Multi-language Dynamic Taint Analysis in a Polyglot Virtual Machine*

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

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

Lukas Stadler
Oracle Labs
Austria
lukas.stadler@oracle.com

David Leopoldseder
Oracle Labs
Austria
david.leopoldseder@oracle.com

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

ABSTRACT

Dynamic taint analysis is a popular program analysis technique in which sensitive data is marked as *tainted* and the propagation of tainted data is tracked in order to determine whether that data reaches critical program locations. This analysis technique has been successfully applied to software vulnerability detection, malware analysis, testing and debugging, and many other fields. However, existing approaches of dynamic taint analysis are either language-specific or they target native code. Neither is suitable for analyzing applications in which high-level dynamic languages such as JavaScript and low-level languages such as C interact. In these approaches, the language boundary forms an opaque barrier that prevents a sound analysis of data flow in the other language and can thus lead to the analysis being evaded.

In this paper we introduce *TruffleTaint*, a platform for multi-language dynamic taint analysis that uses language-independent techniques for propagating taint labels to overcome the language boundary but still allows for language-specific taint propagation rules. Based on the *Truffle* framework for implementing runtimes for programming languages, TruffleTaint supports propagating taint in and between a selection of dynamic and low-level programming languages and can be easily extended to support additional languages. We demonstrate TruffleTaint's propagation capabilities and evaluate its performance using several benchmarks from the Computer Language Benchmarks Game, which we implemented as combinations of C, JavaScript and Python code and which we adapted to propagate taint in various scenarios of language interaction. Our evaluation shows that TruffleTaint causes low to zero slowdown when no taint is introduced, rivaling state-of-the-art dynamic taint analysis platforms, and only up to ~40x slowdown when taint is introduced.

*This research project is partially funded by Oracle Labs.

MPLR '20, November 4–6, 2020, Virtual, UK
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8853-5/20/11...$15.00
https://doi.org/10.1145/3426182.3426184

CCS CONCEPTS

• **Security and privacy** → **Information flow control**; • **Software and its engineering** → *Interpreters*; *Runtime environments*.

KEYWORDS

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

ACM Reference Format:
Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2020. Multi-language Dynamic Taint Analysis in a Polyglot Virtual Machine. In *Proceedings of the 17th International Conference on Managed Programming Languages and Runtimes (MPLR '20), November 4–6, 2020, Virtual, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3426182.3426184

1 INTRODUCTION

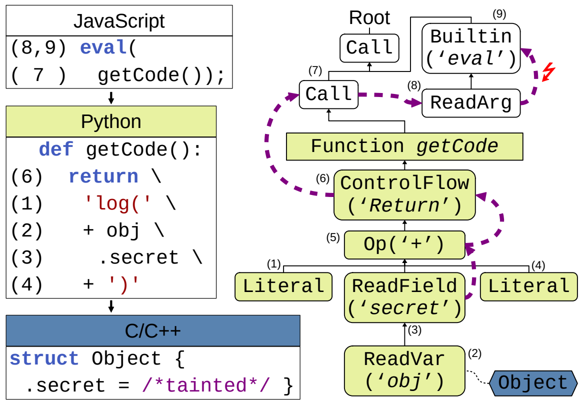
Dynamic taint analysis [43, 55] is a program analysis technique which tracks the propagation of critical data between defined program locations as the analyzed program is executed. Concrete implementations of this analysis technique (we refer to them as *taint analysis applications*) mark sensitive or interesting data as *tainted* by attaching a *taint label* to it. These taint labels are propagated along the flow of data in the analyzed application, enabling the taint analysis application to perform appropriate actions when tainted data reaches certain analysis-defined program locations. The concrete analysis goal determines the program locations at which tainted data is introduced or must be reacted to. These locations therefore differ between taint analysis applications. Dynamic taint analysis has been used extensively to tackle problems of various fields[55]. For example, by tainting data from untrusted sources such as user input such data can be prevented from being used in program locations at which code injection vulnerabilities may occur [20, 46]. However, previous implementations of dynamic taint analysis are all limited to a single programming language or program representation, and therefore fail to propagate taint when data crosses language boundaries.

Programs today often make use of multiple programming languages [31]. For example, many dynamic programming languages such as Python and Ruby support *native extensions*, that is, programs implemented in these languages can invoke native code that was implemented in another language like C. Similarly, by embedding engines for programming languages in a common runtime, interactions between arbitrary programming languages are possible. The popular *node.js* [10] framework is an example of such a *language embedding* in which JavaScript, C++ and WebAssembly code may interact. The choice of implementation languages is not always up to application developers. External libraries are often retrieved from public package repositories using, e.g., *npm* [11] for node.js programs. However, there have been instances of such repositories being hijacked to inject malicious code into unsuspecting users' applications [4, 5, 60, 71]. As dynamic taint analysis is often used to detect vulnerabilities, it is paramount that attackers cannot use the language boundary as a means to evade the analysis. Interactions between multiple programming languages are commonly bidirectional and may involve cross-language function calls as well as sharing of language-specific objects. To support such interactions, and therefore to perform a sound analysis of applications in which they occur, dynamic taint analysis needs to be able to propagate taint across the language boundary.

So far, there has not been a common practical solution for performing dynamic taint analysis on applications in which code of multiple programming languages interacts. One approach is to compile the code to a common program representation, such as native code, and to apply a taint analysis for that representation. While this approach can be applied to languages like C that are compiled anyway, it is unpractical for interpreted high-level languages as it prevents more high-level language-specific instrumentation. Taint analysis applications for higher-level languages may instead determine the data flow effects behind a language boundary from an externally supplied specification [33, 35]. However, especially for larger libraries such specifications are tedious to maintain and are not guaranteed to reflect the actually executed code. We propose a platform for dynamic taint analysis that solves these problems by combining language-specific taint analyses using a common technique for propagating taint across the language barrier.

We propose language-agnostic techniques for propagating taint that can be applied in otherwise language-specific taint analysis applications and can facilitate their interaction. We furthermore propose a core taint propagation semantics for features commonly found in various programming languages to restrict the need for language-specific instrumentation to truly language-specific features. We implemented these techniques and semantics in *TruffleTaint*, a platform for multi-language dynamic taint analysis. TruffleTaint is implemented on top of *GraalVM*[1] [63], a virtual machine capable of executing programs implemented in various programming languages. The novelty of TruffleTaint is its ability to combine taint analyses for various languages which may each propagate taint using language-level instrumentation and which may use language-specific rules for taint propagation. TruffleTaint is currently able to track taint in and between code implemented in C, C++, JavaScript and Python, and can be extended to support additional programming languages. Rather than being a concrete taint analysis application itself, TruffleTaint is a platform for implementing taint analysis applications. To this end, it can also be extended to support different taint propagation semantics.

[1]GraalVM is available at https://www.graalvm.org/.



JavaScript
```
(8,9) eval(
( 7 )  getCode());
```

Python
```
    def getCode():
(6)   return \
(1)     'log(' \
(2)     + obj \
(3)       .secret \
(4)     + ')'
```

C/C++
```
struct Object {
 .secret = /*tainted*/ }
```

Figure 1: Multi-language program that leaks secret data. Numbers in parentheses identify expressions.

In our evaluation we showcase that TruffleTaint is able to propagate taint in programs implemented in multiple programming languages, that it can propagate taint also across the language boundary, and that it is fairly efficient in doing so. We implemented a set of non-trivial multi-language benchmarks that apply common schemes of language interaction and propagate tainted data in these interactions. While we originally focused on interactions of C and JavaScript code, we also extended our benchmarks to Python code and found it quite easy to extend TruffleTaint to also support this additional programming language. Furthermore, we found that in most cases TruffleTaint causes no or only little slowdown on program code that does not operate on tainted data, while code that does operate on tainted data is slowed down by a factor between 5% and 40.37x depending on the instrumented application.

This paper makes the following contributions:

(1) We devised language-agnostic techniques for propagating and storing taint in language-specific runtimes.
(2) We propose a strategy for propagating taint across multiple languages by leveraging a common tainting technique employed by multiple language-specific runtimes.
(3) We defined a core taint propagation semantics for common features of programming languages which can be extended to cover language-specific features.
(4) We implemented our proposed technique in a prototype platform for dynamic taint analysis on top of GraalVM.
(5) We evaluate our platform for dynamic taint analysis with a set of well-known benchmarks regarding aspects of functionality, language support and performance.

This paper is structured as follows. In Section 2 we give an overview of dynamic taint analysis and GraalVM. Section 3 describes TruffleTaint and the language-agnostic tainting techniques it employs. Next, we evaluate TruffleTaint in Section 4. Section 5 presents our plans for extending TruffleTaint in further research. Section 6 discusses related work and Section 7 concludes the paper.

