leakage of its presence.

- **Performance**
  We evaluated the *performance* of BoaSpect with experiments that benchmark the slowdown of the target application execution by the presence of instrumentation for the Sunspider benchmark suite. To this end, we concluded that BoaSpect slows the target application up to two orders of magnitude down. For source code instrumentation we identified with Aran that for the same input programs and analyses the slowdown is higher at all times, slowing down the target application up to three orders of magnitude.

- **Portability**
  We evaluated the *portability* of BoaSpect by running its instrumentation within the NodeJS JavaScript environment. Here, we identify that when running BoaSpect on WebAssembly for increased portability the overall execution slows down such that its time to execute the instrumented target application becomes similar to the time to execute the instrumented target program running on Aran.

- **Expressiveness**
  We evaluated the increased expressiveness by evaluating how BoaSpect compares to source code instrumentation when building an analysis that requires trapping value conversion to primitives.

  To this end, we observed that BoaSpect much more easily exposes information available to the interpreter which is either inaccessible for source code instrumentation or requires much more engineering effort to simulate this information as it requires to mimic what the JavaScript interpreter does.

6.3 Contributions

In this work we make the following contributions:

1. We propose a novel interpreter-based instrumentation platform for JavaScript that operates at the level of interpreting the JIT-compiled bytecode as abstract ECMAScript operations. The novelty is the increased available interpreter information, which can be used for dynamic analyses whereas source code-based solutions require higher coding efforts from the developer.

2. We provide an implementation of our approach, BoaSpect, which is implemented in the JavaScript engine Boa. Moreover, our implementation can also be compiled to WebAssembly, making the approach appeal as a portable instrumentation platform that can be shipped as a whole to run within other JavaScript environments.

3. We propose a novel trap `toPrimitive` that instruments the abstract ECMAScript operation *ToPrimitive*. On top of our trap specification, we showed a dynamic analysis specification in Listing 9, which makes use of this trap to uncover ill-formed or potentially malicious programs as we explain in Section 3.2.2.

6.4 Future Work

We see three avenues for future work for our approach and the development of BoaSpect.

Joinpoints and pointcuts

For BoaSpect, and for our evaluation with Aran, we derived the joinpoints by inspecting the implemented trap functions of the advice, such as "instrument *all* function calls" or "instrument *all* binary operations". However, we would like to explore support for targetting a group of individual points of interest. Similar to aspect-oriented programming, we could provide analysis developers with the ability to target individually pointcuts. This would allow the advice specification to be informed about the specific location in the program where a specific operation takes place.

We note that this extension is orthogonal to our instrumentation implementation. To address this, a starting point would be to implement changes at the abstract syntax tree to augment candidate tree nodes with information that is provided to the analysis at the moment when the operation is instrumented, such as Aran supports *serial numbers* to identify the instrumented nodes [12].

Portability with host-supporting operations

Currently, most of our evaluation limits interoperation between BoaSpect and its host. After all, the JavaScript engine becomes more capable in terms of *real-world applications* as soon as its execution interacts with the host such as affecting the content on a web browser or interacting with a file system. To this end, Boa's public API is being developed to facilitate interactions with the host environment[1]. This could serve as an entry point for developing an interface for BoaSpect to interact with a host environment such as a web browser when porting BoaSpect to WebAssembly.

Enabling analysis development at the level of the interpreter

Another approach to enable the analysis developer to develop their analyses would expose

[1]https://github.com/boa-dev/boa/discussions/1531

an API through which an analysis can directly communicate with the interpreter. This would not be limited to the use of the Rust language in which Boa is implemented, nor would it require recompilation for each reimplementation of an analysis if the analysis API would be designed to make use of the proposed WebAssembly interface types. This proposal for WebAssembly allows a WebAssembly module to interoperate with other APIs at a raised level of abstraction which allows communication through richer data types than exclusively numbers. This would allow one to, for example, implement their analysis in C++ which would communicate with the Boa interpreter which is developed in Rust.

We deliberately opt for JavaScript as our implementation language, however, Sun et al. [11] mention that one can benefit from a lower overhead by implementing the analysis closer to the interpreter.

Bibliography

[1] Boadev Group. Boa: An embeddable and experimental javascript engine written in rust. `https://github.com/boa-dev`, August 2022. (Accessed on 08/20/2022).

[2] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Understanding asynchronous interactions in full-stack javascript. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1169–1180, 2016. doi: 10.1145/2884781.2884864.

