JASMaint: Portable Multi-language Taint Analysis for the Web*

Abel Stuker
Vrije Universiteit Brussel
Brussels, Belgium
Abel.Stuker@vub.be

Aäron Munsters
Vrije Universiteit Brussel
Brussels, Belgium
amunster@vub.be

Angel Luis Scull Pupo
Vrije Universiteit Brussel
Brussels, Belgium
ascullpu@vub.be

Laurent Christophe
Vrije Universiteit Brussel
Brussels, Belgium
Laurent.Christophe@vub.be

Elisa Gonzalez Boix
Vrije Universiteit Brussel
Brussels, Belgium
egonzale@vub.be

Abstract

Modern web applications integrate JavaScript code with more efficient languages compiling to WebAssembly, such as C, C++ or Rust. However, such multi-language applications challenge program understanding and increase the risk of security attacks. Dynamic taint analysis is a powerful technique used to uncover confidentiality and integrity vulnerabilities. The state of the art has mainly considered taint analysis targeting a single programming language, extended with a limited set of native extensions. To deal with data flows between the language and native extensions, typically taint signatures or models of those extensions have been derived from the extensions' high-level source code. However, this does not scale for multi-language web applications as the WebAssembly modules evolve continuously and generally do not include their high-level source code.

This paper proposes JASMaint, the first taint analysis approach for multi-language web applications. A novel analysis orchestrator component manages the exchange of taint information during interoperation between our language-specific taint analyses. JASMaint is based on source code instrumentation for both the JavaScript and WebAssembly codebases. This choice enables deployment to all runtimes that support JavaScript and WebAssembly. We evaluate our approach on a benchmark suite of multi-language programs. Our evaluation shows that JASMaint reduces overtainting by 0.003%–56.20% compared to an over-approximating taint analysis based on function models. However, this comes at the cost of an increase in performance overhead by a factor of 1.14x–1.61x relative to the state of the art.

CCS Concepts: • **Software and its engineering** → **Object oriented frameworks**; **Dynamic analysis**; • **Information systems** → *Web applications*; • **Security and privacy** → Information flow control.

Keywords: dynamic taint analysis, multi-language, JavaScript, Wasm, source code instrumentation

ACM Reference Format:
Abel Stuker, Aäron Munsters, Angel Luis Scull Pupo, Laurent Christophe, and Elisa Gonzalez Boix. 2025. JASMaint: Portable Multi-language Taint Analysis for the Web. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759426.3760982

1 Introduction

Many modern web applications leverage both JavaScript and WebAssembly to improve performance. WebAssembly (Wasm) is a binary instruction format designed for near-native performance within a stack-based virtual machine [10]. It has also become the compilation target of numerous languages, including Rust, C++, and Go. This has also enabled the portability of applications that were previously limited to native platforms to other environments, such as server runtimes, embedded devices, and the web.

As web applications grow in size and complexity, understanding their runtime behaviour becomes crucial for identifying potential security vulnerabilities and performance issues. This is particularly important, given a recent study of the entire Node Package Manager (NPM) repository [20] which found that installing a single npm package introduces implicit trust in 79 other packages, maintained by 39 different developers. This substantially increases the attack surface, rendering applications susceptible to supply chain

*Aäron Munsters is funded by the Research Foundation Flanders, project number 1S53725N. Angel Luis Scull Pupo is funded by the Cybersecurity Research Program Flanders (CRPF) from the Flemish Government.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MPLR '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-2149-6/25/10
https://doi.org/10.1145/3759426.3760982

attacks [20]. Modern web applications can further incorporate third-party Wasm dependencies, increasing the risk of supply chain attacks in the Wasm realm. This unavoidable implicit trust and interlanguage interactions highlight the importance of comprehensive analysis techniques capable of understanding the behaviour of multi-language web applications to detect and mitigate potential security vulnerabilities.

Dynamic taint analysis is a powerful technique for identifying and mitigating integrity [2, 14] and confidentiality vulnerabilities [1, 7, 14]. It tracks the flow of data between *sources* of sensitive information through the application to raise an alarm when they reach *sinks* of public information. Achieving *precise* taint tracking is challenging when the code of a given operation involving taint information is unavailable as it involves a native extension, e.g., as a built-in functions in the interpreter of the target language. In such cases, the burden falls on the analysis developer to define appropriate semantic rules to ensure precise taint propagation.

While many modern web applications integrate JavaScript and WebAssembly (Wasm) code, a portable dynamic taint analysis that considers the interoperability between JavaScript and Wasm programs does not currently exist. Dynamic taint analysis approaches have been proposed for JavaScript [1, 14] and Wasm [8, 17, 23], but they target only one language in isolation. Whenever the analysis needs to deal with data flows beyond a language boundary (e.g. supporting native extensions or language builtins), dynamic taint analysis approaches often treat the component as a black box [12, 22]. Some works address this by allowing developers to provide taint signatures or a *taint model* to specify the incoming and outgoing flows [1, 6, 11, 12, 14, 22]. However, those models are not guaranteed to be aligned with the semantics of the code and can easily result in incorrect implementations that under- or over-taint values. Moreover, maintaining function models requires significant engineering effort [12, 22]. The maintenance of function models may suffice for a relatively small and static set of external abstractions, such as the JavaScript builtins and host extensions. However, it does not scale for modern multi-language web applications with Wasm modules as extensions that evolve continuously.

This paper presents JASMaint, the first taint analysis for multi-language web applications that can precisely track taint flows across language boundaries. Our approach relies on source code instrumentation to ensure portability across various execution environments that utilise JavaScript and WebAssembly, such as web browsers and server-side JavaScript runtimes (e.g., Node.js). The core of our approach resides in the *analysis orchestrator*, which carefully handles taint propagation across the interoperability boundaries between JavaScript and Wasm. We evaluate the analysis precision and performance overhead of JASMaint using a benchmark suite of multi-language programs. Our experiments show that JASMaint does increase the analysis precision w.r.t the state-of-the-art approaches, based on function models, for our set of benchmark programs. Concretely, the precision improvements range from 0.003% to 56.20% depending on the program. However, this introduces an execution overhead compared to single-language taint analyses, with slowdown factors between 1.14x and 1.61x.

The contributions of this paper are:

- We design and implement a novel taint analysis for WebAssembly. Benefiting from source code instrumentation, this analysis can be deployed to all environments that support executing WebAssembly.
- We design a portable multi-language taint analysis for web applications. This analysis relies on our novel analysis orchestrator, which supports precise taint communication across language boundaries.
- We present JASMaint, an implementation of a dynamic taint analysis framework for multi-language web applications. JASMaint integrates the developed taint analysis for WebAssembly, a portable taint analysis for JavaScript and the orchestration infrastructure to track taint communication between JavaScript and Wasm.

2 Background and Motivation

Dynamic taint analysis (DTA) is a runtime analysis technique that tracks the flow of sensitive or potentially malicious data (i.e. tainted data) throughout the execution of a program [3, 21]. The tainting policy specifies how taint gets introduced, checked and removed in a target program [21].

2.1 Dynamic Taint Analysis for JavaScript

Much work has focused on dynamic taint analysis targeted at JavaScript (cf. for a survey [3]). In what follows, we describe a dynamic taint analysis for JavaScript programs based on source code instrumentation using *shadow execution*, a technique that augments the normal execution of a program with an additional "shadow" state alongside the program's actual state [3]. Shadow execution can be implemented through *shadow values* (which augment the program's data with the shadow state) or *shadow runtime* mechanisms (which build parallel, mirrored representations of the program's runtime structures). We use shadow execution through shadow values, since a shadow runtime is more complex due to the intricacies of the JavaScript specification.