2 BACKGROUND

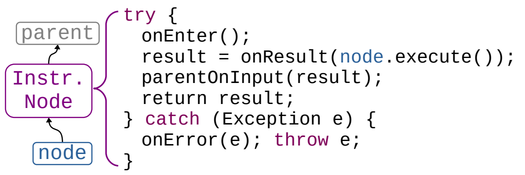
Dynamic taint analysis is a common approach to preventing the exploitation of software vulnerabilities. However, separate runtime environments and propagation rules make taint propagation across language boundaries a complex problem which is unsolved by previous dynamic taint analysis approaches. Consider the program shown in the left-hand side of Figure 1. In the program, the `getCode` function, which is implemented in Python, concatenates two String literals and a value read from a field of an object. That field belongs to an object that was allocated in C/C++ code and contains data which ought not to be revealed. However, the concatenated String value contains code that would print that value. The JavaScript code that calls `getCode` uses JavaScript's builtin `eval` function to execute that code. Language interactions like this can arise, for example, from the use of language embeddings or native extensions. Many languages provide builtins to execute code that is generated at run time, but using them is generally considered a security risk. In our example, the generated code leaks confidential data, but an attacker who is able to manipulate that data could also inject arbitrary code to be executed. By tracking sensitive or unsanitized data, dynamic taint analysis can prevent `eval` from executing code contained in such tainted data. However, language-specific approaches cannot reliably detect whether tainted data is introduced from another language. Such approaches could therefore not track tainted data from a C/C++ object across Python code to JavaScript code.

2.1 Truffle and GraalVM

Our research is based on the GraalVM platform. GraalVM is a virtual machine capable of executing and instrumenting code of various programming languages [63]. At the heart of GraalVM's language support is the *Truffle* framework for implementing runtimes for programming languages. Itself implemented in Java, *Truffle* defines language-agnostic interfaces for the nodes of an abstract syntax tree (AST) and the values flowing through it. Truffle-based language runtimes, we refer to them as *Truffle runtimes*, implement these interfaces in terms of language-specific AST nodes and data types to represent the semantic elements of their targeted programming language. Truffle runtimes represent each function contained in the programs they execute as a *Truffle AST* made up of these AST nodes. Using Truffle's *Polyglot API*, Truffle ASTs and values can be shared between Truffle runtimes [31], which enables the interaction of user code in different programming languages.

Figure 1 also shows the Truffle AST for our running example. Each AST node implements an expression in the program code. The numbers in parenthesis denote which node corresponds to which source code expression and additionally denote the order in which each node is executed. Nodes with white background represent JavaScript expressions, while nodes with green background represent Python expressions. The blue box labeled *obj* represents the object which was allocated in C/C++ code but is stored in a Python scope and accessed by Python code. In contrast to regular functions, builtins such as `eval` have no AST of their own, but are instead implemented as a single node. The children of such nodes only provide the values passed to the builtin as call arguments.

GraalVM supports executing code implemented in, among other languages, JavaScript, Python and LLVM-based languages such as C and C++ and enables complex interactions between these languages [7]. GraalVM contains a Truffle runtime for JavaScript, called *Graal.js* [6], as well as an implementation of node.js based on it. *Sulong* [51] implements GraalVM's support for LLVM-based languages such as C and C++ in the form of a Truffle runtime for LLVM IR, a program representation used by the *LLVM compiler infrastructure* [40]. Various Truffle runtimes for dynamic languages use Sulong to run native extensions of programs they execute. One such runtime is *GraalPython*, which executes Python code.



```
try {
  onEnter();
  result = onResult(node.execute());
  parentOnInput(result);
  return result;
} catch (Exception e) {
  onError(e); throw e;
}
```

Figure 2: Structure of instrumentation nodes that can be inserted into a Truffle AST.

Truffle provides a language-independent framework for dynamic program instrumentation [61], which has been used to implement both language-specific and language-independent tools. This *instrumentation framework* enables analysis tools to instrument the nodes of a Truffle AST. In support of this, Truffle runtimes annotate the nodes they implement with *semantic tags*. A semantic tag represents a semantic construct such as *function call*, *literal expression* or *binary operation*. Analysis tools can use these tags to determine the semantics of the instrumented nodes independent of their implementation. Semantic tags can be language-specific or language-agnostic, and they can be defined by Truffle, Truffle runtimes or Truffle-based tools. GraalVM's debugger back-end [39, 61], for example, uses only language-agnostic tags defined by Truffle such as `Statement` and `FunctionRoot` to identify nodes at which to set breakpoints and to suspend execution during stepping [61]. Nodes can also provide additional metadata, e.g., the name of a builtin they implement or of an object field they access. The nodes in Figure 1 are labeled with the tags (e.g., *Literal*, *ReadVar*) and metadata they provide.

Dynamic analyses can be implemented on top of GraalVM in the form of Truffle-based *instruments*. Truffle defines *instrumentation nodes* which Truffle's instrumentation framework can insert between nodes and their original parents in the AST as shown in Figure 2. The pseudo-code in this example illustrates that an instrumentation node executes callbacks whenever the node it instruments is entered, when one of its children produces a value, and when it returns either successfully or with an error. Instruments implement instrumentation nodes for specific semantic tags to execute instrumentation code in these callbacks. Like Truffle runtimes, instruments can also use the Polyglot API to interact with language-specific objects. Since the structure of instrumentation nodes is language-independent, they can be used across language runtimes. This language-independence of the underlying instrumentation framework enables the same Truffle instrument to support multiple languages. To do so, the instrument can either implement language-agnostic instrumentation nodes or it can target language-specific semantic tags of multiple languages with specialized instrumentation nodes.

Due to its multi-lingual nature and instrumentation capabilities, the GraalVM [63] platform is suited to support cross-language taint propagation. GraalVM supports both language embeddings and dynamic language programs using native extensions. The platform's support for program instrumentation allows for fine-grained, language-specific instrumentation that can be used for implementing language-specific dynamic taint analyses. At the same time, GraalVM's support for language interoperability offers an opportunity to integrate these language-specific instrumentations to also enable cross-language dynamic taint analysis.

2.2 Dynamic Taint Analysis

Taint analysis[43, 55], also referred to as *taint tracking*, is a program analysis technique which aims to detect data dependencies between sensitive program locations. Program locations that produce sensitive data are referred to as *taint sources*. Data originating from a taint source is marked as *tainted* by attaching a *taint label* to it. For each kind of operation supported by the targeted programming language, a *propagation semantics* specifies a rule to determine whether the values produced by this operation should be tainted. For example, a rule that often appears in various propagation semantics is that the result of an addition is tainted if either of the addition's inputs was tainted. A data dependency can be detected when tainted data reaches certain program locations, which are referred to as *taint sinks*. The selection of taint sources, taint sinks and propagation semantics depends on the purpose for which taint analysis is applied, and therefore varies between concrete taint analysis implementations.

In contrast to static taint analysis that is performed at compile time, *dynamic taint analysis* is performed at run time [55]. It is commonly implemented by instrumenting the program to be analyzed so that it propagates taint labels. Such instrumentation employs a *tainting technique*, that is, a mechanism to associate a taint label to a value. That tainting technique determines how to access the taint labels of values flowing into an operation and how to attach taint labels, which are determined by the implemented propagation semantics, to the values produced by that operation. Compared to a static taint analysis, a dynamic taint analysis can use its access to run-time data for more precise taint propagation and more versatile instrumentation of taint sources and sinks. For example, a dynamic taint analysis can support selective taint sources, propagate taint according to the observed control flow, and access the actual values in taint sinks. While system-level approaches to dynamic taint analysis exist [26, 29, 69], application-level dynamic taint analysis can allow for a more fine-grained selection of taint sources and sinks, e.g. specific functions or conditions instead of system calls. Application-level dynamic taint analysis is commonly implemented by extending a language runtime, by applying source transformations, or on top of a binary analysis platform. Some programming languages, e.g., Ruby [13], Perl [12] and Ballerina [1], have limited built-in support for marking values as tainted. They do not, however, feature adaptable propagation semantics and do not propagate taint in native extensions.

Dynamic taint analysis is well suited to detect certain kinds of injection vulnerabilities [41, 46, 62, 70]. Our running example from Figure 1 also constitutes such a vulnerability. By using dynamic taint analysis to track external data such as user input, data retrieved from the file system, from the network, or from untrusted libraries, the `eval` function can detect when it is called with such tainted data and throw an error instead of executing potentially malicious code contained in it. In the Truffle AST shown in Figure 1 the flow of tainted data is depicted using dashed purple arrows. The Figure assumes that, when the Truffle AST is executed, the value stored in `obj.secret` is tainted because it originated from one of the aforementioned external sources. The Figure further assumes a propagation semantics which defines that a concatenated String value is tainted if at least one of the String values used in the concatenation was tainted. Therefore, the concatenated String value is tainted and when it used as input to `eval` the instrumentation throws an error to prevent the code it contains from being executed. Dynamic taint analysis also has applications to vulnerability detection in binaries [20], attack prevention [32, 37, 47, 54, 56, 57, 64, 68], malware analysis [28, 46, 49, 65, 67], fuzz-testing [21, 22, 30, 34], debugging [24], program comprehension [44, 45, 66], reverse engineering [19, 25, 27], and other fields.

