Towards Efficient, Multi-Language Dynamic Taint Analysis*

Jacob Kreindl
Johannes Kepler University Linz
Austria
jacob.kreindl@jku.at

Daniele Bonetta
Oracle Labs
USA
daniele.bonetta@oracle.com

Hanspeter Mössenböck
Johannes Kepler University Linz
Austria
hanspeter.moessenboeck@jku.at

Abstract

Dynamic taint analysis is a program analysis technique in which data is marked and its propagation is tracked while the program is executing. It is applied to solve problems in many fields, especially in software security. Current taint analysis platforms are limited to a single programming language, and therefore cannot support programs which, as is common today, are implemented in multiple programming languages. Current implementations of dynamic taint analysis also incur a significant performance overhead.

In this paper we address both these limitations (1) by presenting our vision of a multi-language dynamic taint analysis platform, which is built around a *language-agnostic core framework* that is extended by *language-specific front-ends* and (2) by discussing the use of speculative optimization and dynamic compilation to reduce the execution overhead of dynamic taint analysis applications. An implementation of such a platform would enable dynamic taint analyses that can target multiple languages in one analysis implementation and can track tainted data across language boundaries. We describe this approach in the context of the GraalVM runtime and its included JIT compiler, Graal, which allows us to target both dynamic and static languages.

CCS Concepts • **Software and its engineering** → **Interpreters**; **Software defect analysis**; • **Security and privacy** → *Software security engineering*.

Keywords Multi-Language Taint Analysis, Cross-Language Taint Analysis, Dynamic Taint Analysis, GraalVM, Sulong, LLVM, Node.js, JavaScript, Native Extensions

*This research project is partially funded by Oracle Labs.

ACM Reference Format:
Jacob Kreindl, Daniele Bonetta, and Hanspeter Mössenböck. 2019. Towards Efficient, Multi-Language Dynamic Taint Analysis. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357390.3361028

1 Introduction

Dynamic Taint Analysis [30, 39], which is often also referred to as *dynamic taint tracking*, is a program analysis technique in which *taint labels* are attached to sensitive values, marking them as *tainted*, and their propagation through the program is tracked during execution. An application of this analysis technique defines *taint sources*, i.e., locations within the program where data is marked as tainted, and it also defines *taint sinks*, i.e., locations in the program at which the analysis needs to detect the presence of tainted data and has to react to it. Dynamic taint analysis has been applied to solve problems of many fields such as software vulnerability detection [11, 29, 33], software testing [12, 14], and debugging [16]. It is commonly implemented on top of binary analysis platforms [26, 39] or within interpreters for a single dynamic programming language [30]. Such an approach, however, introduces two problems which our research addresses.

The first problem we want to address is the limited scope of taint analyses implemented on a platform that supports only a single target language. Dynamic languages such as Ruby or frameworks such as Node.js typically contain a *foreign function interface* which enables programs to invoke *native extensions*, i.e., libraries implemented in a lower-level programming language such as C [19] and compiled to native code. An analysis implemented on a language-specific platform can only target either the native extensions or the dynamic language, and cannot propagate taint across languages. Similarly, an analysis for one language would need to be re-implemented entirely on another analysis platform to support another language. To address this problem, we propose a design for a taint analysis platform that supports taint tracking within an extensible set of programming languages. This platform would be based on a common core taint framework, which aggregates language-agnostic functionality for taint propagation, and which can be reused by multiple extensions that integrate language-specific runtimes.

The second problem we want to address with our research is the often significant execution time overhead of dynamic taint analysis, which is caused by instrumenting each program instruction to perform taint propagation [15, 26]. While taint tracking platforms have been proposed that exhibit low overhead as long as the analyzed program does not actually operate on tainted data [17, 37], full taint propagation can increase execution time significantly. For example, LIFT [37] and libDFT [26], two platforms for taint analysis in x86 binaries that claim to implement taint propagation efficiently, increase execution time by up to 10x and 6x, respectively, for some applications. Performance also remains a significant challenge for taint tracking in dynamic languages [30]. To tackle this challenge, we are investigating the potential of designing dynamic taint analysis to be optimized by generic compiler optimizations and to leverage speculative execution, which a state-of-the-art JIT compiler is already capable of.

We plan to implement our ideas in *GraalVM*[1] [46], a virtual machine which can execute and instrument programs implemented in various dynamic programming languages such as JavaScript, Ruby and Python, as well as in languages such as C and C++ that are typically compiled statically. GraalVM also supports interoperability between these languages, and includes the *Graal* optimizing JIT compiler to improve execution performance.

2 Motivation

In complex software systems, tight interactions between multiple programming languages happen frequently. This includes user-provided native extensions used by dynamic language programs [19], but also *language embeddings*, that is, software systems that embed one or more language runtimes to enable advanced functionality such as scripting capabilities, stored procedures execution, etc. A notable example of a language embedding is Node.js [4], a popular web programming framework for server-side applications development in JavaScript. In Node.js, much of the functionality exposed as JavaScript APIs is actually implemented in C++. This includes all modules that expose operating system functionality, such as interaction with the file system or network, and Node.js even exposes the native heap to JavaScript in the form of buffer objects [5]. As a result, many JavaScript objects do not reside in the JavaScript engine's memory space, but rather in native memory. Traditional taint tracking techniques for Node.js (e.g., Karim et al. [25]) are restricted to JavaScript code and therefore cannot observe changes to such objects within native code. Because of this restriction, malicious native code could circumvent a dynamic taint analysis trying to detect software vulnerabilities in JavaScript code. Such malicious code could be injected by insecure Npm [6] modules, which are increasingly common [7].