2.1.1 Taint Analysis API. Our taint analysis provides an API that enables the specification of taint sinks, sources, assertions, and checks within the target program. The `Taint` class exposes the following API to target programs:

- **Source Marking:** `Taint.source(x)` taints the value of `x`. This value `x` is returned.
- **Sink Marking:** `Taint.sink(x)` aborts the program when the value of `x` is tainted or else, it is returned.

- **Taint Sanitization Marking:** `Taint.sanitize(x)` removes taint from `x`. The value is returned.
- **Taint Assertions:** `Taint.assertIsTainted(x)` and `Taint.assertIsNotTainted(x)` check whether `x` is tainted or not, respectively. An error is thrown if the assertion fails.
- **Taint Check:** `Taint.checkIsTainted(x)` returns a boolean indicating whether `x` is tainted or not.

2.1.2 Taint Propagation.

Our taint analysis implementation relies on source code instrumentation to weave the analysis within the target program. All values in the instrumented program are wrapped by a membrane that allows the maintenance of taint for all objects and primitive values [5]. A membrane allows to wrap entire object graphs by transtitivelly wrapping values observed by the any object on such a graph [24].

To account for the taint analysis semantics, we provided the taint with *propagation rules* of JavaScript operations. Propagation rules operate on wrapped values that carry the taint label of the shadowed program value. Our approach distinguishes between intrinsic and non-intrinsic operations. *Intrinsic operations* are language operations for which the interpreter implements their semantics, or their source code is not available during instrumentation. For example, the JavaScript operators such as `-` or `++` or built-in functions such as `Math.pow` are intrinsic operations, as the source code implementing their semantics is implemented within the JavaScript engine, and thus unreachable to instrumentation. For intrinsic operations, the analysis must provide operation-specific taint propagation rules to precisely attach the taint label to the result of the operation. *Non-intrinsic* operations are functions defined in the target program, for which their source code is available. For non-intrinsic operations, the analysis instruments each expression within the function body to precisely track the propagation of taint.

An example of a propagation rule for the intrinsic function `String.prototype.at` is shown by Listing 1. At runtime, the propagator function will be called with the callee, the `this` object, the list of arguments and the result of the function call. Any primitives in this object or in the list of arguments are wrapped and labelled by the taint analysis, allowing the propagator body to implement the adequate propagation rule given these intrinsic semantics. In this example, the `taintFromArgumentsAndThis` taints the result if either one of the arguments or the `this` value is tainted, which is the most commonly used propagation rule.

2.1.3 Control Flow Taint Tracking.

Our analysis also tracks implicit control flows. When the result of a conditional is tainted and results in a branching operation, all data handled in the subsequent control flow path must also be tainted. We use a conditional taint stack to track (potentially nested) control flow conditional taints. When a segment block is entered conditionally, the taint of the conditional is pushed onto the stack, since the subsequent program execution is now influenced by the corresponding conditional value. Consequently, when the program leaves a segment block, the conditional taint is popped from the stack. Any operation covered by the analysis must check whether the taint stack has a tainted entry. If this is the case, the result of the operation must be tainted due to a tainted conditional influence.

```
1 export const string_apply_behavior_mapping = {
2   ...
3   "global.String.prototype.at": (callee, that, args, result) {
4       result.__taint = taintFromArgumentsAndThis(args, that);
5       return result;
6   }
7   ...
8 }
```

Listing 1. Example of a custom propagator for the intrinsic operation `String.prototype.at`.

```
1 // Replace where `pattern` occurs with `replacement` in `text`
2 async function replaceAsync(text, pattern, replacement) {
3   // Instantiate, then extract the functions & linear memory
4   const {instance:{exports:{replace_in_wasm,alloc_u8,memory}}}
5   = await WebAssembly.instantiate(readFileSync("replace.wasm"));
6   // Functions to en/decode strings to & from linear memory
7   const encode = (str) => {
8     const bytes = new TextEncoder().encode(str);
9     const ptr = alloc_u8(bytes.length);
10    new Uint8Array(memory.buffer,ptr,bytes.length).set(bytes);
11    return ptr; };
12  const decode = (ptr, len) => {
13    const view = new Uint8Array(memory.buffer, ptr, len);
14    return new TextDecoder().decode(view); };
15  // Encode all strings, retrieving a pointer & length
16  const [[nPtr, nLen], [vPtr, vLen], [hPtr, hLen]]
17  = [pattern, replacement,
18     text].map(s =>[encode(s), s.length]);
19  // Perform replace in wasm & decode rewritten string
20  replace_in_wasm(nPtr, nLen, vPtr, vLen, hPtr, hLen);
21    return decode(hPtr, hLen);}
22
23 (async () => {await replaceAsync("The user input is safe.",
24   "safe", "probably a ticking time bomb")})();
```

Listing 2. A JavaScript function that replaces occurrences of `pattern` with `replacement` in a given `text`. The function calls into WebAssembly to perform the replacement.

2.2 Motivation

Like current state-of-the-art analyses for JavaScript [1, 3, 14], the precision of the dynamic taint analysis presented in the previous section is constrained to the JavaScript boundaries. The analysis applies conservative tainting when taint propagates outside the JavaScript boundaries, e.g. to Wasm, and thus loses precision. To demonstrate the challenges in developing a precise taint analysis in multi-language web applications, consider the multi-language program in Listing 2,

where JavaScript delegates string replacement operations to a Wasm module. The `replaceAsync` function replaces the occurrences of the `pattern` parameter by `replacement` in the given `text`. To speed up the operation, the function uses a Wasm module to perform the actual replacement. First, the Wasm module is instantiated (lines 4–5), and the `replace_in_wasm` and `alloc_u8` functions, and the module's linear memory are imported into the JavaScript function. Then, the function declares the `decode` and `encode` functions. The function `encode` is used in line 18 to encode and write three parameters into the linear memory, and return the pointer and the length of each written parameter. At line 20, the function calls the `replace_in_wasm` function using the pointer and length of each parameter. Finally, the function calls `decode` at line 21 to read from the linear memory and decode the modified `text` string.

This example exposes key limitations of using a taint analysis designed for JavaScript in the context of multi-language applications. Without access to a detailed specification or source code of the Wasm functions being used in the program, the taint analysis cannot precisely track taint information. Even when source code is available, maintaining hand-crafted models is error-prone and hard to maintain as the codebase evolves. Furthermore, implementing a precise model may entail re-implementing complex logic to compute the taint information. In our example, a precise model for `replace_in_wasm` must perform the same logic to determine whether the `pattern` is present in the given `text` string to determine the taint after the function returns.

Sharing linear memory between JavaScript and Wasm complicates further taint analysis as a Wasm memory is a byte-addressable array that both languages can access and mutate. To track taint precisely, each byte in the linear memory must carry its corresponding taint label. Read and write operations to the linear memory, in both JavaScript and Wasm environments, must handle this taint information.

Even assuming the existence of a precise tracking mechanism within the JavaScript and Wasm environments, taint exchange across language boundaries remains an issue. For example, `encode` must get the taint information of each byte from the JavaScript environment and pass it along to the Wasm environment. The `decode` function must retrieve taint labels alongside the bytes being read and pass them to the JavaScript environment. Finally, this problem becomes more complex when JavaScript functions and objects are exported to the Wasm module.