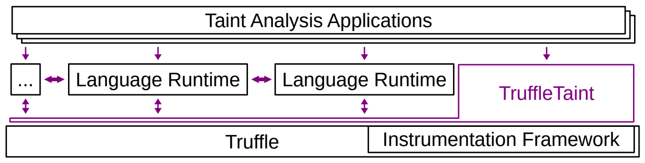


Figure 3: TruffleTaint System Overview.

3 TRUFFLETAINT

In this paper we propose *TruffleTaint*, an extensible platform for multi-language dynamic taint analysis built on top of GraalVM. TruffleTaint applies language-agnostic tainting techniques in separate runtimes for various programming languages. Each of these runtimes provides support for language-specific instrumentations that apply these generic techniques.[2] By using a common tainting technique, these instrumentations can share the taint labels attached to values that cross the language boundary by traversing from one runtime to another. As a result, a taint analysis application built on top of TruffleTaint can target multiple programming languages each with language-specific propagation semantics and can track tainted data as it crosses language boundaries. If propagation semantics for individual features or actions to be performed in taint sinks are similar in multiple languages, these applications can also reuse code between language-specific instrumentations. TruffleTaint embraces this principle by providing a default propagation semantics and implementation thereof for several features often found in programming languages. This default semantics and its implementation can be reused and extended to implement language-specific propagation semantics.

An overview of TruffleTaint is shown in Figure 3. TruffleTaint is implemented on top of Truffle and its instrumentation framework. Truffle runtimes additionally integrate with TruffleTaint to support its tainting techniques. TruffleTaint can be used to implement taint analysis applications of various kinds. Such applications may target

[2] Our previous work [38] suggested that a similar composition of language-agnostic, language-specific and analysis-specific components can potentially be used for a dynamic taint analysis platform. This paper describes in the form of TruffleTaint how that suggestion can be realized in practice.

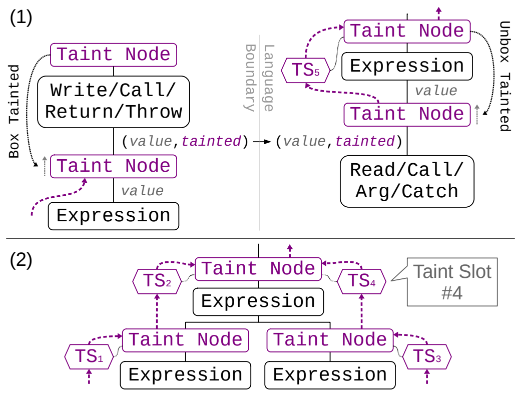


Figure 4: Tainting techniques in TruffleTaint.

implementations of language-specific propagation semantics, taint sources and taint sinks to semantic tags provided by the respective language runtimes. TruffleTaint also provides language-agnostic semantic tags for common language features such as *variable write* or *function call*, which taint analysis applications may target as well. Truffle runtimes support these tags alongside equivalent but language-specific tags of their own. Targeting these tags can avoid duplication of instrumentation code for a feature that appears in multiple programming languages.

Taint analysis applications built on top of TruffleTaint implement propagation semantics in terms of *taint nodes*. TruffleTaint provides a base class for such taint nodes, which is itself an extension of Truffle's instrumentation nodes and implements TruffleTaint's tainting techniques. Taint nodes are inserted into the instrumented program using Truffle's instrumentation framework. When an instrumented node returns a value, the corresponding taint node decides based on its implemented propagation semantics whether that value is tainted. If so, the taint node uses an appropriate tainting technique to attach a taint label to that value. For each language-specific semantic tag, and therefore for each language feature, as well as for each language-agnostic tag, taint analysis applications can define specialized taint nodes that implement the corresponding part of a propagation semantics.

3.1 Taint Propagation

Taint nodes based on TruffleTaint use two language-agnostic tainting techniques, which are illustrated in Figure 4. The first technique is applied if an expression produces a tainted value that flows into a potential language boundary. The taint node instrumenting that expression combines both that value and the corresponding taint label into a *boxed value*, and lets that boxed value flow into the language boundary instead. Conversely, a taint node that instruments an operation which reads such values, potentially in another language, may split up boxed values and propagate the taint labels stored in them using the second tainting technique instead. The second tainting technique is used for operations that create new values. Taint nodes instrumenting these operations allocate *taint slots*, that is, special storage locations into which the taint labels of their input values are stored directly. An *input taint node*, i.e., the taint node that determines the taint labels of values that flow into a certain expression, asks its *parent taint node*, i.e. the taint node instrumenting that expression, which tainting technique to use.

TruffleTaint's first language-agnostic tainting technique is based on *boxed values*. This technique is applied to propagate taint for operations that only move or store existing data. More specifically, it is applied to propagate taint for values that are (1) written to a field of an object, to an element of an array, or to a named symbol, (2) used as argument to a function call, (3) returned from a function, or (4) thrown as an exception. As is shown in Figure 4, taint nodes that instrument such operations direct the corresponding input taint nodes to employ *boxing*. This means that when such an input taint node determines that the input value is tainted, it merges that input value and the corresponding taint label into a boxed value. Figure 4 depicts boxed values, which consist of a value and an arbitrary attachment[3], as tuples of a value and the tainted label. The input taint node returns that boxed value instead of the original input value to the instrumented move or store operation. Truffle runtimes need to accept boxed values as valid inputs to these operations as part of their runtime support for TruffleTaint. Whether such an input value is tainted can be determined by checking whether it is a boxed value and contains a taint label as an attachment.

Operations that compute new values from their inputs typically require a semantic understanding of these inputs. These operations include, among others, arithmetic, conditional and bitwise operations and some builtin functions. To access these inputs, these operations would need to internally *unbox* these values, i.e., extract the original values while ignoring the attachment objects. For example, an integer addition needs to know the type and content of its operands to produce a meaningful result, thus unboxing is necessary. TruffleTaint permits such unboxing; the taint labels for new values are determined and attached solely by taint nodes that instrument the nodes which produce them. By keeping runtimes and propagation semantics separate, TruffleTaint is not limited to a single propagation semantics. Instead, each individual taint analysis application can implement its own propagation semantics.

Boxing tainted values that flow into expressions which immediately unbox them would negatively impact TruffleTaint's run-time overhead due to many short-lived object allocations. To avoid this overhead, TruffleTaint employs a separate tainting technique for such expressions. Each taint node anyways allocates a *taint slot*, that is, an object which can hold a taint label, for each input to the expression it instruments. When that expression receives a boxed value as input, the respective taint label is stored into the corresponding taint slot until the taint node returns a value. When the propagation semantics is applied to decide whether to taint that value, the taint node can retrieve the taint labels of the inputs even if these inputs were internally unboxed. In TruffleTaint's second tainting technique, the input taint nodes are directed to store the taint labels for the respective input values into the taint slots of their parent taint node directly. This way, unnecessary boxing can be avoided. In practice, this second tainting technique also saves implementation effort in Truffle runtimes. Since values are only

[3]While TruffleTaint currently supports only one taint label, the tainted flag, we plan to eventually support arbitrary analysis-defined taint labels.

boxed for operations that do not need to unbox them, other operations need not actually implement unboxing and can instead just throw an exception if a boxed value occurs unexpectedly.

TruffleTaint enables the propagation of taint labels across language boundaries by defining a common representation for boxed values. TruffleTaint requires that tainted values may cross the language boundary only in the form of a boxed value. Since the instrumentations for the value-producing language and for the value-receiving language use the same kind of boxed values, which language produced the tainted value is irrelevant for the instrumentation. Scenarios in which tainted data may cross the language boundary include (1) function calls to another language, (2) accesses to a field or array element of an object defined in another language, and (3) catching an exception thrown by another language. On the value-producing side, taint nodes that instrument nodes which may make data accessible to another language direct their input taint nodes to box all tainted values flowing into these nodes. On the value-receiving side, nodes that access values which could have been tainted in another language may thus return boxed values. The taint nodes instrumenting these nodes may *unbox* these values and propagate the corresponding taint labels in some other way. This unboxing is necessary if the propagation semantics determines that these values should not be tainted and thus the taint labels need to be removed. However, the respective parent taint nodes may demand propagation by boxing too, in which case boxed values that are still tainted need not be unboxed.

TruffleTaint uses boxing to propagate taint for all operations which could make data accessible to another language, regardless of whether they actually do. For example, call arguments are boxed even if caller and callee are implemented in the same language. Also, tainted values that are written to objects are boxed even if that object is only ever accessed in the same language. Because of this, language-specific taint nodes do not need to handle interactions between multiple languages separately and language-specific taint instrumentations can interact seamlessly.

TruffleTaint requires Truffle runtimes to support boxed values as inputs for all nodes that move or store data and may thus be instrumented by taint nodes that force such inputs. Conversely, nodes that access this data again must be able to return boxed values. Attachments also must not be lost or changed between data being stored and being read again. For example, when a boxed value is stored to a field of an object, the next time a node reads this field it needs to get a boxed value with the same attachment. Similarly, when builtins like LLVM's `memmove` move or copy multiple values the according attachments must be moved and copied, respectively, as well. These restrictions allow Truffle runtimes various freedoms in working with boxed values. One such freedom is the ability to internally access the original values stored in boxed values, ignoring the attachments. For example, a node that catches exceptions of specific types may access the original exception object to determine whether to catch or rethrow a boxed exception.