[1]Releases and documentation can be found at https://www.graalvm.org/

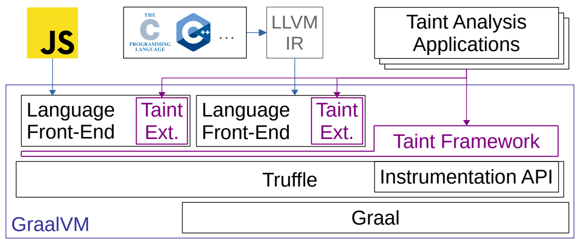


Figure 1. Composition of the proposed taint tracking platform in GraalVM.

In our paper, we are proposing a platform that is capable of tracking taint in multiple programming languages, including those used to implement native extensions. Besides increasing the effectiveness of previous taint analysis applications, such a platform would also help researchers to implement and evaluate new applications of dynamic taint analysis for multiple languages without targeting multiple analysis platforms. In order to make these analyses feasible in production environments, we are also investigating strategies to reduce their performance impact.

3 Efficient, Multi-Language Dynamic Taint Analysis in GraalVM

The goal of our research is to create a platform for the implementation of dynamic taint analyses that are capable of targeting multiple programming languages and of propagating taint when data crosses language boundaries. To achieve this we propose to design our taint analysis platform around language-agnostic abstractions which can be used by concrete taint analyses as well as by front-ends targeting individual languages. To reduce the runtime overhead of such analyses, we plan to investigate how these abstractions can be implemented in such a way that speculative execution and runtime optimizations by a dynamic JIT compiler can be leveraged. We plan to integrate these abstractions and techniques as a taint tracking platform in GraalVM and to use this platform to implement concrete taint analyses that operate on both dynamic language code and native extensions used by it. An overview of the proposed platform in the context of GraalVM can be seen in Figure 1. In the following sub-sections we will further describe this platform and our approach to making it efficient and extensible.

3.1 Background: Truffle & GraalVM

We have chosen GraalVM [46], a multi-language virtual machine, as a basis for our proposed taint tracking platform. As Figure 1 shows, GraalVM contains multiple language runtimes which are based on the *Truffle* framework.

Graal.js [1] is GraalVM's runtime for JavaScript code. Similarly, Sulong [38] can execute LLVM IR [28], which is an intermediate representation into which C and C++ programs can be compiled by, e.g., the Clang[2] compiler. Additional Truffle-based language implementations exist for dynamic languages such as Ruby, Python and R [2], and also for x86 native code [35]. These runtimes use Sulong to execute native extensions of dynamic language programs they execute.

Truffle-based language runtimes parse programs of their respective languages into this framework's common abstract syntax tree representation. We refer to a function parsed into this representation as a *Truffle AST*. Truffle ASTs are in principle language-independent, and Truffle further provides an API for language interoperability. Together, these features enable programs executed on separate Truffle-based runtimes to share functions and even complex, structured values. These capabilities for language interoperability can also be used by Truffle-based runtimes to efficiently implement foreign function interfaces by executing native extensions on Sulong. Truffle also contains a framework for efficient, language-independent program instrumentation [44]. GraalVM further includes the *Graal* Just-In-Time compiler, and uses it to dynamically compile programs it executes, even instrumented ones, to efficient machine code.

3.2 Multi-Language Taint Tracking

The central idea of our proposed platform is the separation of taint analysis functionality into (1) language-independent, (2) language-specific, and (3) analysis-specific components. We plan to implement these components in GraalVM as shown in Figure 1. In the following, we describe them in more detail.

Language-Independent Components The *taint framework* forms the core of our proposed platform. It aggregates language-independent functionality for taint tracking, such as allocating shadow memory, persisting the taint labels for local and global variables, and setting, propagating and retrieving the taint labels attached to execution values as the program executes. By aggregating common functionality in one reusable component we aim to avoid code duplication in analysis implementations and limit the taint extensions to implementing only aspects of taint propagation that are specific to a language implementation.