The aforementioned challenges demonstrate the need for precise, multi-language taint analysis for modern web applications. In this work, we introduce the first multi-language taint analysis approach departing from two dynamic analyses, one for JavaScript and one for WebAssembly, and devising a taint communication mechanism for operations that cross language boundaries. Devising such a multi-language taint communication mechanism requires careful consideration of the runtime differences between JavaScript and WebAssembly. We employ the dynamic taint analysis from 2.1 and develop a novel Wasm taint analysis, as current state-of-the-art approaches [8, 17, 23] fail to be portable.

3 Taint Analysis for WebAssembly

We now discuss the design of our portable dynamic taint analysis for Wasm programs. The implementation is based on source code instrumentation, allowing the taint analysis to be deployed alongside the target module in various WebAssembly (Wasm) execution environments. As the taint analysis is implemented at the level of the Wasm semantics, the target module can be the compilation output from any higher-level language that compiles to Wasm.

The analysis is implemented using shadow execution using a shadow runtime that mirrors the Wasm execution environment state through a shadow execution stack and an abstract store. The shadow execution stack contains record values and control constructs. This stack is manipulated by instructions such as "i32.add" which pops two operands of type `i32` from the stack and pushes the sum back onto the stack. The abstract store maintains global state during execution, such as the module's linear memory or values pointed to by the global variables. The shadow runtime mirrors a structurally equivalent representation of these data structures, substituting all effective Wasm values with a label corresponding to the taint of each value. In addition to the explicit execution state, the dynamic analysis maintains an implicit control flow stack that is dictated by control instructions that conditionally execute based on tainted or untainted conditions (described in Section 3.2).

Generally, our taint analysis is designed for byte-level taint precision. However, some parts of the analysis currently lack propagation rules that can account for this byte-level precision in the `ShadowValueWithTaintStatus` values. In such cases, the byte-level taint precision is weakened to value-level taint precision, marking either all or none of the `ShadowValueWithTaintStatus` taint bytes. The implementation propagates taint at the byte level for all considered programs that are used within the evaluation (cf. Section 6).

3.1 Taint Propagation Rules

WebAssembly instructions can be categorised into four types, each serving a specific purpose in the execution of Wasm code. The following describes the semantics of taint analysis for each of these instruction categories.

3.1.1 Numeric Instructions.

When a numeric instruction is encountered during the execution of an instrumented Wasm module, the shadow execution simulates the instruction's semantics on the shadow stack while propagating the taint of the arguments.

Numeric constant instructions, such as `i32.const` or `f64.const`, introduce a new value onto the execution stack.

In the shadow execution, a corresponding ShadowValueWithTaintStatus is pushed onto the shadow stack. Constants are considered untainted, *unless* they are being introduced under the indirect influence of a tainted control flow condition (cf. Section 3.2).

Unary numeric instructions, such as `i32.clz` or `f64.neg` consume one operand from the execution stack and produce one result. For such unary operations the existing taint label is preserved.

Similarly, *binary numeric instructions*, such as `i32.add` or `f64.mul` consume two operands from the execution stack and produce one result. The taint analysis pops two ShadowValueWithTaintStatuss from the shadow stack, performs the binary operation on the Wasm values, and pushes a new ShadowValueWithTaintStatus onto the shadow stack. The result is tainted if *either* of the operands is tainted.

3.1.2 Function Calls.

The analysis tracks taint labels over both calls to host functions (i.e., imported functions) and internal Wasm function invocations. It precisely follows the Wasm call and return semantics, allowing for a seamless integration of the taint analysis. Similar to our taint analysis for JavaScript, our Wasm taint analysis distinguishes between two types of function calls, which are discussed below.

Calls to Internal Functions For internal function applications, the analysis tracks taint labels both before and after the function call. Prior to the function invocation, the advice retrieves the ShadowValueWithTaintStatus values corresponding to the function arguments from the shadow stack. A new activation frame, which represents the local context of the callee, is then pushed onto the shadow stack. This frame includes the transferred ShadowValueWithTaintStatus values, effectively propagating them to the shadow execution context of the called function.
The actual execution of the Wasm function proceeds under the control of the Wasm runtime. The instrumented code within the function body operates on the ShadowValueWithTaintStatus values on the shadow stack, passing through other parts of the taint analysis logic. The execution of a function can end in two ways: either by reaching the end of the function body or by means of an early return statement. In both cases, the ShadowValueWithTaintStatus values being returned from the function are left on the shadow stack, containing the propagated taint information of the corresponding return values. These ShadowValueWithTaintStatus values must be briefly removed from the shadow stack in order to restore the caller's context by removing the callee's activation frame, before pushing the values onto the shadow stack again as return values to the caller. This ensures that the taint label of the return values is correctly transferred back to the calling context, allowing the taint to propagate further through the remainder of the program execution. Note that this interaction across activation frames corresponds to the Wasm's specification.

Calls to Imported Functions Calls from Wasm to imported functions (i.e., JavaScript host functions) may be present in the Wasm code. Since the function body of the invoked function is not present, and therefore, not instrumented, the taint analysis can not handle the taint propagation of the function call. We will present in Section 4 our solution to handling such a precision issue. Alternatively, the implementer of a taint analysis that does not support communication with a taint analysis implementation of the host should conservatively taint all the values returned from the function call (e.g., using function models [12]).

3.1.3 Memory Operations.

The taint analysis implementation uses a shadow linear memory that mirrors the addressable memory of the Wasm instance, with a taint label associated with the corresponding byte value at the same index in the Wasm memory. This shadow memory allows for tracing all Wasm load or store instructions. Upon execution of such an instruction, the analysis is informed about the base memory address, the offset, and the specific memory operation. When loading a value from memory, the specific load operation (e.g., I32Load8U, I64Load32U) determines the number of bytes being loaded. The function then iterates over these bytes in the shadow memory and returns whether any of the addressed bytes is tainted. Only when all of the loaded shadow bytes are untainted, the result of the memory load is untainted. When storing a value into memory, the specific store operation (e.g., I64Store, I64Store8) determines the number of bytes being stored. Conversely, the analysis propagates the taint of the stored value to all corresponding bytes in the shadow memory.

3.1.4 Global Variables.

Tracking taint labels for global variables makes use of a shadow global store. Each element in this store represents a shadow representation of a global variable of the Wasm module. The GlobalHandle struct holds the taint label of this global variable as a boolean.

When a `global.get` instruction is executed, the corresponding GlobalHandle is retrieved from the shadow global store based on the global index represented as GlobalAddress. The global value and its retrieved taint label are then pushed onto the stack as a ShadowValueWithTaintStatus. When executing a `global.set` instruction, the analysis similarly retrieves the GlobalHandle from the global store using the GlobalAddress. After obtaining the GlobalHandle, its current taint label is replaced with the taint label of the value being set (i.e., the taint label of the value that was consumed from the stack).

3.2 Control Flow Taint

Our taint analysis further supports implicit taint flows through control flow instructions, including `block`, `if`, `br`, `br_if`, `br_table`, `loop`, and `return`.

Besides a shadow stack that mirrors the Wasm execution stack, a conditional taint stack is maintained to track the taint of control flow conditions. Each entry represents the taint of the condition at a specific control flow depth. Upon execution of the instruction `if-then-else`, the taint of the condition variable is pushed onto the conditional taint stack. Once either conditional branch is completely executed, the conditional taint is popped from the stack again. In addition to this conditional taint stack, a conditional block branching taint flag is maintained. This is required as some instructions conditionally branch out of one or more blocks. If this branch is the consequence of a tainted condition, the conditional block branching taint flag should be set, regardless of whether the branching was performed or not, as subsequent control flow is affected by the tainted condition.