Another aspect of runtime support required by TruffleTaint is the presence of semantic tags of sufficient detail. For example, tags provided by a Truffle runtime must clearly separate operations that move, read or store data from operations that create new data to allow for selecting the correct tainting technique for each. Because of this, in Sulong LLVM IR instructions are represented as nodes

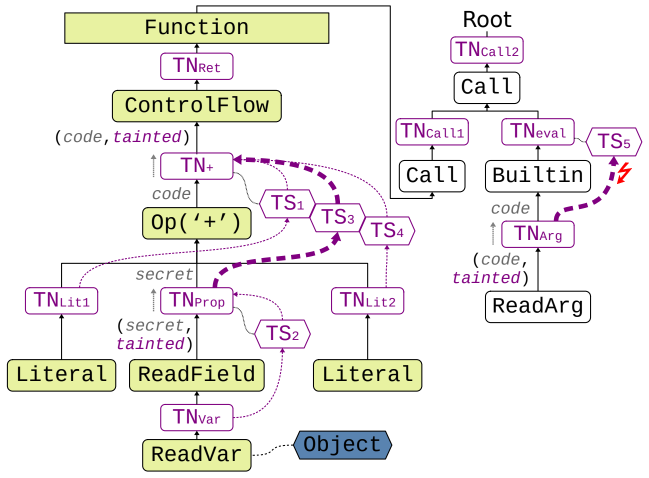


Figure 5: Taint propagation using boxed values and taint slots in the example AST of Figure 1. Taint nodes are labeled *TN*. Purple arrows indicate taint propagation via taint slots while gray arrows indicate boxing or unboxing.

that read the inputs, which are children of the node that executes the instruction, which is itself a child of the node writing the result. Sulong provides separate tags for each of these nodes, even though producing a value and storing that value in a frame slot are an atomic action in LLVM IR. If value computation and value storage could not be instrumented separately, instrumentation could not apply taint labels to the value before it is stored. Truffle runtimes are also required to provide semantic tags of sufficient detail to target each semantic element of their respective language separately.

Using boxed values to propagate taint in function calls as well as for arbitrary storage operations simplifies instrumentation code. This simplification is most noticeable in terms of language support. Truffle runtimes already implement the language-specific semantics of many high-level operations. For example, storing a value to a variable may involve traversing runtime-specific data structures for a scope hierarchy. By boxing the value to store, the instrumentation can delegate such implementation details of language-specific semantics to the Truffle runtimes that already implement them. This delegation also enables TruffleTaint to provide several language-agnostic semantic tags for its default propagation semantics. Furthermore, the boundary between value storage and function calls is blurred in some languages. For example, in JavaScript objects can define *Getter* and *Setter* functions which are implicitly invoked to access certain fields. Since instrumentation propagates taint for both value access and function calls in the same way, this is supported by default. Similarly, whether a local variable receives its value from an expression or from a caught exception object is irrelevant.

Taint nodes implement the *onResult* function of instrumentation nodes, which is referenced in Figure 2, to determine and propagate the taint label of the produced value. More specifically, the following steps are performed.

(1) Apply propagation semantics. Based on the propagation semantics the taint label of the output value is computed. To do so, the taint node can access the taint labels of the operation's input

values which are stored in the respective taint slots. The taint node can also access the instrumented node's input values as well as the output value that is produced. Taint analysis applications can implement taint sources and taint sinks in this step by tainting the produced value or reacting to the inputs' taint labels.

(2) Select tainting technique. The parent taint node is queried which tainting technique to use to propagate the taint label for the input value produced by the instrumented node.

(3) Update Taint Slots. If requested by the parent taint node, the taint label of the value produced by the instrumented node is stored into the corresponding taint slot. The concrete taint slot to use is queried from the instrumented node's parent's taint node, which may allocate that slot on demand. That parent taint node may react to the particular taint label being stored, allowing it to perform actions as a taint sink before the instrumented node is executed.

(4) Apply Boxing or Unboxing. If requested by the parent taint node, the produced value is boxed or unboxed depending on the value and its determined taint label. Some nodes that read data (for example read accesses to local variables or function calls) may produce a boxed value themselves. However, if the propagation semantics determines that that value should not be tainted after all, that value is unboxed.

Figure 5 illustrates how a potential taint analysis application built on top of TruffleTaint propagates taint in order to prevent the data leak in our running example from Section 2. The first node to produce a value (i.e., the first String literal) produces a value that is not tainted. This value flows into an expression that computes new data, namely the String concatenation. The taint node instrumenting the literal node, TN_{Lit1} thus stores the corresponding taint label, that is, an empty label, into a taint slot allocated by TN_{+}. Next, the value stored in the local variable `obj` is not tainted and therefore the node reading it does not produce a boxed value. The `ReadField` node allocates a taint slot to store that its input is not tainted. `ReadField` returns a boxed value, which contains the tainted taint label as an attachment, but the concatenation requires its inputs to be unboxed. Thus that taint label is stored in a second taint slot allocated by TN_{+}. Like the first String literal, the second one is not tainted either. The value returned by TN_{+} is tainted, but will also be returned from `getCode`. TN_{Ret} therefore directs TN_{+} to box this value instead of allocating a taint slot. In the JavaScript code, the value returned by the call to `getCode` is directly used as an argument in another call and therefore remains boxed. As a result, the node reading `eval`'s argument returns this boxed value. The corresponding taint node, TN_{Arg}, unboxes it since the `eval` builtin requires unboxed inputs. However, when TN_{eval} receives the tainted label for the String which contains the code to execute, it can abort the program before that code would be executed. TN_{eval} thus acts as a taint sink that effectively prevents the secret value that originated behind a language boundary from being leaked.

3.2 Propagation Semantics

TruffleTaint aims to support the implementation of arbitrary taint propagation semantics. Taint analysis applications built on top of TruffleTaint implement propagation semantics in terms of taint nodes. Nodes that provide a specific semantic tag are instrumented by taint nodes that implement the propagation semantics for the specific language feature denoted by that tag. This requires taint analysis applications to provide taint nodes for each semantic feature of each language they wish to support.

Table 1: Language-agnostic semantic tags defined by TruffleTaint and expected tainting technique for each input.

Semantic Tag	Input Values	Tainting Technique
`UnaryOp` / `BinaryOp`	Operands	Taint Slot
`WriteVar`	Value	Boxed Value
`ReadVar`	-	-
`WriteField`	Target Object	Taint Slot
	New Value	Boxed Value
`ReadField`	Source Object	Taint Slot
`WriteElt`	Target Object	Taint Slot
	Identifier	Taint Slot
	New Value	Boxed Value
`ReadElt`	Source Object	Taint Slot
	Identifier	Taint Slot
`Call`	Arguments	Boxed Value
	Call Target	Taint Slot
`ReadArg`	-	-
`Literal`	Elements	Boxed Value
`Cast`	Source Value	Both
`ControlFlow`	Various	Per Control Flow Kind
`Input/Expression/ Statement`	Optional	Taint Slot
`Builtin`	Optional	Per Builtin

There are certain semantic features that often occur in various programming languages. For example, most languages have a concept of named variables and feature corresponding read and write operations. TruffleTaint provides language-agnostic semantic tags for many similarly generic operations, which are listed in Table 1. Truffle runtimes annotate their nodes with these tags, which are generic enough to enable various kinds of dynamic analyses. Targeting instrumentation to these tags can avoid code duplication if the actions to be performed for the corresponding features are equivalent in multiple languages.

In addition to implementing a propagation semantics, each taint node must also choose tainting techniques for the taint labels of each input to the operation it instruments. As stated in Section 3.1, this choice depends on the kind of instrumented operation and how it processes the respective input values. Table 1 states appropriate choices for each semantic tag defined by TruffleTaint.

TruffleTaint defines a default propagation semantics for the language features covered by its language-agnostic semantic tags. Taint propagation semantics for various languages often contain similar rules for these features, which form the basis of this default propagation semantics. TruffleTaint also provides taint nodes that implement this default propagation semantics and use the tainting techniques stated in Table 1. By using this semantics and its implementation as a basis, the effort to support the full semantics of a particular programming language can be focused on language-specific features. We applied this default taint propagation semantics, extended to support some language specific builtins, to propagate taint in LLVM IR, JavaScript and Python code.[4]

Our work focuses on propagating taint across language boundaries and thus completeness with regards to language features that do not involve that boundary is out of scope. Nevertheless, TruffleTaint's language support is sufficient to analyze the non-trivial JavaScript, Python and C benchmark programs we used in our evaluation. To cover all features of these languages, either specific taint nodes can be implemented or the language-specific features can be broken down to already supported features. The latter approach may be desirable, e.g., for language features that constitute syntactic sugar as it requires no additional taint nodes. However, this is not possible for, e.g., features related to asynchronous execution, and may increase the run-time analysis overhead due to the increased number of instrumentation events.