As shown in Figure 1, we plan to implement the taint framework in part by using Truffle's API for program instrumentation [44]. Using this API, we intend to insert nodes that perform taint propagation into the Truffle AST, which is also shown in Figure 4. The process of propagating the taint labels attached to a value from the expression that produced it to an expression that consumes it is in principle independent of any language semantics. It can thus be used in Truffle ASTs produced from any language runtime and also works between Truffle ASTs produced from different language runtimes. One way to implement this is for the first expression to write the taint labels into a buffer in shadow memory

[2]Clang is available at https://clang.llvm.org/

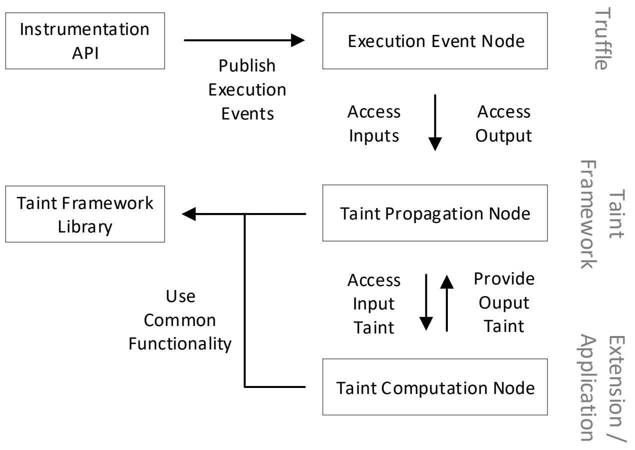


Figure 2. Structure of instrumentation nodes.

which is then read by the second expression. However, we are also investigating more efficient ways to perform this propagation.

Figure 2 shows the general structure of the nodes that perform taint propagation. Truffle provides a base class for nodes that receive execution events from an instrumented node. This base class provides the functionality of accessing both the values flowing into the instrumented node and the value produced by executing it. The taint framework implements this base class in an abstract *taint propagation node*, which contains the logic for retrieving the taint labels attached to the instrumented node's input values and for propagating the taint label of its return value to its parent. *Taint computation nodes* extend such a propagation node with the strategy to compute the taint label of the return value, or to perform other taint propagation actions such as propagating taint through native memory. These computation nodes are implemented by language-specific extensions or concrete analyses, but they can also make use of language-agnostic functionality, such as allocating shadow memory, which is provided as part of the taint framework's library.

Language-Specific Components A *taint extension* integrates a specific language runtime into the taint framework by defining default strategies for taint propagation and storage. To this end, it can reuse functionality provided by the taint framework, such as strategies to store the taint labels for local variables, but can also opt to implement this functionality itself to, e.g., achieve better performance. Since language-specific functionality is kept separate from the core taint framework, our proposed platform can be extended to support additional languages by implementing taint extensions for their language runtimes. As shown in Figure 1, we intend to implement such taint extensions for multiple Truffle-based runtimes.

First, a taint extension implements taint propagation nodes to provide a default strategy for taint propagation within the semantic features of the language targeted by the runtime. To this end, the language runtime must define instrumentation tags [44] to describe the semantic features of the supported language, and attach these tags to each of its AST nodes that represent such a feature. The taint framework maintains a mapping between these tags and the corresponding taint propagation node implementation. While taint extensions define the default mapping, the taint framework provides an API to analysis implementations for changing this default.

Each taint extension essentially defines a taint analysis specific to the language it supports, and the taint framework defines the interfaces needed to integrate multiple such language-specific taint analyses. This design enables taint extensions to instrument their supported language at the level of its features, which results in less taint propagation events than a lower-level instrumentation would exhibit. This comes at the expense of analysis implementers having to understand the semantics of each language they want to support in case they deem the default propagation semantics insufficient. In contrast, a low-level instrumentation, such as on native code or on LLVM IR [3], would represent each language feature by potentially multiple instructions, and exhibit higher analysis overhead as it needs to propagate taint for each of these instructions. In a Truffle AST, a semantic feature of the encoded language may be represented by a subtree rather than a single node. Truffle's instrumentation framework instruments the entire subtree, rather than each node, and propagates input events accordingly [42]. However, even for high-level instrumentation, compiler optimizations are still required to further optimize and avoid redundant taint propagation.

A taint extension also implements the strategy to store and access taint labels within complex objects produced by the respective runtime, such as structured objects or arrays. The taint extension registers this strategy with the taint framework upon application startup, and the taint framework exposes it to other taint extensions, enabling them to also propagate taint for complex objects of another language. By separating the storage strategy from the taint framework, this storage can be optimized to the value implementation. For example, an object with a fixed set of members may provide a slot for each of these members in which its taint labels are stored. In contrast, a low-level program representation like LLVM IR may instead store taint in one common shadow memory for the stack and the heap.

Analysis-Specific Components A *taint analysis application* defines aspects of taint propagation that are specific to a concrete analysis, such as taint sources and sinks, and concrete data types to be used as taint labels. Where the default semantics of taint propagation are sufficient, taint analysis applications should be implementable on our proposed platform in a language-independent manner. While taint analysis applications that require a deviation from these default semantics need to target specific languages, developers of such applications could still re-use the existing analysis code and the knowledge of the analysis platform when applying the application to another language.

Analysis implementers can define taint sources and sinks by implementing suitable taint propagation nodes and replacing the default nodes for the according language constructs. For example, by implementing a taint propagation node for function calls that marks the call's return value as tainted if the name of the called function has a specific name, an analysis implementation can define the function with that name as a taint source. Analysis implementations can use this to, e.g., define taint sources and sinks according to a policy that is external to the program, but they may also expose taint control as a language-specific API that the analyzed programs may access. Similarly, the analysis implementer could change default taint propagation semantics by implementing taint propagation nodes that check an expression's input or output values to determine whether the output value should be tainted or other taint propagation actions need to be taken. This will also enable our framework to be used for evaluating and comparing different taint propagation strategies, as, e.g., Araujo et al. [8] have done. Most notably however, this leaves the choice of whether to consider implicit taint flows [39] up to the analysis application rather than defining it already at the framework-level. The default taint propagation semantics may ignore implicit taint flows to avoid over-tainting, but the analysis application may instead opt to consider implicit flows to avoid under-tainting.