[3] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for javascript. *ACM Comput. Surv.*, 50(5), September 2017. ISSN 0360-0300. doi: 10.1145/3106739.

[4] Anders Møller and Michael I. Schwartzbach. Static program analysis, October 2018. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa`.

[5] Wögerer Wolfgang. A survey of static program analysis techniques. Technical report, Citeseer, 2005.

[6] Thomas Ball. The concept of dynamic analysis. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 216–234, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48166-9.

[7] Florent Marchand de Kerchove, Jacques Noyé, and Mario Südholt. Open Scope: A Pragmatic JavaScript Pattern for Modular Instrumentation. working paper or preprint, 2015. URL `https://hal.archives-ouvertes.fr/hal-01181143`.

[8] Erick Lavoie, Bruno Dufour, and Marc Feeley. Portable and efficient run-time monitoring of javascript applications using virtual machine layering. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 541–566, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-44202-9.

[9] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail: A general-purpose platform for shadow execution of javascript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 260–270, 2016. doi: 10.1109/SANER.2016.91.

[10] 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*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491447.

[11] 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*, CC 2018, page 196–206, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356442. doi: 10.1145/3178372.3179527.

[12] Laurent Christophe. Github - lachrist/aran: Javascript code instrumenter. `https://github.com/lachrist/aran`, May 2022. (Accessed on 05/15/2022).

[13] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062363.

[14] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Pearson/Addison Wesley, 3rd edition, 2006.

[15] Donald Bradley Roberts. *Practical analysis for refactoring.* University of Illinois at Urbana-Champaign, 1999.

[16] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Trans. Softw. Eng. Methodol.*, 25(1), December 2015. ISSN 1049-331X. doi: 10.1145/2744200.

[17] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, page 86–96, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135629. doi: 10.1145/566172.566186.

[18] Tosapon Pankumhang and Matthew Rutherford. Iterative instrumentation for code coverage in time-sensitive systems. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015. doi: 10.1109/ICST.2015.7102594.

[19] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936332. doi: 10.1145/1250734.1250746.

[20] Feng Qin, Shan Lu, and Yuanyuan Zhou. Safemem: exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *11th International Symposium on High-Performance Computer Architecture*, pages 291–302, 2005. doi: 10.1109/HPCA.2005.29.

[21] 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)*, pages 655–670, San Diego, CA, 2014. USENIX Association. ISBN 978-1-931971-15-7.

[22] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, page 1663–1671, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450324694. doi: 10.1145/2554850.2554909.

[23] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.

[24] Mozilla. Firefox web browser. `https://www.mozilla.org/en-US/firefox/`, May 2022. (Accessed on 05/15/2022).

[25] Google. Chrome web browser. `https://www.google.com/chrome`, May 2022. (Accessed on 05/15/2022).

[26] OpenJS Foundation. Node.js is a javascript runtime built on chrome's v8 javascript engine. `https://nodejs.org/en/`, May 2022. (Accessed on 05/15/2022).

[27] ECMA International. Ecma-262: Ecmascript 2021 language specification. `https://www.ecma-international.org/publications-and-standards/standards/ecma-262/`, May 2022. (Accessed on 05/19/2022).

[28] Inc. Apple, Inc. Google, Inc. Mozille, and Inc. Microsoft. Web hypertext application technology working group: Standards. `https://spec.whatwg.org/`, May 2022. (Accessed on 05/19/2022).

[29] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 1–12, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300193. doi: 10.1145/1806596.1806598.

[30] Laurence Tratt. Chapter 5 dynamically typed languages. volume 77 of *Advances in Computers*, pages 149–184. Elsevier, 2009. doi: https://doi.org/10.1016/S0065-2458(09)01205-4.

[31] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do. In Mira Mezini, editor, *ECOOP 2011 – Object-Oriented Programming*, pages 52–78, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22655-7.

[32] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 49–59, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. doi: 10.1145/2610384.2610385.

[33] Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013. doi: 10.1109/SCAM.2013.6648192.

[34] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. Dlint: Dynamically checking bad coding practices in javascript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 94–105, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. doi: 10.1145/2771783.2771809.

[35] Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 251–262, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312059. doi: 10.1145/2254064.2254095.

[36] Liang Gong, Michael Pradel, and Koushik Sen. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 357–368, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2786831.

[37] Xiao Xiao, Shi Han, Charles Zhang, and Dongmei Zhang. Uncovering javascript performance code smells relevant to type mutations. In Xinyu Feng and Sungwoo Park, editors, *Programming Languages and Systems*, pages 335–355, Cham, 2015. Springer International Publishing. ISBN 978-3-319-26529-2.

[38] Vincent St-Amour and Shu yu Guo. Optimization Coaching for JavaScript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 271–295, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-86-6. doi: 10.4230/LIPIcs.ECOOP.2015.271.

[39] Laurent Christophe, Coen De Roover, Elisa Gonzalez Boix, and Wolfgang De Meuter. Orchestrating dynamic analyses of distributed processes for full-stack javascript programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2018, page 107–118, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360456. doi: 10.1145/3278122.3278135.

[40] Florent Marchand de Kerchove, Jacques Noyé, and Mario Südholt. Towards modular instrumentation of interpreters in javascript. In *Companion Proceedings of the 14th International Conference on Modularity*, MODULARITY Companion 2015, page 64–69, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332835. doi: 10.1145/2735386.2736753.

[41] Jan Kasper Martinsen, Håkan Grahn, and Anders Isberg. Combining thread-level speculation and just-in-time compilation in google's v8 javascript engine. *Concurrency and computation: practice and experience*, 29(1):e3826, 2017.

[42] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542528.

[43] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 662–676, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. doi: 10.1145/3062341.3062381.

[44] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324724. doi: 10.1145/2509578.2509581.

[45] Christian Wimmer, Vojin Jovanovic, Erik Eckstein, and Thomas Würthinger. One compiler: Deoptimization to optimized code. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 55–64, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450352338. doi: 10.1145/3033019.3033025.

[46] Koushik Sen, Manu Sridharan, and Esben Sparre Andreasen. Github - ksen007/jalangi2: a framework for writing dynamic analyses for javascript. `https://github.com/ksen007/jalangi2`, May 2022. (Accessed on 05/21/2022).

[47] Ariya Hidayat. Esprima: Ecmascript parsing infrastructure for multipurpose analysis. `https://esprima.org/`, May 2022. (Accessed on 05/21/2022).

[48] Estools. Github - estools/escodegen: Ecmascript code generator. `https://github.com/estools/escodegen`, May 2022. (Accessed on 05/21/2022).

[49] acornjs. Github - acornjs/acorn: A small, fast, javascript-based javascript parser. `https://github.com/acornjs/acorn`, May 2022. (Accessed on 05/21/2022).

[50] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 165–178, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310833. doi: 10.1145/2103656.2103677.

[51] Jeff Terrace, Stephen R. Beard, and Naga Praveen Kumar Katta. JavaScript in JavaScript (js.js): Sandboxing Third-Party scripts. In *3rd USENIX Conference on Web Application Development (WebApps 12)*, pages 95–100, Boston, MA, 2012. USENIX Association. ISBN 978-931971-94-2.

[52] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, page 133–144, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311762. doi: 10.1145/2151024.2151043.

[53] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. Towards transparent introspection. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 248–259, 2016. doi: 10.1109/SANER.2016.25.

[54] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.

[55] James Ide, Rastislav Bodik, and Doug Kimelman. Concurrency concerns in rich internet applications. 2009.

[56] Veselin Raychev, Martin Vechev, and Manu Sridharan. Effective race detection for event-driven programs. *SIGPLAN Not.*, 48(10):151–166, October 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509538.

[57] Shin Hong, Yongbae Park, and Moonzoo Kim. Detecting concurrency errors in client-side java script web applications. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 61–70, 2014. doi: 10.1109/ICST.2014.17.

[58] Fraser Brown, Shravan Narayan, Riad S. Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, 2017. doi: 10.1109/SP.2017.68.

[59] Mitchell Wand and Daniel P. Friedman. *Essentials of programming languages ; Third edition.* MIT Press, 2008.

[60] Madhukar N. Kedlaya, Behnam Robatmili, Cundefinedlin Caşcaval, and Ben Hardekopf. Deoptimization for dynamic language jits on typed, stack-based virtual machines. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, page 103–114, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327640. doi: 10.1145/2576195.2576209.

[61] Rustwasm. Rust and webassembly documentation. `https://rustwasm.github.io/docs/`, August 2022. (Accessed on 08/19/2022).

[62] Mozilla. Mdn web docs. `https://developer.mozilla.org/en-US/`, August 2022. (Accessed on 08/19/2022).