In the following, we list the language-independent semantic tags provided by TruffleTaint and describe TruffleTaint's default taint propagation semantics for nodes annotated with them.

`UnaryOp` & `BinaryOp`. These tags represent operations that compute a new value based on one or two input values, respectively. That output value is tainted if at least one of the input values is tainted. The operator, such as `+` or `!`, is provided as metadata by the instrumented node. Some taint analysis applications, such as Cai et al. [20], require this metadata to determine, e.g., whether an overflow occurred in the operation. Furthermore, it is not uncommon in low-level taint analyses to drop taint when, e.g., a value is `XOR`'d with itself or `AND`ed with a constant `0`.

`WriteVar` & `ReadVar`. Nodes that provide either of these tags access a named variable, e.g., a local variable, and provide the name of that variable as metadata. Instrumentation delegates choosing the correct variable and its corresponding storage location to the instrumented node. Corresponding taint nodes ensure that tainted inputs to write accesses are boxed, and that boxed values returned from read accesses are unboxed depending on the parent taint nodes' choice of tainting technique.

`WriteField` & `ReadField`. Nodes that provide either of these tags access a field of an object and provide the name of that field as metadata. The first input to such nodes is the object whose field to access. Instrumentation delegates the access scheme to the instrumented node. Corresponding taint nodes ensure that tainted inputs to write accesses are boxed, and that boxed values returned from read accesses are unboxed depending on the parent taint nodes' choice of tainting technique. Values read from tainted objects are also tainted.

`WriteElt` & `ReadElt`. Nodes that provide either of these tags access an element of an array or a dynamic property of an object. The first inputs to such nodes are the object which to access and the identifier of the element or property to access. Corresponding taint nodes ensure that tainted inputs to write accesses are boxed, and that boxed values returned from read accesses are unboxed depending on the parent taint nodes' choice of tainting technique. Values read from tainted objects are also tainted.

[4]Note that for reason of brevity in Figures 1 and 5 we omitted the `Cast` nodes and reduced two nested `BinaryOp` nodes to a combined node for the String concatenation.

TruffleTaint currently ignores the taint status of element indices. This decision for a propagation semantic is not uncommon. For example, Araujo et al. [16] have found that ignoring the taint status of a pointer when reading from it does not significantly reduce analysis precision but instead reduces overtainting, and our decision is to a similar effect. However, writing to a property of a JavaScript object may, in fact, modify the object by adding that property. Therefore, the presence or absence of a property on an object already constitutes an *implicit data dependency*. Since supporting implicit data dependencies is not in scope for TruffleTaint's reference implementation, this is an acceptable limitation. However, in JavaScript it is also possible to iterate over all properties of an object. By not storing taint labels for them, TruffleTaint misses an explicit data dependency here. It would be possible to store taint labels also for properties in addition to their values by requiring language runtimes to support property identifiers with metadata, but since this feature was not required to correctly run our benchmark applications, its implementation is part of future work.

`Literal`. Nodes that provide this tag return a literal. Literals are not tainted, but may contain values which are provided as boxed inputs by the instrumented children of these nodes.

`Call`. Nodes that provide this tag perform a call to an either user-defined or builtin function. While the call arguments are boxed since the call target may be implemented in another language, the call targets themselves are unboxed inputs to these nodes. The values returned by these nodes, i.e., the values returned by the called functions, are unboxed depending on the parent taint nodes' choice of tainting technique.

`Builtin`. Nodes that provide this tag implement a builtin and provide the name of that builtin as metadata. As builtin functions are language-specific per definition, taint analysis applications need to provide a corresponding taint node for each builtin that is used in programs they intend to analyze. Although languages often define a large number of builtins, not all of them are frequently used. For example, Rigger et al. [52, 53] found out that most software projects in the C language only use a small subset of the inline assembly and compiler-specific builtins available.

`ReadArg`. Nodes that provide this tag read an argument value that their parent function or builtin was called with. Corresponding taint nodes ensure that boxed argument values are unboxed depending on the parent taint nodes' choice of tainting technique.

`ControlFlow`. Nodes that provide this tag affect control flow. The kind of this control flow, which may include conditions, loop entry and exit, throw or catch statements, and function returns, is provided as metadata. For reasons already stated, the inputs for *function returns* and *throws* are boxed. Less generic kinds of control flow, such as Python's `yield` and JavaScript's `await` keyword, can be supported by language-specific taint nodes.

`Cast`. Nodes that provide this tag represent an explicit or implicit type cast. Casting a value may entail calling a conversion function. For example, adding a number and an object in JavaScript code may involve executing a conversion function defined in the object. Depending on the parent taint nodes' choice of tainting technique, taint nodes targeting this tag may unbox boxed values returned from such internal calls. In general, the taint labels of the input value are applied also to the output value.

`Input`, `Expression` & `Statement`. Nodes that provide these tags have no specifically defined semantics, but produce a value without side effects. Such a value is tainted if the node producing it received a tainted input. The presence of such nodes is a remnant of previous support for Truffle's instrumentation framework by certain Truffle runtimes, such as GraalVM's Python and JavaScript runtimes.

3.3 Platform Extensibility

TruffleTaint is intended as a platform to build various dynamic taint analysis applications upon. This is suggested in Figure 3, and is also the reason why TruffleTaint is implemented using the Truffle instrumentation framework rather than being integrated into Truffle and the language runtimes based on it. Taint analysis applications can change all aspects of TruffleTaint. This includes choosing whether to use and extend the default propagation semantics or to use only their own taint nodes. By providing taint nodes for certain language features, taint analysis applications can also implement taint sources and taint sinks.

To evaluate TruffleTaint, we implemented a reference taint analysis application on top of it. This reference application applies the default propagation semantics, extends it with support for some language-specific builtins, and additionally provides builtins for each supported language which an instrumented program can use to manually introduce and check for tainted values. These builtins enable programs to query and change whether a value is tainted. They are listed in Appendix A for reference.

Our reference taint analysis application supports JavaScript, Python and LLVM-based languages such as C/C++ in the form of LLVM IR. We modified GraalVM's Truffle runtimes for LLVM IR, JavaScript and Python code to support boxed values. We additionally extended the JavaScript and Python runtimes to provide TruffleTaint's language-agnostic tags. However, TruffleTaint's tags are intended to represent source-level operations, but Sulong does not have enough information to instrument LLVM IR at this level. Thus we instead targeted language-specific tags provided by Sulong to instrument LLVM IR programs. Since TruffleTaint supports this by design, the taint instrumentations for LLVM IR and for the dynamic languages interact seamlessly. The instrumentation for LLVM IR even reuses taint nodes from the default propagation semantics for equivalent features in LLVM IR. For example, LLVM IR-specific tags equivalent to `Read-/WriteVar`, `Literal`, `Call`, and `ReadArg` are targeted with the same taint nodes. Random byte-wise access to heap memory has proven to be the most significant difference between instrumentations. To support such read accesses, Sulong takes advantage of the fact that TruffleTaint currently only supports a single taint label. When Sulong reads multiple bytes and finds that only some of them are boxed values, it can assert that all these boxed values contain the same attachment. This attachment is then used when boxing the result of that read operation. To enable analysis-defined taint labels instead, TruffleTaint could provide an API for merging such taint labels.

4 EVALUATION

TruffleTaint is a novel framework for dynamic taint analysis that targets applications in which code of multiple programming languages interacts. For a taint analysis framework to be practical,

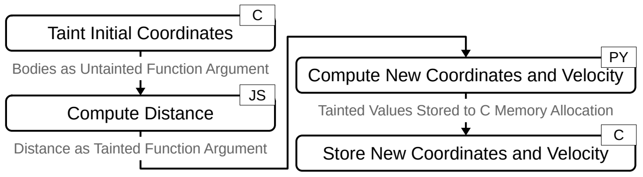

Taint Initial Coordinates
C
Bodies as Untainted Function Argument
Compute Distance
JS
Distance as Tainted Function Argument
Compute New Coordinates and Velocity
PY
Tainted Values Stored to C Memory Allocation
Store New Coordinates and Velocity
C

Figure 6: Language interaction and taint flow in our multi-language implementation of the NBody benchmark.

the imposed run-time overhead has to be low. Furthermore, benchmarks should show the effectiveness of language interoperability. However, existing benchmarks are normally designed to use only a single programming language instead of multiple ones. To demonstrate TruffleTaint's capabilities we therefore implemented several benchmarks from the *Computer Language Benchmarks Game* [3] as a combination of C, JavaScript and Python code and adapted them to operate on tainted data. These benchmarks are often used to demonstrate the performance of a programming language implementation by implementing a fixed algorithm in that language and comparing the results to those achieved with other languages. For our purposes, we developed multi-language versions of these benchmarks[5] that are aimed at stressing the performance of taint propagation across multiple languages. These benchmarks show that TruffleTaint is indeed capable of multi-language taint propagation. To execute these benchmarks, we compiled the C code to LLVM IR for execution on Sulong in managed mode [2, 14]. Our benchmarks show that the tainting techniques we implemented in TruffleTaint enable it to propagate taint across multiple languages, and that TruffleTaint does not exhibit prohibitive slowdown in doing so. In the following we will illustrate the structure of these benchmarks and present initial performance numbers.