Taint analysis applications can define their own representation of taint labels, based on a base type provided by the taint framework. The main purpose of this base type is to provide a reference to a strategy for merging multiple taint labels, which is used, e.g., when multiple tainted values flow into a taint propagation node to compute the output taint. While this merge strategy may be as simple as a binary or if only boolean flags are used as taint labels, such as in SwordDTA [11], other analysis applications that require more complex taint tags, such as Penumbra [16], may instead require creating a new taint label altogether. The taint framework can provide default implementations for commonly used types of taint labels, such as boolean flags [34, 37, 45, 47] or distinct objects [10, 15, 27, 33]. To this end, we are also looking into how these types of taint labels and merging strategies can be implemented to benefit from compiler optimizations such as escape analysis[3] or partial evaluation[4].

Challenges We see a number of challenges for our research, some of which we provide here. We expect new challenges to arise as we implement our proposed framework.

[3]An optimization that aims to avoid object allocations [41].

[4]Executing parts of the program already at compile time [46].

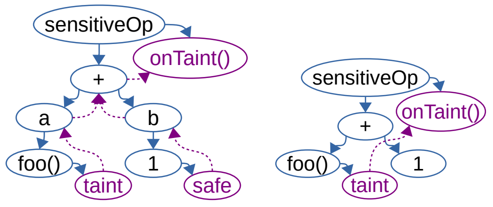
- The main challenge of our work is to determine which functionality should be part of the taint framework and how it can be implemented to suit different kinds of languages and taint analysis applications.The implementation of this functionality should be adaptable enough to enable multiple taint extensions to use them to model the semantics of taint propagation within their respective target languages. One part of this challenge is the granularity to which values can be tainted. Taint analyses for dynamic languages often taint data at the granularity of values and object members [25, 27, 32], while taint analyses for lower-level program representations rather use finer granularity, such as individual bytes [10, 15, 47] or bits [22, 31].
- Another challenge is to design abstractions for retrieving and assigning taint labels that enable a taint extension for one language to access and store taint labels for values that have been created in another language runtime. For example, it is common in Node.js programs that values which are used in JavaScript code to actually reside on the native heap. In GraalVM, this problem is exacerbated since code and values of different languages can be mixed arbitrarily, enabling, e.g., JavaScript code to operate on Python objects and to call functions implemented in Ruby. The interaction between native pointers and dynamic languages is of particular interest here. While native memory may be modified arbitrarily, a dynamic language may not have a concept of a value being partially tainted.
- A third challenge is to design abstractions in the taint framework that are generic enough to enable dynamic taint analysis applications to be implemented in a language-independent manner. This entails not only an abstraction of different language semantics, but also of common sources and sinks of taint such as network or file system interaction.

3.3 Speculative Optimizations and Dynamic Compilation for Efficient Taint Tracking

We are investigating strategies for using the Graal JIT compiler to reduce the execution time overhead of taint analysis applications implemented on our proposed platform. Graal aggressively performs optimizations such as inlining, partial evaluation and escape analysis on any program. Graal also has special knowledge of Truffle ASTs, which enables the compiler to compile such ASTs to efficient machine code. Furthermore, Truffle-based language runtimes can direct the compiler to speculate on assumptions about values and conditions within the code. If these assumptions are invalidated during execution, code that was compiled under these assumptions is deoptimized. We are researching how both generic compiler optimizations and explicit speculation can be used to reduce the execution overhead of taint analysis applications implemented on our proposed platform.

```
    int a = foo(); int b = 1;
    sensitiveOp(a + b);
```

Figure 3. Example Program



(a) Unoptimized AST **(b)** Optimized AST

Figure 4. Optimizing taint propagation for the program of Figure 3 with Graal.

Similar to static taint tracking, optimizing compilers already analyze the flow of data through a program to find opportunities for simplification. We believe that such simplifications can also be applied to optimize away instructions for taint propagation. To illustrate this, consider the example program shown in Figure 3. In this program, a number returned from the function `foo` and a constant are assigned to local variables `a` and `b`, respectively. While `foo` may return a tainted value, the constant cannot be tainted. These local variables are then added, and their sum is passed to a sensitive function, which is instrumented as a taint sink. A Truffle-based interpreter may represent this program as shown in Figure 4a, where continuous lines represent control flow and dotted lines represent the flow of taint. As the Figure shows, taint labels need to be propagated through the assignments to any subsequent reads from the local variables. However, a compiler can reduce the size of this AST significantly by removing the ultimately unnecessary assignments[5]. Furthermore, if the taint labels consist of a boolean flag, which is `true` for tainted data and `false` otherwise, and if the result of the addition operation is tainted if either of its inputs are, then constant propagation enables the compiler to see that the second operand is never tainted. By evaluating the resulting taint tracking logic statically, as is done in the partial evaluation optimization, the compiler is able to determine that the taint label of the value returned by foo is also the taint label of the value flowing into the sensitive operation. Such an optimized AST is shown in Figure 4b. In this example, both taint labels and their merging strategy are highly amenable to common compiler optimizations. One

[5]In the code, the local variables are only used once and as soon as they are written. An optimizing compiler can use this fact to remove the assignments to a named symbol, which may involve pushing a value onto a specific slot on the stack, and instead inline the expressions that produced the values into the expression that uses them.

challenge of our research is how to implement complex taint labels, attach them to execution values, and merge them in a manner that Graal can similarly optimize.