Nevertheless, when a function call returns control flow to the caller, it is crucial to clean up the conditional taint labels that have accumulated on the taint stack during the function execution. This prevents conditional taints from incorrectly affecting the program execution beyond the function's scope. Therefore, within the instrumentation of a function application, all conditional taint labels associated with the current function invocation are popped from the stack.

At any point in the program, the current operation is considered tainted by control flow when either the conditional taint stack contains a tainted label or when the conditional block branching taint flag is set. If this is the case, the result of the operation should be tainted. For example, the analysis intercepting a `i32.const` instruction pushes a corresponding `ShadowValueWithTaintStatus` onto the shadow stack with taint being `false` unless tainted by control flow.

3.3 Taint Analysis API Design

Our analysis API is designed such that marking sources and sinks happens at the higher abstraction level (e.g., Rust, C or C++) from the application code. This design benefits users of the analysis as they can use the API in their language of choice, rather than being forced to modify the Wasm module post-compilation by hand. This choice requires the host to expose functions that accept a value from the application code, call the analysis function to taint this value and return the value back to the application. Subsequently, the taint propagation will compute where taint flows throughout the application code. Section 4.2.2 further describes an example interaction of communicating taint from WebAssembly to the host.

4 JASMaint: a Multi-language Taint Analysis

This section describes our approach for enhancing the precision of taint analyses for programs that involve JavaScript and WebAssembly interoperability. We call *interlanguage taint communication* to describe the concept of taint propagation across the interoperability boundaries between these languages. This communication includes interlanguage function calls, shared linear memory, shared global variables and shared tables. Our approach considers an interlanguage taint communication *orchestrator* which is responsible for communicating the taint information of values flowing across the language boundaries. We extended the functionality of the JavaScript and Wasm taint analyses to enable a multi-language taint analysis, which we refer to as JASMaint. The orchestrator is hosted by the JavaScript taint analysis and implements a protocol to communicate the taint for the identified interlanguage operations whenever a Wasm module is interacted with by the JavaScript application.

4.1 Orchestrator Setup

When the JavaScript program instantiates a Wasm module, our orchestrator takes the import object and the resulting instantiated module object, and attaches metadata and functionality necessary for the interlanguage taint tracking. Concretely, this involves adding flags and taint communication hooks to the functions, memories and globals imported into the Wasm binary, for later use by the orchestrator throughout the analysis. Additionally, the instantiation object gets metadata attached, allowing the orchestrator to later attach the flags and taint communication hooks to the exported functions, memories and globals retrieved from the instantiation. Besides the target program exports, the instantiated Wasm module exports a set of analysis-specific functions that enable the orchestrator to manage the taint information of the module. The rest of the section explains those Wasm exports used by the orchestrator to communicate and retrieve taint from interactions with the Wasm module.

4.2 Interlanguage Function Calls

For function calls, JavaScript can invoke a call to WebAssembly functions and WebAssembly can invoke a call to JavaScript functions. We describe each function call direction individually.

4.2.1 JavaScript to Wasm Function Calls.

Taint propagation through function calls between JavaScript and Wasm requires the propagation of taints through both the arguments and return values of the call. The first two columns of Figure 1 show a high-level architecture of taint flow through interlanguage function calls. The interlanguage taint propagation for calls from JavaScript to Wasm is shown in the first column, and from Wasm to JavaScript in the second column.

Table 1. The WebAssembly functions for interlanguage function call taint communication.

Exported Wasm Function	Signature	Description
Calls from JavaScript to Wasm		
`prepare_for_argument_taints`	*i32* → ()	Initialises the argument taints buffer based on the number of arguments.
`set_argument_taint`	*i32*, *i32* → ()	Sets the taint of the argument at the specified index.
`get_result_count`	() → *i32*	Returns the number of values returned by the function.
`get_result_taint`	*i32* → *i32*	Returns the taint of the return value at the specified index.
Calls from WebAssembly to JavaScript		
`get_argument_count`	() → *i32*	Returns the number of arguments passed to the function.
`get_argument_taint`	*i32* → *i32*	Returns the argument taint for the specified argument index.
`prepare_for_result_taints`	*i32* → ()	Initialises the result taints buffer based on the number of results.
`set_result_taint`	*i32*, *i32* → ()	Sets the taint of the return value at the specified index.
`is_wasm_calling_host`	() → *i32*	Returns whether the Wasm is currently calling a JavaScript function.
`all_argument_taints_received`	() → *i32*	Returns if the argument taints of the active function call have been retrieved via `get_argument_taint()`.
Shared Linear Memory		
`get_memory_taint`	*i32* → *i32*	Retrieves the taint of the value at a given memory index.
`set_memory_taint`	*i32*, *i32* → ()	Mutates the taint of the value at a given memory index.
Shared Global Variables		
`get_global_taint`	*i32* → *i32*	Retrieves the taint of the global variable with the specified index.
`set_global_taint`	*i32*, *i32* → ()	Mutates the taint of the global variable with the specified index.

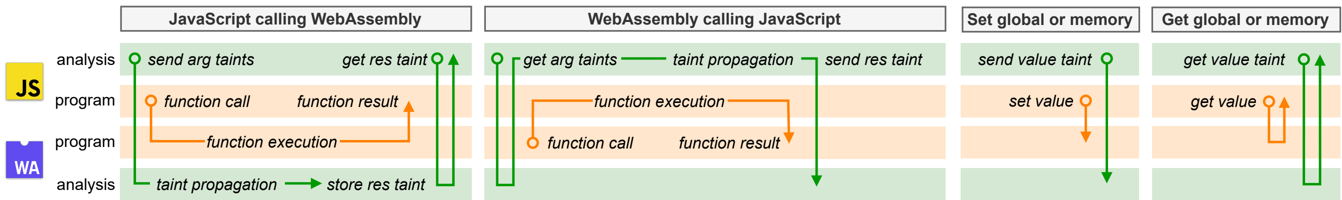

Figure 1. The architecture of the taint communication across the interlanguage structures.

When JavaScript invokes a Wasm function, the taint values of the JavaScript arguments are communicated to the Wasm analysis. To achieve this, the Wasm taint analysis exposes four taint communication functions, as specified in Table 1. These functions are exported in the Wasm binary for use by the orchestrator.

Functions `prepare_for_argument_taints` and `set_argument_taint` are invoked by the orchestrator analysis before JavaScript makes the Wasm function call. Specifically, they write the taint value of each call argument into the argument taints buffer. This allows the Wasm analysis to retrieve the argument taints from this buffer for taint propagation during interlanguage calls.

After the function call, the taint of the return values must be retrieved to JavaScript. The orchestrator analysis retrieves the taint values of the results by invoking `get_result_count` and `get_result_taint`. These functions operate on the result taint buffer, again implemented and managed within the Wasm analysis.

4.2.2 Wasm to JavaScript Function Calls.

When a Wasm module invokes a JavaScript function, the taint value of the arguments must be retrieved by the JavaScript taint analysis and the taint of the result must be communicated back to the Wasm analysis. This interaction is shown in the second column in Figure 1 and handled through six taint communication functions exposed by the Wasm binary, as defined in Table 1. The `is_wasm_calling_host` and `all_argument_taints_received` allow the orchestrator to detect a function invocation from Wasm to JavaScript. It then uses `get_argument_count` and `get_argument_taint` functions to retrieve the taints of the arguments from the argument taints buffer. Then, the orchestrator assigns the taint to the argument wrappers in the JavaScript analysis, and the function's body is executed with precise taint information. After the JavaScript function returns, the `prepare_for_result_taints` and `set_result_taint` functions are called to communicate the taints of the return values back to Wasm, where they are stored in the result taints buffer. The Wasm analysis can then proceed with the rest of the execution.