4.1 Cross-Language Taint Propagation

We demonstrate how we implemented our multi-language benchmarks using the *NBody* benchmark as an example. NBody computes the position and velocity of five celestial bodies in a star system after a fixed time interval has elapsed for a certain number of times. We implemented this benchmark using the C, JavaScript and Python programming languages, and connected them using GraalVM's polyglot API which is also used in several Truffle runtimes to implement the foreign function interfaces of the respective languages. In our implementation, the planetary bodies are represented as C *structs* which are allocated by LLVM IR compiled from C code and are executed on Sulong. Each of these structs stores the current position and velocity of the respective celestial body, which are continuously updated by JavaScript and Python code. These languages can access the structs as if they were objects native to them. They do so in order to compute the new position and velocity of each body based on the current values they can read from the structs and store these new values back. The distribution of tasks to languages in our multi-language benchmarks is arbitrary. As our goal in these implementations was to stress language interactions

[5]The code of these benchmarks is available at https://github.com/jkreindl/taint-benchmarks.

of various kinds (as discussed in Section 3.1) and in both directions, this distribution differs between benchmarks.

In our multi-language NBody benchmark, taint needs to be propagated correctly across C, JavaScript and Python code as well as across expressions, object properties and function calls. The flow of tainted data between languages in the benchmark is also summarized in Figure 6.

In C code. When the celestial bodies are initialized in C code, their position values are tainted. These tainted values are stored in memory allocated by the C code, which is executed by Sulong in managed mode. A pointer to this allocated memory is passed to the JavaScript code. Using GraalVM's polyglot API (and Sulong's special support for this) the JavaScript code can interact with this pointer as if the referenced object were an array of JavaScript objects providing the same properties as the C structs. Note that, while the structs partially contain tainted values, the pointer that is passed to JavaScript is not tainted itself. However, the JavaScript code accesses these tainted position values in order to compute the distance between bodies.

In JavaScript code. In order to compute the distance between two bodies, their x, y and z coordinates are subtracted and the result is stored in local variables. When reading these coordinates from the C structs, TruffleTaint also accesses their taint labels. This taint is then propagated across the subtraction expression and stored it in local variables. These local variables are then read again and passed as function arguments to the Python functions computing new position and velocity values. This time, the arguments passed in a cross-language call are tainted.

In Python code. The Python code receives the tainted distance value, as well as the respective celestial bodies. The same mechanism that enables JavaScript code to treat these structs as JavaScript objects also enables Python code to treat them as Python objects. The new position and velocity values are computed based on tainted function arguments, and are therefore tainted themselves.

The benchmark harness verifies that taint is propagated correctly when our multi-language NBody benchmark is executed. It introduces taint into the benchmark by tainting the initial position values of the celestial bodies in C code, while the initial velocity values are left untainted. After this initialization, the benchmark only writes to the fields containing the velocity values in Python code. If any part of the aforementioned taint propagation were to drop taint, the velocity values would be untainted after the benchmark completes. Furthermore, if the taint propagation failed, the final position values would not be tainted anymore, even though they also always depend on the tainted original position values. The benchmark harness considers the benchmark to have failed if the benchmark produces an incorrect result, if the final position values are not tainted anymore, and if the final velocity values are not tainted. In doing so, it checks that the taint propagation did not incorrectly interfere with the program semantics, that the taint propagation did propagate the taint labels according to the present data dependencies, and that it did not mistakenly drop taint. Since TruffleTaint executes the benchmark successfully, we see that it correctly propagates taint between (1) a read access to memory allocated by C code performed in JavaScript, (2) expressions and local variables in JavaScript and Python code, (3) a function call between JavaScript and Python code, (4) a write access to memory allocated by C performed in Python code and (4) a memory read performed in C code. Our other multi-language benchmarks similarly introduce taint to data that is propagated throughout the benchmark and verify that at the end of each benchmark all data exhibits the appropriate taint labels.

4.2 Multi-Language Taint Propagation Benchmarks and Tests

In addition to NBody, we also implemented several other benchmarks of the Computer Language Benchmarks game to propagate taint in multiple languages. More specifically, we selected the *BinaryTrees*, *FannkuchRedux*, *Fasta*, *Mandelbrot*, *NBody*, and *SpectralNorm* benchmarks. Additionally, we implemented 3 separate versions of the NBody benchmark, which each exercise different styles of language interactions. We implemented each of these benchmarks in a combination of C and JavaScript code, a combination of C and Python code, a combination of all three languages, as well as in each of these languages individually. We selected these combinations of languages since they commonly interact in real-world programs. In node.js, programs are commonly implemented in JavaScript, but make heavy use of node.js' API which is largely implemented in C++. Similarly, Python's foreign function interface is often used in popular Python libraries, such as *numpy*. TruffleTaint is able to execute all our benchmarks correctly, both in that taint labels are propagated correctly and the benchmarks produce the correct values.

Our multi-language benchmarks each exercise various kinds of language interaction. In some benchmarks, tainted data is used in function calls. In others, tainted data is stored in objects. Some benchmarks exercise their workload in one language, but use objects of another language to store tainted values in. TruffleTaint supports all constellations of language interactions.

We originally implemented taint propagation only for C and JavaScript, but extending the propagation support to Python required only to implement taint nodes for the Python builtin functions used in our benchmarks. Since the taint propagation logic for the language-agnostic tags was already implemented, we only needed to annotate the nodes provided by the Python runtime with these tags. Besides these necessarily language-specific additions, no further additions to the taint agent were required to execute our benchmarks implemented in C and Python. Since Python is a dynamically typed language, it even supported storing boxed values in its data structures out of the box, which was also the case for JavaScript. This ease of adding support for a new language illustrates the effectiveness of TruffleTaint's propagation capabilities and the suitability of the default propagation semantics it defines.

We also implemented a number of language-specific unit tests to test TruffleTaint's functionality and verify its semantically correct taint propagation. While these tests are limited in number and make no claim to cover the entire semantics of any of these languages, they do exercise the most elementary features of these languages as well as all those language features used in our benchmarks. These tests each use TruffleTaint's reference taint analysis application's language-specific builtins to (1) taint a value, (2) exercise a syntactic feature of the tested language using that value in any possible way, and (3) check that any values used in or created by this exercise are appropriately tainted or not tainted. Each syntactic feature is also exercised with multiple types of values which it can be used with. For example, LLVM IR binary operations are tested with both integer and floating point types of various bit widths.

4.3 Performance Evaluation

TruffleTaint imposes little slowdown when no taint is introduced and is able to propagate taint with non-prohibitive slowdown. Table 2 states the average slowdown TruffleTaint causes for each of our benchmarks both when taint is actually introduced in a benchmark run and when it is not. The slowdown is computed relative to the baseline of uninstrumented execution and is an average of 10 benchmark runs after appropriate warmup iterations.

The first number shown in Table 2 for each benchmark is the slowdown caused by instrumenting the benchmark to propagate taint, but not actually introducing taint. As tainted data often reaches only few parts of a program [50], we specifically optimized TruffleTaint's implementation for this case, and plan to extend these optimizations in the future. Our optimizations are proven effective for this case, as some benchmarks exhibit almost no overhead. These numbers rival even self-described fast taint propagation platforms such as libDFT [36] and Decaf++ [26]. The speedup exhibited by some benchmarks comes from the fact that certain instrumentations coincidentally lead GraalVM's optimizing JIT compiler to make better optimization choices, an effect that can be observed also for other Truffle instruments [61].

The second number shown in Table 2 for each benchmark is the slowdown caused by actually propagating taint. Depending on the number and kind of taint propagation events, this slowdown ranges between 5% and $\tilde{4}$0x. This slowdown is partly caused by implementation choices in the taint propagation and runtime support preventing GraalVM's JIT compiler from performing certain optimizations. We plan to address these issues in future work.

Overall, the benchmarks show that TruffleTaint achieves multi-language taint propagation with non-prohibitive execution time overhead. While dynamic taint analysis is often associated with significant slowdown [43, 55], TruffleTaint limits this slowdown to code that actually operates on tainted data. When no taint is introduced, TruffleTaint exhibits better performance than state-of-the-art platforms in some cases. While highly-optimized dynamic taint analysis platforms such as libDFT [36] may exhibit less slowdown when taint is introduced, TruffleTaint's performance is still in range of other taint analysis platforms such as Dytan [23], which exhibits up to 50x slowdown.

5 FUTURE WORK

TruffleTaint is currently a prototype platform for dynamic taint analysis with opportunities for future work in several directions. We are developing TruffleTaint as a platform for our research in the field of dynamic taint analysis. As part of this research we plan to improve TruffleTaint with respect to its performance and taint propagation capability, and to evaluate possible application areas.