While we believe common compiler optimizations to be effective at reducing the overhead of taint propagation, the compiler may require additional hints to enable them. In the presented example, the compiler is able to partially evaluate the taint propagation logic. This relies on the compiler's ability to first inline both the taint propagation logic and the logic for merging taint labels. As we have described in Section 3.2, the concrete strategy for merging taint labels is defined by the analysis application and encoded in the taint labels themselves. Using Truffle, we can direct the compiler to speculate on the same strategy being used for all taint labels seen at a particular code location, and thus inline it. We expect partial evaluation to be most effective at avoiding redundant taint propagation, but also other common optimizations to be effective at reducing the overhead of taint propagation. For example, escape analysis [41] could potentially avoid the allocation of complex taint labels. However, we are also investigating whether we can provide additional hints to the compiler to enable it to better optimize the code for taint propagation.

Implementations of dynamic taint analysis applications can also benefit from speculative optimizations. For example, dynamically enabling taint tracking on-demand has been explored to allow for an application to more effectively serve as a *honeypot* [36], i.e., to deliberately make the application appear vulnerable in order to attract the attention of attackers and to observe their attack methodology. The code of such a taint analysis may be compiled by Graal under the assumption that taint tracking is inactive, and therefore no taint propagation is performed. This compiled code would then be deoptimized once the assumption is invalidated by enabling taint propagation, after which the code could be compiled again with included taint propagation. Similarly, a taint analysis application may support multiple, run-time-configurable policies for taint propagation and how to handle tainted data in taint sinks. Such an application may direct the compiler to assume the currently active policy as constant and explicitly deoptimize the compiled code once the policy is changed, rather than continuously polling the currently active policy. Consider also a system like SwordDTA [11], which detects and taints the values produced by, e.g., arithmetic expressions that resulted in an integer overflow. By profiling values flowing into the respective expressions, such a system may detect that some of these values are in practice constant and avoid doing expensive checks for whether the result may be tainted in these cases. We are exploring these and other approaches to leverage program specialization in concrete applications of dynamic taint analysis.

```
1   void *concat(void *a, void *b) {
2       uint64_t aSize =
3               polyglot_get_string_size(a);
4       char *cStrA = malloc(aSize + 1);
5       uint64_t cLenA = polyglot_as_string(
6               a, cStrA, aSize, "utf-8");
7       // conversion is omitted for b
8       uint64_t cLen = cLenA + cLenB;
9       char *cStr = malloc(cLen);
10      memmove(cStr, cStrA, cLenA);
11      memmove(cStr + cLenA,
12              cStrB, cLenB);
13      void *ab = polyglot_from_string_n(
14              cStr, cLen, "utf-8");
15      free(cStrA); free(cStrB); free(cStr);
16      return ab; }
```

Figure 5. C function that concatenates two strings for another language such as JavaScript.

```
1   var tainted = 'tainted'
2   taint.taintValue(tainted)
3   var c = Polyglot.evalFile(
4       "llvm", "concat.bc");
5   var n = c.concat("also ", tainted)
6   assert(taint.tainted(n))
```

Figure 6. JavaScript code using the `Polyglot` module provided by Graal.js to load the `concat` function from Figure 5, compiled to LLVM IR, and executing it on Sulong.

4 Evaluation of Feasibility

We show the feasibility of implementing cross-language taint analysis in GraalVM using the example JavaScript code shown in Figure 6, which calls the C function shown in Figure 5. We implemented a simple taint analysis using Truffle instrumentation [44] and used it to track tainted strings across the language boundary.

Figure 6 shows JavaScript code that first uses the `taint` module, which is provided by our taint analysis, to mark a string as tainted (at line 2). It then uses the `Polyglot` module, which is provided by Graal.js, to load an LLVM IR file that contains the `concat` function (at line 3). The JavaScript code finally calls this function with the tainted string and another, untainted string as arguments (at line 5), and uses the `taint` module to check that the returned value is tainted (at line 6).

The `concat` function shown in Figure 5 uses `polyglot_*` functions, which are provided by Sulong as part of its support for Truffle language interoperability, to copy the contents of the JavaScript string objects it received as arguments into native memory it allocates using `malloc` (at lines 2-7). It then allocates another block of native memory large enough to hold the contents of both strings (at lines 8-9). `concat` then uses C's `memmove` function to copy the now native strings into that new block (at lines 10-12), which it subsequently converts back to a String representation that can be used by Graal.js (at lines 13-14). Finally, after releasing the native memory it allocated, `concat` returns this concatenated string (at lines 15-16). `concat` is a simplified version of the actual C++ code that implements the `Buffer.concat` JavaScript function[6] in Node.js, which is used to concatenate two buffer objects that are used, e.g., when interacting with the network.