4.3 Shared Linear Memory

A JavaScript program can either define a `Wasm.Memory` which it then shares with Wasm on instantiation (i.e., imported memory), or retrieves exported memory from Wasm after instantiation (i.e., exported memory). From the analysis perspective, the shared memory is managed by the Wasm analysis; therefore, the orchestrator must communicate with the Wasm analysis whenever any byte of this memory is accessed or mutated. The third and fourth columns of Figure 1 show the architecture for interlanguage linear memory interaction. The third column in this figure shows the interaction between analyses when the JavaScript program stores information in the shared linear memory. The fourth column in this figure shows how the analysis retrieves taint information when the JavaScript program reads from the shared memory.

The functions `set_memory_taint` and `get_memory_taint` in Table 1 are exposed by the Wasm analysis to handle the taint information of shared memory accesses from the JavaScript program. All shared memory instances are extended with the `wasm_memory` flag and a reference to these exposed functions. This enables the analysis to identify shared memory instances throughout the program and access their required taint communication functionality.

4.4 Shared Global Variables

A JavaScript program can either define a `Wasm.Global` object which it then shares to Wasm on instantiation (i.e., imported globals), or retrieves exported globals from Wasm after instantiation (i.e., exported globals). The architecture for managing taint involving operation on globals is shown in the two rightmost columns of Figure 1, implementing an approach similar to the shared memory described in Section 4.3.

To account for precise interlanguage taint communication of operations on global variables, the analysis exposes two taint communication functions to be invoked by the orchestrator, as specified in Table 1. Specifically, `set_global_taint` and `get_global_taint` are attached to each global such that the orchestrator can set and get the taint of the shared global when used within the context of an interlanguage operation. All global instances are extended with a reference to these exposed functions. Communicating about the taint label of a global variable using these attached functions requires knowledge of the global's index in the Wasm module. However, JavaScript does not provide a mechanism to retrieve this index for a global variable. To address this issue, we analyse the Wasm module and create a map of names and their corresponding indices of imported and exported entities (e.g., memories, functions, tables and globals). The mapping of global variable names to their index is whenever a Wasm instantiation occurs. In this phase, the computed mapping is attached as `global_index_mapping` to the result of the module instantiation. This allows every taint communication function to look up the index of the global variable it is associated with. The mutation of a global variable is intercepted by the analysis orchestrator, which uses the `set_global_taint` function of the respective global with the taint of the value being set. This ensures the taint is communicated to the Wasm analysis. The retrieval of the value of a global variable is performed by accessing the `value` property on the `Wasm.Global` object, and is therefore intercepted in the analysis orchestrator. The orchestrator then requests the taint label for the given global by calling the attached `get_global_taint` method.

5 Implementation

We now present the relevant details of the JASMaint multi-language taint analysis implementation, which combines two single-language source code instrumentation-based analyses. The JavaScript taint analysis is implemented using the Aran v5.1.1 source code instrumentation framework, together with Linvail v7.6.6 access control membrane framework for tracking primitives and objects across the execution. The analysis consists of 1487 lines of code and implements custom propagators for Aran intrinsic [4] operations and some of the ECMAScript built-ins, including Array, DataView, Math, Number, Object, Reflect, and String. Propagators cover only commonly used intrinsics (e.g., `String.toLowerCase`), including all used by the benchmark suite (cf. Section 6). Missing propagators can be easily added using the existing definition structure (cf. Section 2.1.2). The Aran taint analysis currently only supports taint tracking with a single Wasm instance and allows only one interlanguage operation at a time, per direction.

The Wasm analysis is implemented in Wastrumentation [19] and comprises 1227 lines of Rust code. It is built on top of an existing shadow execution framework present within Wastrumentation. Our current analysis lacks support for Wasm tables because Wastrumentation does not yet support the instrumentation of tables. This prevents taint propagation through tables, but this was not a problem in practice in the benchmarks used for evaluation, as they do not appear to extensively use tables.

6 Evaluation

This section describes the evaluation of the proposed dynamic taint analysis implementations. We assess their effectiveness across various benchmarks with a focus on the performance overhead and precision improvement.

The evaluation addresses the following research questions.

- **RQ1**: What is the runtime performance overhead for source-code-based taint analysis for JavaScript?
- **RQ2**: What is the runtime performance overhead for source-code-based taint analysis for Wasm?

- **RQ3**: What is the runtime performance overhead for source-code-based taint analysis for multi-language applications?
- **RQ4**: How does the taint tracking precision improve for the JASMaint analysis?

6.1 Evaluation Methodology

We evaluated all three developed taint analyses, including the JavaScript and Wasm taint analysis implementations, and the JASMaint extension. To this end, we employed a benchmarking suite with three variants per benchmarking program: a JavaScript version, a Rust version and a multi-language version. This benchmark suite features programs derived from the Computer Language Benchmarks Game (CLGB), a collection of algorithms used for comparing language implementations [9]. We partially leveraged an existing CLBG implementation, specifically adapting those found in Kreindl's 'taint-benchmarks' suite [16]. This benchmarking suite was designed to evaluate TruffleTaint [15], a platform for multi-language dynamic taint analysis for the GraalVM. It implemented CLBG benchmarks for JavaScript, C, and a combined multi-language approach. While these provided a strong foundation for our benchmark suite, their specific interoperation characteristics and taint methodology differed from ours. For instance, these benchmarking programs use a TruffleTaint API for defining taint sinks and sources in the programs, which are specific to the Truffle framework within the GraalVM ecosystem. Additionally, these programs did not include the capabilities to establish interoperability with Wasm. Finally, the suggested input parameters made some experiments infeasible due to the high slowdown rates incurred by the taint analysis. For these reasons, the obtained results may not be compared with other experiments employing this benchmark suite. We provide an overview of the adapted and developed benchmark programs for the JavaScript, Wasm, and combined multi-language JASMaint taint analysis[1].

For the evaluation of the JavaScript-only taint analysis, all the benchmarks were adapted from the aforementioned 'taint-benchmarks' repository. The modifications involved replacing the TruffleTaint API integration with an integration of our taint API for JavaScript (cf. Section 2.1.1) to specify taint sources, sinks, assertions, and checks in the target program. Furthermore, the taint assertions in the existing benchmarking programs did not account for implicit taint flows. Therefore, some assertions had to be inverted, replaced, or removed.

The Wasm-only taint analysis (cf. Section 3) was evaluated using a set of benchmark programs compiled from Rust. The existing Kreindl's 'taint-benchmarks' suite contained the equivalent C benchmark programs, but they were not used due to issues with the Emscripten compiler for the integration with our Wasm taint analysis. Therefore, a selection of benchmarks was adapted from the official Rust suite of The Computer Language Benchmarks Game (CLBG)[2]. Other benchmarks were reimplemented in Rust to ensure structural resemblance to their equivalent JavaScript counterpart. Structural resemblance was preferred since this allows for a more straightforward construction of the multi-language program. Furthermore, all Rust benchmark programs include a foreign function interface that provides the capability to mark sinks, sources, assertions, and checks in the Rust source code. These functions must be included in the import object during instantiation in JavaScript.

The multi-language JASMaint taint analysis (cf. Section 4) evaluation involved implementing custom multi-language benchmark programs that combine the implemented JavaScript and Rust programs. The chosen interlanguage flow structure on each multi-language program was inspired by, but does not strictly follow, the aforementioned 'taint-benchmarks' from TruffleTaint [15]. The custom benchmark suite consists of the binary-trees, fannkuch-redux, fasta, mandelbrot, n-body, pi-digits, spectral-norm, regex-redux, k-nucleotide and reverse-complement programs.

6.1.1 Experimental Setup.

The experiments were conducted on a 2023 MacBook Pro with these specifications:

- **Processor:** Apple M2 Max System on Chip (SoC), featuring a 12-core CPU (8 performance cores and 4 efficiency cores). Externally performed benchmarks report clock speeds up to 3.7 GHz for the performance cores and up to 3.4 GHz for the efficiency cores [13].
- **Memory:** 32 GB of unified RAM.
- **Storage:** 1 TB Solid State Drive (SSD).
- **Graphics:** Integrated 38-core GPU.
- **Operating System:** macOS version 15.4.1.
- **Node.js Version:** v22.12.0

6.2 Performance Evaluation

The performance of the three dynamic taint analysis implementations was evaluated by measuring the runtime overhead introduced across their respective benchmark suite. For every benchmark program variant, four configurations were measured individually. Every experiment consisted of 18 measurement iterations and 2 warmup iterations. The first experiment of every benchmark program consists of the execution of the program without any analysis, which we consider as the baseline performance measurement. The second configuration consists of the *Forward* analysis which has an empty advice logic. The performance measurement of this configuration serves as the baseline overhead of the instrumentation framework. The third configuration consists of the *Shadow* (for Wasm) or *Membrane* (for JavaScript)

[1] We provide an open-source the implementation of our benchmarks here: https://gitlab.soft.vub.ac.be/disco/jasmaint-js-wasm-taint-benchmarks.

[2] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

execution analysis performance. This configuration was measured for the JavaScript and the Rust variants only, without interoperability.

For the interoperating variants, the third configuration measured the performance of a state-of-the-art (SOTA) source-code instrumentation-based taint analysis. Concretely, the SOTA analyses used for precision evaluation are the Aran and Wastrumentation single-language analyses variants described in Sections 2.1 and 3, respectively. For the multi-language benchmarks in Figure 3, we implemented an over-approximating taint analysis, labelled "Taint SOTA". This implementation does not include the integrated multi-language taint communication capabilities and conservatively overtaint values crossing the language boundaries.

The last configuration measured the performance of the implemented taint analysis, being the JASMaint analysis for the interoperating variants and the respective constituent analyses for the JavaScript- and Rust- only variants.

Each of these configurations contributes incrementally to the overhead of the implemented taint analyses. This allows a better understanding of the overhead measurement and may allow future research to steer efforts in reducing this overhead.

6.2.1 Discussion. Figure 2 displays the box plots showing the performance overhead for the separate taint analysis implementations per benchmark program. Figure 3 shows the performance overhead boxplots for the interoperating programs. Within each graph, each box plot represents the slowdown of a specific analysis relative to its respective baseline.

Visually comparing the box plots across graphs would yield a biased, skewed, and incorrect conclusion for two reasons. First, the x- and y-axis labels differ between graphs. Second, the baseline used to compute slowdowns varies: for example, the JavaScript implementation (top row of Figure 2) and the WebAssembly implementation (bottom row of Figure 2) each use different baselines, which also differ from the multi-language implementation (Figure 3).

Across all benchmarks and analysis types, we observe a trend of increased slowdown as the complexity of the analysis increases, which is expected. The measured slowdowns, however, vary significantly between the different benchmark programs.

- **RQ1** For the JavaScript-only benchmarks, the *Forward* analysis shows the lowest slowdown. The *Membrane* configuration, which represents the overhead of the wrapping and unwrapping behaviour established by this analysis, considerably increases the overhead with respect to the *Forward*. Finally, the taint analysis consistently exhibits the highest slowdown. Although this trend is noticeable across all benchmark programs, the actual slowdown factors are heavily dependent on the particular program being benchmarked, ranging from an average taint analysis slowdown of 1.27x for the pi-digits to 12670x for the mandelbrot program.
- **RQ2** Similarly, the Wasm-only benchmarks demonstrate an increasing slowdown for the more computationally complex analyses. The *Forward* analysis also shows a relatively low overhead. The *Shadow* analysis, which maintains a shadow runtime, introduces a more noticeable slowdown, consistently falling between the slowdowns of the *Forward* configuration and *Taint* analysis. The taint analysis consistently yields the highest performance overhead. Again, the slowdown ratios substantially differ between the various benchmark programs, now ranging from 249.21x for the mandelbrot to 1203.38x for the k-nucleotide program when performing the taint analysis.
- **RQ3** The multi-language experiments, representing multi-language interoperation, again show a significant overhead introduced by the developed JASMaint analysis (cf. Figure 3). Consistent with the JavaScript- and Wasm-only experiments, the actual slowdown factors considerably vary across benchmark programs. For the JASMaint taint analysis experiments, this factor ranges from 11.93x for the pi-digits to 2573.89x for the mandelbrot program. Additionally, every program exhibits a slowdown in JASMaint when compared with the SOTA analysis, representing the overhead of the added interlanguage taint communication capabilities. This slowdown is less impactful, ranging from 1.14x (regex-redux) to 1.61x (spectral-norm).

Despite the significant variation in slowdown factors across benchmark programs, the results indicate a trade-off between the complexity of the analysis and its performance. The taint analysis for multi-language applications incurs substantial performance overheads for JavaScript, Wasm, and multi-language programs. These results align with our expectations due to the increased complexity of the taint analysis. This highlights the considerable performance cost of running a taint analysis on a program. However, from our limited set of benchmark programs, we could not establish a correlation between the performance overhead of JASMaint and the number of crossings from one language to the other.

6.3 Precision Evaluation

Besides performance evaluation, the multi-language JASMaint taint analysis requires a precision evaluation to show its precision improvements compared to the state-of-the-art (SOTA) taint analyses. As explained before, these SOTA analyses do not cover taint propagation mechanisms for the interlanguage structures between JavaScript and Wasm.

To measure the precision change, the tainting behaviour of the JASMaint analysis was compared against this SOTA taint analysis implementation. The "precision change" can

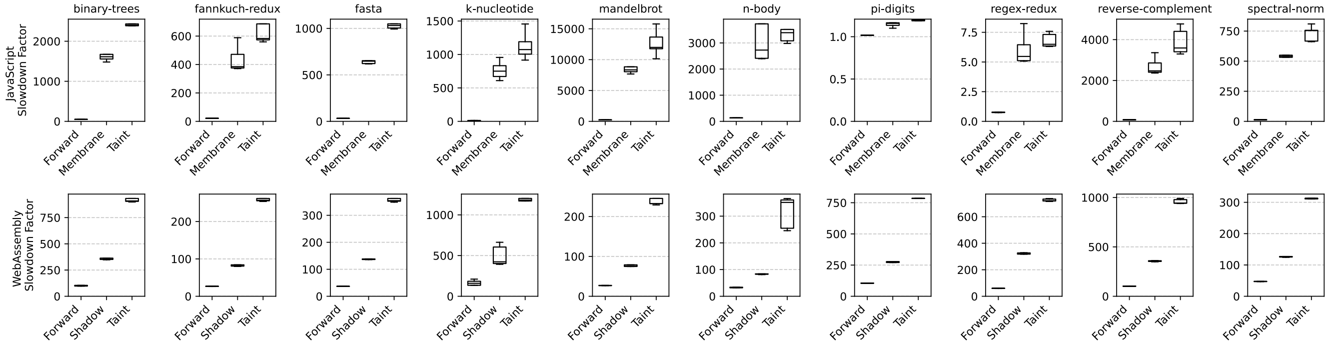

Figure 2. The runtime overhead evaluation of the benchmark suite for the non-interoperating programs.

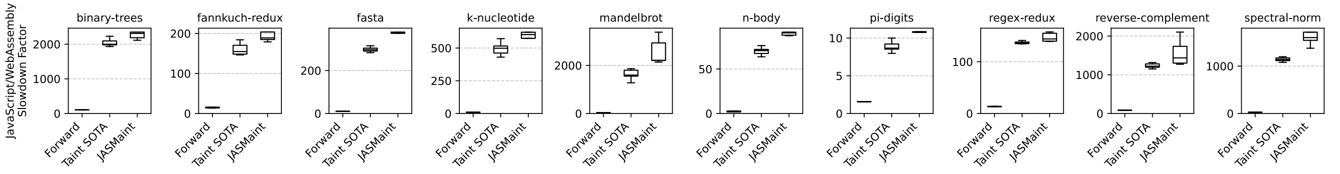

Figure 3. The runtime overhead evaluation of the benchmark suite for the interoperating programs.

be called the "precision improvement" under the (safe) assumption that the JASMaint taint analysis is more precise than the SOTA taint analysis. To implement this evaluation, all multi-language benchmark programs were run with both versions (i.e., SOTA and JASMaint) of the taint analysis. During execution, a log file containing runtime taint labels was constructed for both taint analysis versions. The logging responsibility was implemented in the Aran and Wastrumentation analyses, which record taint labels at specific points of taint propagation. This approach establishes an unbiased and consistent selection of logging points, which cannot be ensured when statically selecting and inserting logging points for every target program.

After completion of the benchmarks, every benchmark program has two associated logs: one with the recorded taints of the SOTA taint analysis and one with the recorded taints of the JASMaint taint analysis. Since both logs are constructed using the same benchmark program and the same logging points, a line-by-line comparison of both log files allows for the comparison of taint values between the two approaches at any point during the analysis. Under the assumption that the JASMaint taint analysis is precise (i.e., it does not undertaint or overtaint), the number of correctly and incorrectly labelled values by the SOTA analysis can be identified. Every entry for which the JASMaint log contains a tainted label and the SOTA log contains an untainted label is considered to be incorrectly undertainted by the SOTA, and is therefore called a *false negative* (F_0). This scenario does not appear since the SOTA analysis has been configured to conservatively overtaint values when taint propagation rules are missing. However, this overtainting results in log entries where JASMaint contains an untainted label while the SOTA contains a tainted label, called a *false positive* (F_1). When the taint labels of both analyses in an entry are identical, the taint label is assumed to be correct. In this case, we speak of a *true positive* (T_1) when both labels are tainted, or a *true negative* (T_0) when both labels are untainted.

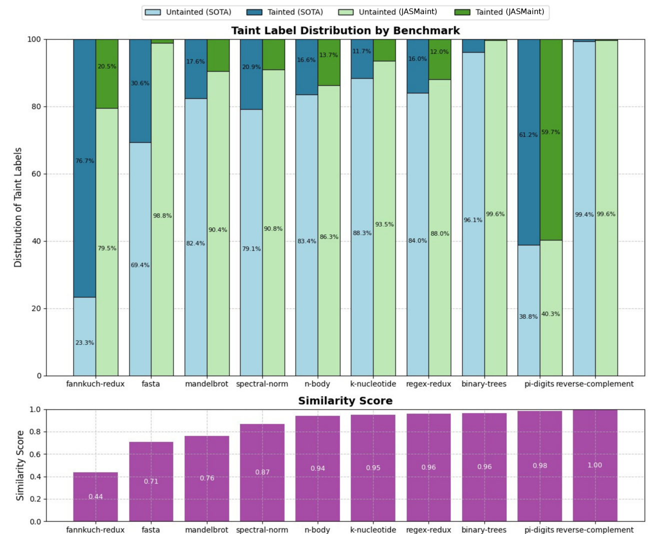

Figure 4. The distribution of taint labels for the SOTA and the proposed analysis, for every benchmark program.

The similarity score $S = \frac{T_1+T_0}{T_1+T_0+F_1+F_0}$ determines how precise the SOTA analysis is compared to the multi-language

JASMaint analysis. Therefore, $1 - S$ is considered to be the precision improvement of the JASMaint analysis relative to the SOTA analysis, when the JASMaint analysis is assumed to be precise.

6.3.1 Discussion.

The upper chart in Figure 4 shows the comparison between the distribution of the recorded labels for the SOTA and the JASMaint analysis, for every benchmark program. The number of false positives F_1 and the absence of false negatives F_0 confirm the overtainting strategy of the SOTA analysis. This explains the consistent reduction of the proportion of tainted log labels when comparing the JASMaint analysis with the SOTA analysis. These similarity scores for each program are visualised in the lower chart of Figure 4.

RQ4 For every similarity score S, the precision change can be computed as $1 - S$. The precision improvements in the used benchmarking suite range from 0.003% (for the reverse-complement benchmark) to 56.20% (for the fannkuch-redux benchmark), effectively answering RQ4.

From our limited set of benchmark programs, we could not establish a correlation between the similarity score and the total recorded operations, the proportion of operations spent in either language or number of crossings from one language to the other.

7 Related Work

Dynamic taint analysis is a vast area of research. We focus our comparison of related work on multi-language taint analysis and dynamic taint analysis for Wasm.

7.1 Multi-Language Taint Analysis

Supporting precise taint propagation on multi-language applications has been approached in three different ways.

1. Multi-language taint analysis based on taint signatures. To support taint propagation when applications interact with native extensions or host abstractions (e.g. for I/O), some approaches propose the use of approximate models or signatures of the external abstractions [6, 7, 25]. Theoretical frameworks [12, 22] have studied this problem, typically distinguishing between *shallow models*, intended for side-effect-free function calls, and *deep models*, which can track information flows even in the presence of side-effectful libraries. Such models enable the analysis to approximate taint propagation when the program code calls into an external language. In the context of JavaScript, approaches based on deep models [6, 11] can offer better precision for taint tracking in the presence of side-effectful external abstractions. Karim et al. [14] propose Ichnaea, a platform-independent taint analysis framework for JavaScript featuring hand-crafted function models to track taint information for native functions and built-ins. Such models approximate the taint propagation of the native implementation by implementing propagation rules following the ECMAScript language specification. Augur [1] builds on Ichnaea, but extends the analysis to handle asynchronous code. Overall, maintaining these models requires considerable engineering effort, as each native or built-in extension necessitates a corresponding model to be implemented. Although it may be feasible to implement models for the built-in and host abstractions, this approach does not scale well to applications where JavaScript interacts with Wasm, as the model implementor usually does not have access to the specification of the code implemented on Wasm modules.

2. Multi-language taint analysis on a polyglot virtual machine. TruffleTaint [16] implements a language-agnostic taint analysis framework that leverages the Truffle language implementation framework, enabling tracking of taint across language implementations supported by the GraalVM. The analysis can achieve full precision because it operates on a common representation, i.e. Truffle ASTs, for all languages used in the program. This approach does not require an analysis orchestrator like our approach. However, the analysis is limited to language implementations on the GraalVM. In contrast, our approach is portable and can be deployed on any platform that supports both JavaScript and Wasm, such as web browsers and server-side JavaScript runtimes (e.g., Node.js). TruffleTaint also requires runtime integration for encoding language-specific propagation semantics, e.g. for defining taint propagation for intrinsic or built-in functions.

3. Multi-language taint analysis using language-independent representations. PolyCruise [18] propose a hybrid multi-language taint analysis approach where a language-specific static analysis computes symbolic dependencies, which are then used to drive a dynamic language-agnostic data flow analysis.

7.2 Taint analysis for Wasm

Several dynamic taint analyses have been implemented for Wasm [8, 17, 23]. They can be categorised depending on the technique used to deploy the analysis: either using source code or interpreter instrumentation.

Source code instrumentation-based analysis. Wasabi [17] is a source code instrumentation-based dynamic analysis platform for Wasm. As part of its evaluation, the paper presents a first taint analysis for Wasm programs on top of it. In contrast to our analysis, this implementation fails to detect implicit flows, resulting in a loss of precision. Furthermore, Wasabi requires the Wasm program under analysis to run in the browser, limiting its portability.

Interpreter instrumentation-based analysis. Szanto et al. [23] propose a Wasm interpreter implemented on JavaScript, which they extend with support for taint analysis.

Their analysis operates at the byte level and it is able to track implicit taint flows between variables. Fu et al. [8] implement a taint analysis engine by modifying the V8 JavaScript engine. Our approach inlines the taint analysis within the Wasm module directly to ensure the portability of the analysis across all contexts where Wasm modules can be executed.

8 Conclusion

This paper presented a novel approach to dynamic taint analysis for modern web applications, addressing the lack of support for taint propagation between interoperating JavaScript and WebAssembly programs. To this end, we first developed a Wasm-only taint analysis that is portable across WebAssembly execution environments. Second, we integrate this taint analysis with a JavaScript taint analysis and orchestrator that enables precise taint communication across language boundaries, making multi-language taint analyses based on source-code instrumentation possible. Our precision experiments show a reduction in overtainting by 0.003%–56.20% of JASMaint with respect to single-language taint analyses in the context of multi-language applications. The results indicate a trade-off between increasing taint precision and the added performance overhead, with JASMaint incurring an additional overhead of 1.14x–1.61x relative to the state of the art. The developed multi-language JASMaint analysis represents a significant step forward in understanding data flows and mitigating security risks in complex, multi-language web applications.

References

[1] Mark W. Aldrich, Alexi Turcotte, Matthew Blanco, and Frank Tip. 2023. Augur: Dynamic Taint Analysis for Asynchronous JavaScript. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 153, 4 pages. doi:10.1145/3551349.3559522

[2] Amir-Mohammadian, Sepehr and Skalka, Christian. 2016. In-Depth Enforcement of Dynamic Integrity Taint Analysis. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (Vienna, Austria) *(PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 43–56. doi:10.1145/2993600.2993610

[3] Andreasen, Esben and Gong, Liang and Møller, Anders and Pradel, Michael and Selakovic, Marija and Sen, Koushik and Staicu, Cristian-Alexandru. 2018. A Survey of Dynamic Analysis and Test Generation for JavaScript. *Comput. Surveys* 50, 5 (Sept. 2018), 1–36. doi:10.1145/3106739

[4] Laurent Christophe, Coen De Roover, and Wolfgang De Meuter. 2024. Aran: JavaScript Instrumentation for Heavyweight Dynamic Analysis. (2024). Presented at the 23rd Belgium-Netherlands Software Evolution Workshop (BENEVOL'24), Namur, Belgium.

[5] Christophe, Laurent and Boix, Elisa Gonzalez and De Meuter, Wolfgang and De Roover, Coen. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 260–270. doi:10.1109/SANER.2016.91

[6] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 629–643. doi:10.1145/2810103.2813684

[7] Enck, William and Gilbert, Peter and Han, Seungyeop and Tendulkar, Vasant and Chun, Byung-Gon and Cox, Landon P. and Jung, Jaeyeon and McDaniel, Patrick and Sheth, Anmol N. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. doi:10.1145/2619091

[8] William Fu, Raymond Lin, and Daniel Inge. 2018. Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint arXiv:1802.01050* (2018).

[9] Brent Fulgham and Isaac Gouy. 2024. Measured : Which programming language is fastest? (Benchmarks Game) — benchmarksgame-team.pages.debian.net. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. [Accessed 03-04-2025].

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

[11] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing* (Gyeongju, Republic of Korea) *(SAC '14)*. Association for Computing Machinery, New York, NY, USA, 1663–1671. doi:10.1145/2554850.2554909

[12] Hedin, Daniel and Sjösten, Alexander and Piessens, Frank and Sabelfeld, Andrei. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Springer-Verlag, Berlin, Heidelberg, 49–70. doi:10.1007/978-3-662-54455-6_3

[13] Klaus Hinum. [n. d.]. Apple M2 Max Processor - Benchmarks and Specs — notebookcheck.net. https://www.notebookcheck.net/Apple-M2-Max-Processor-Benchmarks-and-Specs.682771.0.html. [Accessed 28-05-2025].

[14] Karim, Rezwana and Tip, Frank and Sochůrková, Alena and Sen, Koushik. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1364–1379. doi:10.1109/TSE.2018.2878020

[15] Jacob Kreindl, Daniele Bonetta, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2022. Polyglot, Label-Defined Dynamic Taint Analysis in TruffleTaint. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 152–153. doi:10.1145/3546918.3560807

[16] Kreindl, Jacob and Bonetta, Daniele and Stadler, Lukas and Leopoldseder, David and Mössenböck, Hanspeter. 2020. Multi-language dynamic taint analysis in a polyglot virtual machine. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes*. ACM, Virtual UK, 15–29. doi:10.1145/3426182.3426184

[17] Lehmann, Daniel and Pradel, Michael. 2019. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1045–1058. doi:10.1145/3297858.3304068

[18] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. PolyCruise: A Cross-Language Dynamic Information Flow Analysis. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2513–2530. https://www.usenix.org/conference/

usenixsecurity22/presentation/li-wen

[19] Aäron Munsters, Angel Luis Scull Pupo, and Elisa Gonzalez Boix. 2025. Wastrumentation: Portable WebAssembly Dynamic Analysis with Support for Intercession. In *39th European Conference on Object-Oriented Programming (ECOOP 2025) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 23:1–23:29. doi:10.4230/LIPIcs.ECOOP.2025.23

[20] Marc Ohm, H. Plate, Arnold Sykosch, and M. Meier. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment* 12223 (2020), 23 – 43. doi:10.1007/978-3-030-52683-2_2

[21] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy*. IEEE, Oakland, CA, USA, 317–331. doi:10.1109/SP.2010.26

[22] Sjösten, Alexander and Hedin, Daniel and Sabelfeld, Andrei. 2018. Information Flow Tracking for Side-Effectful Libraries. In *Formal Techniques for Distributed Objects, Components, and Systems*, Baier, Christel and Caires, Luís (Ed.). Springer International Publishing, Cham, 141–160.

[23] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. Taint tracking for webassembly. *arXiv preprint arXiv:1807.08349* (2018).

[24] Tom Van Cutsem and Mark S. Miller. 2013. Trustworthy proxies: virtualizing objects with invariants. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 154–178. doi:10.1007/978-3-642-39038-8_7

[25] Jian Xiang and Stephen Chong. 2021. Co-Inflow: Coarse-grained Information Flow Control for Java-like Languages. In *2021 IEEE Symposium on Security and Privacy (SP)*. 18–35. doi:10.1109/SP40001.2021.00002

Received 2025-06-24; accepted 2025-07-28