We see significant potential in leveraging GraalVM's dynamic JIT compiler and its speculative optimization capabilities to optimize taint propagation. Using this approach, we were already able to significantly reduce TruffleTaint's run-time overhead for cases where no taint is introduced. We plan to devise new optimization strategies for taint propagation that similarly benefit code in which taint is propagated. In addition to optimizing TruffleTaint's propagation performance, we also plan to evaluate it in more detail using larger and more diverse benchmark applications. Most of the benchmarks we presented so far are CPU-bound, but we intend to also evaluate IO-bound workloads such as file compression.

TruffleTaint currently supports the most common instructions of LLVM IR as well as basic features of JavaScript and Python. Missing functionality includes mostly builtins, but also features related to asynchronous execution which may require more specific instrumentation. Furthermore, certain kinds of loops and meta-programming of the dynamic languages are currently not supported. In general, our plan is to extend TruffleTaint's language support as required by the applications and benchmarks we will use to evaluate TruffleTaint in the future. Especially with regard to builtins this means to focus on the most commonly used ones, rather than to strive for completeness.

We also plan to apply TruffleTaint in practice. As we stated in Section 2.2, dynamic taint analysis has many applications. We would like to apply such existing applications to a multi-language environment. For example, tracing program inputs to aid delta debugging, such as in Penumbra [24], may be useful in identifying bugs in native extensions. Similarly, applying dynamic taint analysis to debugging would allow for implementing value-specific stepping strategies and breakpoints. We have also previously noted that language boundaries could be used as a means to defeat a vulnerability detection taint analysis, while TruffleTaint could avoid such exploits. *Data provenance* is another possible application area. For example, a system based on dynamic taint analysis to trace sensible information in multi-language programs, perhaps even multi-host systems and across database interaction, would have applications for auditing in enterprise systems as well as to prevent the leakage of said data, e.g., personally identifiable information. Program comprehension is another interesting area, where dynamic taint analysis could, e.g., be used to collect information about how often specific values cross the language boundary or where each part of a value is often used, which may be of use to identify opportunities for refactoring to improve performance. Another useful application would be to infer data-flow specifications of libraries in order to aid static taint analysis, which Taser [58], a dynamic taint analysis implemented on top of GraalVM's JavaScript-specific *NodeProf* [59] analysis framework, has shown to be feasible. Multi-language dynamic taint analysis may also guide a fuzzer such as Angora [22] more effectively for applications that use multiple languages.

Verifying correctness of taint propagation is a challenge for all languages supported by TruffleTaint. Taint analyses tend to use analysis-specific propagation semantics and usually do not provide a test suite that TruffleTaint could be tested against. These analyses usually argue for their correctness and coverage of language semantics by presenting results such as found program vulnerabilities or correctly reverse-engineered protocol format. One approach to verify correctness of TruffleTaint could be to reimplement such previous applications of dynamic taint analysis on top of TruffleTaint and reproduce the results of these applications.

Table 2: Average slowdown of execution with taint propagation compared to uninstrumented execution.

Benchmark	Slowdown for language combination (without tainted data / with tainted data)					
	C	JS	Python	C & JS	C & Python	C & JS & Python
BinaryTrees	1.16x/1.79x	1.06x/1.28x	1.39x/1.93x	1.15x/1.84x	1.16x/1.75x	1.16x/1.96x
FannkuchRedux	1.00x/3.92x	1.06x/3.32x	1.41x/6.08x	1.29x/3.44x	1.00x/10.72x	1.19x/1.74x
Fasta	1.01x/2.52x	1.11x/1.31x	1.13x/1.28x	0.85x/1.59x	1.06x/1.27x	0.99x/1.05x
Mandelbrot	1.00x/15.44x	0.99x/11.96x	1.20x/40.37x	1.00x/18.94x	1.00x/19.36x	1.00x/20.49x
NBody 1	1.02x/19.36x	0.94x/11.81x	1.57x/17.96x	1.27x/1.22x	1.00x/1.18x	1.76x/3.74x
NBody 2	0.88x/5.74x	1.20x/7.30x	2.44x/16.00x	0.80x/1.16x	1.01x/1.23x	1.00x/1.20x
NBody 3	1.22x/11.04x	1.72x/8.08x	2.60x/16.68x	1.06x/15.72x	1.30x/3.10x	1.07x/5.49x
SpectralNorm	1.00x/6.46x	1.00x/2.23x	1.08x/2.06x	1.00x/7.73x	1.00x/4.77x	1.12x/1.41x

6 RELATED WORK

Previous work on dynamic taint analysis is large in quantity but has so far not directly addressed language interactions. Publications mostly focus on specific programming languages, runtime environments or binary analysis platforms. Frameworks for implementing taint analysis applications, while more generic, are also not geared towards supporting language interactions.

Language-specific dynamic taint analysis systems typically treat data flow outside of the targeted programming language as a black box. For example, for language-embeddings such as node.js, the instrumentation typically only supports the dynamic language code [35, 58]. Some of these systems, e.g. Karim et al. [35], allow for manual specification of the taint flow in the other language. Such specifications can even take the form of complex models that take into account dynamic data flow to some degree [33]. However, these models have to be provided by users and cannot be guaranteed to match the executed code. TruffleTaint, in contrast, does not require such specifications and can instead observe the actual data flow.

On top of Graal.js, Staicu et al. [58] have implemented dynamic taint analysis for JavaScript applications. Azadmanesh at al. [17] have extracted data dependencies for offline analysis tools using Truffle instrumentation. In contrast to these approaches, TruffleTaint, supports taint propagation across multiple languages as well as adaptable propagation semantics, taint sources and taint sinks.

Multi-language taint analysis can be achieved by compiling all involved code to the same intermediate representation or to native code and applying a dynamic taint analysis which targets that representation. For example, the *LLVM Dataflow Sanitizer* (DFSan) [9], which is part of the LLVM tool-chain [40], instruments LLVM IR to propagate user-defined taint labels together with data. Although DFSan's API is limited to languages of the C family, it can in principle support any language also supported by LLVM. Similarly, *Phosphor* [18] can instrument Java bytecode to perform taint propagation. Many languages can be compiled to LLVM IR or Java bytecode [8, 15, 42, 51]. This approach to multi-language taint tracking is similar to TruffleTaint in that all code is executed using the same analysis platform. However, in contrast to this approach, TruffleTaint allows for language-level instrumentation and can target specific language features in its taint sources, sinks and propagation policy. Additionally, the reduced number and lower granularity of instrumentation events compared to low-level instrumentation reduces excessive taint spread and run-time overhead.

Another approach to multi-language dynamic taint analysis is to execute the runtime environments of all involved languages on a binary analysis platform. DECAF++ [26] and libDFT [36] use such platforms to implement dynamic taint analysis for native code. While libDFT is an application-level analysis platform, DECAF++ is a whole-system analysis platform based on the QEMU emulator. However, while this approach can support even applications whose source code is not available, it does not allow for language-specific propagation semantics without being tailored towards a specific language runtime. In contrast, TruffleTaint allows for language-specific instrumentation by design and could be extended to support native code using a suitable Truffle runtime, e.g., by Pekarek [48].

7 CONCLUSION

In this paper we presented TruffleTaint, a platform for dynamic taint analysis in and across multiple languages. TruffleTaint employs language-agnostic tainting techniques which enable it to propagate taint labels in multiple languages even for data that crosses language boundaries. Furthermore, we introduced a language-agnostic core taint propagation semantics which can be reused when adding support for another programming language. Moreover, TruffleTaint can be extended to support additional programming languages and taint analysis applications. TruffleTaint is currently capable of propagating taint across code implemented in C, JavaScript and Python. Evaluation using well-known benchmarks from the Computer Language Benchmarks Game has shown that TruffleTaint exhibits little run-time overhead in code that is not reached by tainted data. In the future, we plan to improve TruffleTaint's language support, taint propagation functionality and performance as well as to apply it in practice by implementing concrete taint analysis applications to support multiple programming languages. We believe that TruffleTaint has the potential to become a premier platform for research on multi-language dynamic taint analysis.

ACKNOWLEDGMENTS

We thank all members of the Virtual Machine Research Group and the Program Analysis Group at Oracle Labs. Oracle, Java, GraalVM, and HotSpot are trademarks or registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. We also thank all researchers at the Johannes Kepler University Linz's Institute for System Software for their support of and valuable feedback on our work.

Table 3: Builtins provided by the TruffleTaint reference instrument to represent taint sources and sinks.

Action	C/C++ Function	JavaScript/Python Function
Mark value as tainted	`__truffletaint_add_`*<type>*`((`*<type>*`) value)`	`Taint.add(value)`
Mark value as not tainted	`__truffletaint_remove_`*<type>*`((`*<type>*`) value)`	`Taint.remove(value)`
Check if value is tainted	`__truffletaint_check_`*<type>*`((`*<type>*`) value)`	`Taint.check(value)`
Throw error if value is not tainted	`__truffletaint_assert_`*<type>*`((`*<type>*`) value)`	`Taint.assertTainted(value)`
Throw error if value is tainted	`__truffletaint_assertnot_`*<type>*`((`*<type>*`) value)`	`Taint.assertNotTainted(value)`

A TRUFFLETAINT REFERENCE INSTRUMENT

As stated in Section 3.3, TruffleTaint provides a reference implementation of a taint analysis application. This reference instrument provides the user with builtins to facilitate the addition, removal, and querying of taint labels for runtime values. In C/C++ code these builtins can be accessed by calling intrinsified functions, which are declared in a header file provided by the reference instrument. For JavaScript and Python code, the reference instrument provides a special module, named *Taint*, which exposes these builtins. Table 3 shows the available builtins and their function.