Our simple taint analysis maintains a global hash-set containing all currently tainted JavaScript strings as well as the addresses of each byte of native memory that contains a part of a tainted string. We used Truffle instrumentation to receive execution events when a function is invoked that may influence the taint status of strings or native memory, such as Sulong's `polyglot_*` intrinsic functions or the `memmove` function, which may propagate tainted data. Based on these execution events, our taint analysis updates the global hash-set accordingly. Our taint analysis also provides the `taint` module used in Figure 6 to set and check whether a string is tainted. Other Truffle language implementations, such as Sulong, could also use the module via Truffle language interoperability. We have verified that our simple taint analysis is capable of supporting this scenario.

In addition to showing functional feasibility of tracking taint between different languages, we also used the code of Figures 5 and 6 as a micro-benchmark. To this end, we measured the peak performance of 60 runs, in each of which the code is executed 100000 times. We define peak performance as the mean of the execution times of the last 45 runs, using the first 15 runs as warmup to give Graal the opportunity to collect runtime information and compile the instrumented code. Compared to a baseline of executing the code without any instrumentation, registering empty callbacks for all execution events supported by Truffle instrumentation slowed down peak performance by only ~0.3%. Van de Vanter et al. [44] describe how Truffle's instrumentation framework leverages Graal's dynamic optimization capabilities to ensure the pure act of relaying execution events to instrumentation code has near-zero impact on peak performance. The instrumentation code itself is further inlined and optimized by Graal, which causes empty callbacks to be effectively discarded. As a result, the execution overhead of a taint analysis based on Truffle instrumentation depends only on the analysis implementation itself. In contrast, tools like LIFT or libDFT are based on instrumentation frameworks that do incur overhead even for empty execution callbacks. Since we did not optimize our simple taint analysis for performance, it introduces a considerable overhead of ~5.5x compared to uninstrumented execution.

[6]The full C++ implementation of this function can be found at: https://github.com/nodejs/node/blob/master/src/node_buffer.cc

5 Related Work

Much research in the area of dynamic taint analysis has focused on reducing its impact on execution time for specific programming languages or analysis platforms. However, despite its potential value, hardly any work has been done in the field of multi-language taint analysis, which we believe can at least partially be attributed to a lack of suitable analysis platforms. In the following, we discuss this related work.

The *LLVM DataFlowSanitizer (DFSan)* [3] is a taint analysis framework built into LLVM and relates to both areas of research. DFSan provides an API to attach user-defined taint labels to application memory, and instruments the LLVM IR into which the analyzed program was compiled by adding instructions to propagate these taint labels. As a result, DFSan is agnostic to the analyzed program's source language, but the lower-level instrumentation also increases the amount of taint propagation actions to be taken. While LLVM can optimize the inserted taint propagation logic together with the program, the optimizer does not have access to runtime profiling information and therefore cannot perform speculative optimizations to the same extent as a dynamic compiler like Graal could. In contrast to our proposed framework, the analyzed program must be modified to call the functions of DFSan's API to explicitly introduce taint. Besides altering program semantics, this API accessibility makes DFSan unsuitable for analysis of adversarial code which could use it to detect and disable the analysis. Like our proposed framework, DFSan supports analysis-defined taint labels and is per default limited to tracking only explicit data flow and propagate taint at byte-level, but could in concrete analysis implementations be extended to support also implicit flows and other propagation granularities. Based on DFSan, different tools have been implemented, such as the Angora fuzzer [13] or Araujo et al. [8].

5.1 Multi-Language Taint Tracking

Previous work on taint tracking in dynamic languages requires developers to manually provide models for the propagation of data in native extensions [20, 21, 25, 40]. However, creating and maintaining such models requires a substantial amount of work, especially for large or frequently changing libraries. Besides, it is subject to human error. Similarly, languages that support native extensions and taint tracking as a first-class feature, such as Ruby [43], rely on programmers to manually maintain the taint status of objects that cross the language boundary.

One could execute both native extensions and the engine running the dynamic language program that uses them on a taint analysis platform for native x86 code [15, 26, 37]. However, this would produce a different semantic fact as for the dynamic language program not its actual code is analyzed, but rather the interpreter executing it. Even though the analysis could be implemented to recover the executed statements of the interpreted code, doing so would again restrict the analysis to a single language and even to a specific interpreter. Similar arguments also apply to the possibility of compiling both high-level code and native extensions to a common intermediate representation like LLVM IR.

Azadmanesh et al. [9] have implemented a language-independent information-flow tracking engine on top of Truffle to enable the implementation of program comprehension tools.Their tool records the data-flow within an application execution, which can then be used for offline analysis. Our work instead aims to enable taint propagation while the analyzed program is running.

5.2 Optimizing Performance of Taint Propagation

Previous work has attempted to reduce the performance overhead of taint propagation by dynamically switching from instrumented program code to uninstrumented code whenever taint propagation is not required. Qin et al. [37] check whether any data flowing into a sequence of instructions or any variables modified by these instructions are tainted. If not, they execute the sequence in uninstrumented mode. Ho et al. [23] and Ermolinsky et al. [18] switch the execution of an entire system from a hypervisor, in which taint is not propagated, to a taint tracking emulator whenever the hypervisor accesses tainted data, and switch back once the emulator has not operated on tainted data for a given time. Compared to our approach of taking advantage of a JIT compiler, these approaches exhibit overhead from frequent context switches, rather than being able to optimize taint propagation in the context of the actual program.

DECAF++ [17] similarly switches between instrumented and uninstrumented execution. Being based on whole-system emulation, it avoids context switches to a hypervisor by instead dynamically enabling and disabling taint propagation in an emulator. If taint propagation is disabled, DECAF++ instruments taint sources, such as memory loads, to check if tainted data is accessed and, if so, enable taint propagation. When all registers are clear of tainted data, DECAF++ disables taint propagation again. In case no tainted data is introduced, DECAF++ exhibits only 4% overhead, compared to the baseline execution overhead of the emulator it is based on, but performance degrades the more taint is introduced. Our proposed framework could similarly be implemented to specialize the instrumentation to such modes, and it additionally benefits from a JIT compiler dynamically optimizing the executing program together with the taint propagation logic, which an emulation-based platform like DECAF++ might not be capable of. What is more, dynamic language programs would exhibit less execution overhead on our proposed platform due to higher-level instrumentation which also targets the program itself rather than the interpreter executing it.

Kangkook et al. [24] manually apply well-known compiler optimizations to the taint tracking logic introduced by the analysis. Kerschbaumer et al. [27] modified a JIT compiler to perform optimized taint propagation in JavaScript programs, complementing an already taint-tracking interpreter. In contrast to that, we propose to use an existing JIT compiler already capable of these and other optimizations and perform them on both the instrumented program and the inserted taint tracking logic together.

Some approaches restrict dynamic taint propagation to those parts of the program, in which static taint analysis is unable to precisely determine the flow of taint [30, 47]. In contrast to that, our proposed framework guides a dynamic JIT compiler to detect and remove unnecessary taint propagation while the program is executing.

Minemu [10] and libDFT [26] both claim to achieve low-overhead dynamic taint analysis for x86 binary code. However, they both achieve this by either taking advantage of features specific to the binary analysis platform they are implemented on or by optimizing themselves for specific CPU architectures. Our approach is more high-level and applies to multiple languages.

6 Conclusion

In this paper we have outlined our proposal for designing a dynamic taint analysis platform that is able to propagate taint across language boundaries and can be extended with support for additional programming languages. The proposed platform leverages dynamic compilation and optimization capabilities already available in the Graal compiler in order to reduce its runtime overhead. By building this platform around a re-usable framework for taint tracking, support for additional programming languages can be added. We have implemented these ideas in an early proof-of-concept dynamic taint analysis on top of GraalVM and used it to demonstrate the feasibility of our approach. The evaluation of this proof-of-concept analysis shows that it is able to propagate taint between code in multiple programming languages. We believe that our proposed platform will enable dynamic taint analysis to perform more detailed taint tracking than previous approaches.

Acknowledgments

We thank all members of the Virtual Machine Research Group at Oracle Labs. Oracle, Java, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] 2019. Graal.js GitHub Repository. https://github.com/graalvm/graaljs. Accessed: 2019-07-12.

[2] 2019. GraalVM Language Runtimes. https://www.graalvm.org/docs/reference-manual/. Accessed: 2019-07-12.

[3] 2019. LLVM Data-Flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed: 2019-09-01.

[4] 2019. Node.js. http://www.nodejs.org/. Accessed: 2019-07-06.

[5] 2019. Node.js API Documentation. https://nodejs.org/docs/latest/api/. Accessed: 2019-08-30.

[6] 2019. Node.js Package Manager. http://www.npmjs.com/. Accessed: 2019-07-06.

[7] 2019. Node.js Security Advisories. https://www.npmjs.com/advisories/. Accessed: 2019-07-06.

[8] Frederico Araujo and Kevin W. Hamlen. 2015. Compiler-instrumented, Dynamic Secret-redaction of Legacy Processes for Attacker Deception. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 145–159. http://dl.acm.org/citation.cfm?id=2831143.2831153

[9] Mohammad Reza Azadmanesh, Matthias Hauswirth, and Michael L. Van De Vanter. 2017. Language-Independent Information Flow Tracking Engine for Program Comprehension Tools. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 346–355. https://doi.org/10.1109/ICPC.2017.5

[10] Erik Bosman, Asia Slowinska, and Herbert Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[11] Jun Cai, Peng Zou, Jinxin Ma, and Jun He. 2016. SwordDTA: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences* 21, 1 (Feb. 2016), 10–20. https://doi.org/10.1007/s11859-016-1133-1

[12] J. Cai, P. Zou, D. Xiong, and J. He. 2015. A Guided Fuzzing Approach for Security Testing of Network Protocol Software. In *2015 6th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. 726–729. https://doi.org/10.1109/ICSESS.2015.7339160

[13] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018-05). 711–725. https://doi.org/10.1109/SP.2018.00046

[14] Brian Chess and Jacob West. 2008. Dynamic Taint Propagation: Finding Vulnerabilities without Attacking. *Information Security Technical Report* 13, 1 (Jan. 2008), 33–39. https://doi.org/10.1016/j.istr.2008.02.003

[15] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 196–206. https://doi.org/10.1145/1273463.1273490

[16] James Clause and Alessandro Orso. 2009. Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/1572272.1572301

[17] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2020. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing. https://www.usenix.org/conference/raid2019/presentation/davanian

[18] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L. Fowler, Murphy Mccauley, Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa Fowler, and Murphy Mccauley. 2010. *Towards Practical Taint Tracking.*

[19] Matthias Grimmer, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Dynamically Composing Languages in a Modular Way: Supporting C Extensions for Dynamic Languages. In *Proceedings of the 14th International Conference on Modularity (MODULARITY 2015)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/2724525.2728790

[20] 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 - SAC '14* (2014). ACM Press, 1663–1671. https://doi.org/10.1145/2554850.2554909

[21] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *POST*. https://doi.org/10.1007/978-3-662-54455-6_3

[22] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. ACM, New York, NY, USA, 248–258. https://doi.org/10.1145/2610384.2610407

[23] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. 2006. Practical Taint-Based Protection Using Demand Emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. ACM, New York, NY, USA, 29–41. https://doi.org/10.1145/1217935.1217939

[24] Jee Kangkook, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. 2012. A General Approach for Efficiently Accelerating Software-Based Dynamic Data Flow Tracking on Commodity Hardware. In *NDSS Symposium.*

[25] R. Karim, F. Tip, A. Sochurkova, and K. Sen. 2018. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Transactions on Software Engineering* (2018), 1–17. https://doi.org/10.1109/TSE.2018.2878020

[26] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2151024.2151042

[27] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Information Flow Tracking Meets Just-in-time Compilation. *ACM Trans. Archit. Code Optim.* 10, 4 (Dec. 2013), 38:1–38:25. https://doi.org/10.1145/2541228.2555295

[28] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis Amp; Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[29] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1193–1204. https://doi.org/10.1145/2508859.2516703

[30] Benjamin Livshits. 2012. *Dynamic Taint Tracking in Managed Runtimes.* Technical Report MSR-TR-2012-114. Microsoft Research. https://www.microsoft.com/en-us/research/publication/dynamic-taint-tracking-in-managed-runtimes/

[31] Alyssa Milburn and Niek Timmers. 2018. Efficient Reverse Engineering of Automotive Firmware. Detroit, MI, USA. https://www.riscure.com/publication/efficient-reverse-engineering-automotive-firmware/

[32] Srijith K. Nair, Patrick N. D. Simpson, Bruno Crispo, and Andrew S. Tanenbaum. 2008. A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electronic Notes in Theoretical Computer Science* 197, 1 (Feb. 2008), 3–16. https://doi.org/10.1016/j.entcs.2007.10.010

[33] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS.*

[34] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. 2005. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous*

Computing (2005) *(IFIP Advances in Information and Communication Technology)*, Ryoichi Sasaki, Sihan Qing, Eiji Okamoto, and Hiroshi Yoshiura (Eds.). Springer US, 295–307.

[35] Daniel Alexander Pekarek. 2019. *A Truffle-based Interpreter for x86 Binary Code.* Master's thesis. Johannes Kepler University Linz, Austria. https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-27719

[36] Georgios Portokalidis and Herbert Bos. 2008. Eudaemon: Involuntary and On-demand Emulation Against Zero-day Exploits. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (2008) *(Eurosys '08)*. ACM, 287–299. https://doi.org/10.1145/1352592.1352622 event-place: Glasgow, Scotland UK.

[37] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, and Y. Wu. 2006. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE Computer Society, 135–148. https://doi.org/10.1109/MICRO.2006.29

[38] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[39] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy*. 317–331. https://doi.org/10.1109/SP.2010.26

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

[41] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. ACM, New York, NY, USA, Article 165, 10 pages. https://doi.org/10.1145/2581122.2544157

[42] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. 2018. Efficient dynamic analysis for Node.js. In *Proceedings of the 27th International Conference on Compiler Construction - CC 2018* (2018). ACM Press, 196–206. https://doi.org/10.1145/3178372.3179527

[43] David Thomas, Chad Fowler, and Andrew Hunt. 2004. *Programming Ruby: The Pragmatic Programmers' Guide* (2nd ed.). Pragmatic Programmers.

[44] Michael Van De Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and Other Tools. *The Art, Science, and Engineering of Programming* 2, 3 (March 2018), 14:1–14:30. https://doi.org/10.22152/programming-journal.org/2018/2/14

[45] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07)* (2007).

[46] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[47] Ruoyu Zhang, Shiqiu Huang, Zhengwei Qi, and Haibing Guan. 2012. Static Program Analysis Assisted Dynamic Taint Tracking for Software Vulnerability Discovery. *Computers & Mathematics with Applications* 63, 2 (Jan. 2012), 469–480. https://doi.org/10.1016/j.camwa.2011.08.001