REFERENCES

[1] 2020. Ballerina Taint Checking. https://ballerina.io/learn/by-example/taint-checking.html. Accessed: 2020-07-29.
[2] 2020. Compiling Native Projects via the GraalVM LLVM Toolchain. https://medium.com/graalvm/graalvm-llvm-toolchain-f606f995bf. Accessed: 2020-05-20.
[3] 2020. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Accessed: 2020-04-24.
[4] 2020. CVE Entry for Hijacking of RubyGem rest-client. https://nvd.nist.gov/vuln/detail/CVE-2019-15224. Accessed: 2020-04-20.
[5] 2020. CVE Entry for Hijacking of RubyGem strong-password. https://nvd.nist.gov/vuln/detail/CVE-2019-13354. Accessed: 2020-04-20.
[6] 2020. GraalVM. https://www.graalvm.org. Accessed: 2020-04-21.
[7] 2020. GraalVM Polyglot Reference Manual. https://www.graalvm.org/docs/reference-manual/polyglot/. Accessed: 2020-08-11.
[8] 2020. Incomplete List of Languages with LLVM Backend. https://llvm.org/ProjectsWithLLVM/. Accessed: 2020-09-24.
[9] 2020. LLVM Data-Flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed: 2020-04-17.
[10] 2020. Node.js. http://www.nodejs.org/. Accessed: 2020-04-20.
[11] 2020. Node.js Package Manager. http://www.npmjs.com/. Accessed: 2020-04-20.
[12] 2020. Perl Taint Mode. https://perldoc.perl.org/perlsec.html. Accessed: 2020-04-17.
[13] 2020. Ruby Taint Flags. https://ruby-doc.com/docs/ProgrammingRuby/html/taint.html. Accessed: 2020-04-17.
[14] 2020. Safe and Sandboxed Execution of Native Code. https://medium.com/graalvm/safe-and-sandboxed-execution-of-native-code-f6096b35c360. Accessed: 2020-04-23.
[15] K. Ali, X. Lai, Z. Luo, O. Lhotak, J. Dolby, and F. Tip. 2019. A Study of Call Graph Construction for JVM-Hosted Languages. *IEEE Transactions on Software Engineering* (2019), 1–1.
[16] Frederico Araujo and Kevin W. Hamlen. 2015. Compiler-instrumented, Dynamic Secret-Redaction of Legacy Processes for Attacker Deception. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 145–159. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/araujo
[17] 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
[18] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA, 2014-10-15) *(OOPSLA '14)*. Association for Computing Machinery, 83–101. https://doi.org/10.1145/2660193.2660212
[19] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. 2007. Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 317–329. https://doi.org/10.1145/1315245.1315286
[20] 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
[21] 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
[22] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046
[23] 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
[24] 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
[25] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820
[26] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2019. 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, 31–45. https://www.usenix.org/conference/raid2019/presentation/davanian
[27] Rich Dill. 2018. *Automating Mobile Device File Format Analysis.* Doctoral Dissertation. Airforce Institute of Technology, Air University, Wright-Patterson Air Force Base, Ohio.
[28] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Xiaodong Song. 2007. Dynamic Spyware Analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*, Jeff Chase and Srinivasan Seshan (Eds.). USENIX, 233–246. http://www.usenix.org/events/usenix07/tech/egele.html
[29] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, Lisa L. Fowler, and Murphy Mccauley. 2010. *Towards Practical Taint Tracking.*
[30] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-Based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 474–484. https://doi.org/10.1109/ICSE.2009.5070546
[31] 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
[32] W. Halfond, A. Orso, and P. Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering* 34, 1 (Jan. 2008), 65–81. https://doi.org/10.1109/TSE.2007.70748
[33] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10204)*, Matteo Maffei and Mark Ryan (Eds.). Springer, 49–70. https://doi.org/10.1007/978-3-662-54455-6_3

[34] Vivek Jain, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2018. TIFF: Using Input Type Inference To Improve Fuzzing. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. ACM, New York, NY, USA, 505–517. https://doi.org/10.1145/3274694.3274746

[35] 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

[36] 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

[37] 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

[38] 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 2019*. ACM Press, Athens, Greece, 85–94. https://doi.org/10.1145/3357390.3361028

[39] Jacob Kreindl, Manuel Rigger, and Hanspeter Mössenböck. 2018. Debugging Native Extensions of Dynamic Languages. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang'18)*. ACM, Linz, Austria, 12:1–12:7. https://doi.org/10.1145/3237009.3237017

[40] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[41] 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

[42] Wing Hang Li, David R. White, and Jeremy Singer. 2013. JVM-Hosted Languages: They Talk the Talk, but Do They Walk the Walk?. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Stuttgart, Germany) *(PPPJ '13)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/2500828.2500838

[43] 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/

[44] Jiang Ming, Meng Pan, and Debin Gao. 2012. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Information Security and Cryptology - ICISC 2012 - 15th International Conference, Seoul, Korea, November 28-30, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 7839)*, Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.). Springer, 92–109. https://doi.org/10.1007/978-3-642-37682-5_8

[45] Shashidhar Mysore, Bita Mazloom, Banit Agrawal, and Timothy Sherwood. 2008. Understanding and Visualizing Full Systems with Data Flow Tomography. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. ACM, New York, NY, USA, 211–221. https://doi.org/10.1145/1346281.1346308

[46] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*. The Internet Society. https://www.ndss-symposium.org/ndss2005/dynamic-taint-analysis-automatic-detection-analysis-and-signaturegeneration-exploits-commodity/

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

[48] Daniel Alexander Pekarek. 2019. A Truffle-based Interpreter for x86 Binary Code. https://resolver.obvsg.at/urn:nbn:at:at-ubl:1-27719

[49] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. 2006. Argos: An Emulator for Fingerprinting Zero-Day Attacks for Advertised Honeypots with Automatic Signature Generation. In *Proceedings of the 2006 EuroSys Conference on - EuroSys '06*. ACM Press, Leuven, Belgium, 15. https://doi.org/10.1145/1217935.1217938

[50] 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

[51] 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* (Amsterdam, Netherlands) *(VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[52] Manuel Rigger, Stefan Marr, Bram Adams, and Hanspeter Mössenböck. 2019. Understanding GCC Builtins to Develop Better Tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 74–85. https://doi.org/10.1145/3338906.3338907

[53] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of X86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA) *(VEE '18)*. Association for Computing Machinery, New York, NY, USA, 84–99. https://doi.org/10.1145/3186411.3186418

[54] Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/

[55] 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 *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331. https://doi.org/10.1109/SP.2010.26

[56] R. Sekar. 2009. An Efficient Black-box Technique for Defeating Web Application Attacks. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society. https://www.ndss-symposium.org/ndss2009/an-efficient-black-box-technique-for-defeating-web-application-attacks/

[57] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. 2013. Diglossia: Detecting Code Injection Attacks with Precision and Efficiency. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. ACM, New York, NY, USA, 1181–1192. https://doi.org/10.1145/2508859.2516696

[58] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*.

[59] 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* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/3178372.3179527

[60] the npm blog. 2020. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident. Accessed: 2020-04-20.

[61] 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

[62] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society. https://www.ndss-symposium.org/ndss2007/cross-site-scripting-prevention-dynamic-data-tainting-and-static-analysis/

[63] 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

[64] Wenmin Xiao, Jianhua Sun, Hao Chen, and Xianghua Xu. 2014. Preventing Client Side XSS with Rewrite Based Dynamic Information Flow. In *Sixth International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2014, Beijing, China, July 13-15, 2014*, Hong Shen, Yingpeng Sang, and Hui Tian (Eds.). IEEE Computer Society, 238–243. https://doi.org/10.1109/PAAP.2014.10

[65] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2013. AUTOVAC: Automatically Extracting System Resource Constraints and Generating Vaccines for Malware Immunization. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 112–123. https://doi.org/10.1109/ICDCS.2013.69

[66] Hongfa Xue, Guru Venkataramani, and Tian Lan. 2018. Clone-Slicer: Detecting Domain Specific Binary Code Clones through Program Slicing. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*. ACM. https://doi.org/10.1145/3273045.3273047

[67] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-Wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

[68] Jinfeng Yuan, Weizhong Qiang, Hai Jin, and Deqing Zou. 2014. CloudTaint: An Elastic Taint Tracking Framework for Malware Detection in the Cloud. *The Journal of Supercomputing* 70, 3 (Dec. 2014), 1433–1450. https://doi.org/10.1007/s11227-014-1235-5

[69] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2010. Neon: System Support for Derived Data Management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '10)*. ACM, New York, NY, USA, 63–74. https://doi.org/10.1145/1735997.1736008

[70] 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

[71] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman