Low-Overhead Multi-language Dynamic Taint Analysis on Managed Runtimes through Speculative Optimization

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

Daniele Bonetta
Oracle Labs
Netherlands
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 (DTA) is a popular program analysis technique with applications to diverse fields such as software vulnerability detection and reverse engineering. It consists of marking sensitive data as *tainted* and tracking its propagation at runtime. While DTA has been implemented on top of many different analysis platforms, these implementations generally incur significant slowdown from taint propagation. Since a purely dynamic analysis cannot predict which instructions will operate on tainted values at runtime, programs have to be fully instrumented for taint propagation even when they never actually observe tainted values. We propose leveraging speculative optimizations to reduce slowdown on the peak performance of programs instrumented for DTA on a managed runtime capable of dynamic compilation.

In this paper, we investigate how speculative optimizations can reduce the peak performance impact of taint propagation on programs executed on a managed runtime. We also explain how a managed runtime can implement DTA to be amenable to such optimizations. We implemented our ideas in *TruffleTaint*, a DTA platform which supports both dynamic languages like JavaScript and languages like C and C++ which are typically compiled statically. We evaluated TruffleTaint on several benchmarks from the popular *Computer Language Benchmarks Game* and *SPECint 2017* benchmark suites. Our evaluation shows that TruffleTaint is often able to avoid slowdown entirely when programs do not operate on tainted data, and that it exhibits slowdown of on average ~2.10x and up to ~5.52x when they do, which is comparable to state-of-the-art taint analysis platforms optimized for performance.

CCS Concepts: • **Security and privacy** → **Information flow control**; • **Software and its engineering** → **Software performance**; **Runtime environments**; • **General and reference** → *Performance.*

Keywords: Dynamic Taint Analysis, GraalVM, C/C++, JavaScript, Performance

ACM Reference Format:
Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2021. Low-Overhead Multi-language Dynamic Taint Analysis on Managed Runtimes through Speculative Optimization. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '21), September 29–30, 2021, Münster, Germany.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3475738.3480939

1 Introduction

Dynamic taint analysis [46] (DTA), often also referred to as *dynamic taint tracking*, is a popular program analysis technique in which sensitive data is marked as *tainted* and the propagation of this tainted data and any data derived from it is tracked while the analyzed program is executed. Applications of DTA cover many fields including software vulnerability detection [15, 36, 42], software testing [16, 17], debugging [20] and reverse engineering [21, 48]. DTA has been implemented for programming languages such as JavaScript [26, 31, 33] and for native code [12, 14, 15, 19, 22, 32, 44] compiled from, e.g., C/C++ programs. However, DTA commonly imposes a significant run-time overhead on programs instrumented for taint propagation [39, 46], with even taint tracking engines optimized for performance incurring slowdown of up to several orders of magnitude. Our work aims to address this issue.

Previous approaches to address this run-time overhead often try to avoid run-time taint propagation. Some approaches switch between instrumented and uninstrumented versions of the same program depending on whether the currently executing code needs to propagate taint[22, 28, 44]. These approaches exploit the observation that tainted data often only spreads to a limited part of the program [44], but incur overhead from continuously checking at runtime which version to execute and fail to reduce slowdown for actually occurring taint propagation. Other approaches apply static analysis to predetermine the flow of tainted data where possible and thus reduce the amount of instructions required for run-time taint propagation [30, 52]. However, since it lacks access to run-time information, static analysis cannot remove taint propagation for instructions that could theoretically be reached by tainted data, but in practice are not. Other approaches that optimize taint propagation instructions to leverage specific characteristics of the underlying platform [14, 32] are by design platform-specific and cannot always be applied to other platforms. We propose the use of speculative optimization to gain the benefits of all these approaches in a managed runtime without suffering from their drawbacks.

Managed runtimes use run-time profiling to optimize and dynamically compile frequently executed code. They consist of an interpreter, which collects profiling information, and a dynamic compiler which can use this information for optimization. We propose leveraging both speculative and non-speculative optimization to reduce the slowdown which programs incur from taint propagation. The main insight behind this approach is that if we know at compile time whether an input of a particular statement is tainted or not, common optimizations such as *inlining* [43] and *partial evaluation* [50] can be applied to simplify or optimize away the corresponding code performing taint propagation. Our approach leverages this insight by directing the compiler to perform these optimizations *speculatively*, i.e., under assumptions for each input whether its value is tainted or not. Such optimized code contains run-time checks whether these assumptions are still valid. When they become invalid, it is deoptimized and recompiled with the new knowledge. The benefits of our approach increase with the size of the compilation unit since run-time assumption checks can be merged if they appear in multiple places or optimized away when, e.g., one assumption trivially holds if another does.

We identified speculative assumptions that can be made based on profiling information and are suitable for enabling the simplification or removal of taint propagation code by applying common compiler optimizations. Additionally, we propose strategies for efficiently verifying these assumptions at runtime. These strategies involve optimizing how run-time data structures like objects and arrays of dynamic languages, native allocations of lower-level languages and local or global variables, can store taint labels of values they contain. Our optimization strategies are language-independent and can be applied to any managed runtime capable of dynamic compilation and speculative optimization. They can support both dynamic languages like JavaScript and languages like C and C++ that are typically compiled statically. They only require that both the code implementing taint propagation and the instrumented program are expressed in a program representation which the compiler can perform optimizations on.

We implemented our optimization strategies in *TruffleTaint* [34], a dynamic taint analysis engine based on the GraalVM platform [51], which supports tracking taint both in and between code of multiple languages such as JavaScript, C and C++. We additionally optimized the techniques TruffleTaint uses for propagating taint labels to be amenable to optimization by GraalVM's dynamic JIT compiler. We evaluated TruffleTaint using both JavaScript and C/C++ implementations of benchmarks from the *Computer Language Benchmarks Game* [1], the SPECint 2017 benchmarks suite [9] and a hand-crafted multi-language benchmark implementing a real-world workload for taint tracking based on the HTTP header parser of the Node.js framework [7]. Our evaluation shows that TruffleTaint can avoid any impact of dynamic taint analysis on peak performance of programs it executes if they do not operate on tainted data, and that for some benchmarks it can also avoid such impact even when they do operate on tainted data. The evaluation further shows that TruffleTaint exhibits an average slowdown of ~2.10x for our benchmarks, with the highest observed slowdown being ~5.52x. Though this slowdown is significant, these results are still in range of other state-of-the-art platforms for general, purely dynamic taint analysis optimized for performance.

This paper makes the following contributions:

- We identify opportunities for using language-agnostic speculative optimization techniques to avoid or reduce the overhead of dynamic taint analysis on peak performance of analyzed applications. (Section 3)
- We describe how run-time data structures of multiple languages can store taint labels of contained values in a manner amenable to these speculative optimizations. (Section 3)
- We provide an implementation of our ideas in a dynamic taint analysis platform supporting multiple languages, including C/C++ and JavaScript. (Section 4)
- We evaluate our proposed optimization techniques and their implementation using several well-known benchmark programs and show that they can avoid peak performance slowdown incurred from redundant taint propagation. (Section 5)

2 Background

In dynamic taint analysis, all values that originate from so-called *taint sources* (e.g., functions that read data from the file system or otherwise produce data of interest to the analysis) are marked as *tainted* by attaching a *taint label* to them. When a tainted value is used as input to an expression, its taint label is *propagated* to the result of that expression. To this end, a *propagation semantics* defines for each kind of expression supported by the targeted programming language, whether a value that is written or produced by such an expression is tainted depending on whether its input values were. For example, propagation semantics commonly define that the sum of an addition is tainted if any of its operands are. A *propagation technique* defines how a taint label is attached to a value. In addition to taint sources, an implementation of dynamic taint analysis also defines *taint sinks*, i.e., program locations at which analysis-defined actions should be taken if they observe tainted inputs.

2.1 GraalVM

The GraalVM platform is an ecosystem for compiling, executing [51] and instrumenting [23] programs of various programming languages. GraalVM is based on the Java Virtual Machine. At its heart is the Truffle [11] framework for implementing language runtimes that are based on abstract syntax tree (AST) interpretation. To this end, Truffle defines the interfaces for AST nodes, which can be implemented by individual runtimes with concrete language semantics. A Truffle-based language runtime parses code of its targeted language into such a *Truffle AST* for execution. GraalVM's included JIT compiler has special knowledge of Truffle ASTs which enables it to specifically optimize them in order to produce more efficient machine code. Truffle also provides API for language implementers to provide speculative assumptions to this compiler. This API includes setting and manually invalidating speculative assumptions and for profiling run-time values such as condition checks. Furthermore, Truffle AST nodes can *specialize* themselves to observed runtime conditions, which means that they can replace their implementation with one specifically optimized for these conditions. Specialized nodes check at runtime whether these conditions still apply and, if not, deoptimize the compiled code and rewrite themselves to a more generic version. Truffle AST nodes can also dispatch between multiple active specializations of the same node at runtime.

2.2 TruffleTaint

All optimization techniques discussed in this paper have been implemented targeting a real-world taint tracking system called TruffleTaint. TruffleTaint [34] is implemented on top of Truffle's framework for instrumenting Truffle ASTs. This framework enables TruffleTaint to insert *Taint Nodes*, which are special AST nodes that implement taint propagation according to a user-defined propagation semantics, into the AST. For example, Figure 1 depicts a Truffle AST for a statement which adds 1 to the value of a local variable x and stores the result in the local variable y. The AST nodes

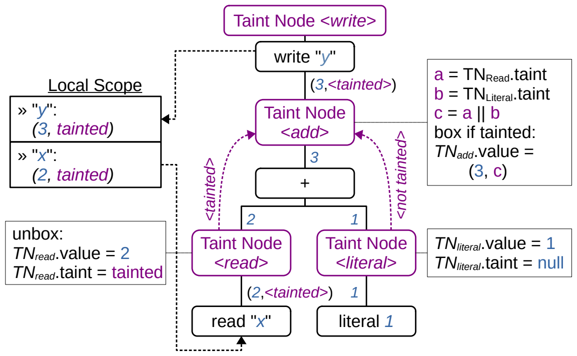

Figure 1. Taint propagation with TruffleTaint for the statement `var y = x + 1`. The current value of x is 2 and it is tainted. Edges between AST nodes are annotated with the values the respective child nodes produce. Taint nodes and taint labels are depicted in purple, other nodes belong to the Truffle AST for the statement.

depicted with a black border correspond to this statement. They consist of a node to *read* the current value of x, a node which returns the *literal* 1, a node for adding these values, and a node to *write* the resulting sum to y. Each of these nodes is the only child of a taint node, which intercepts the values returned by the respective nodes, determines the taint labels of these values according to the propagation semantics for the action implemented by the node, and performs the appropriate actions to propagate these taint labels.

TruffleTaint supports two propagation techniques, which can be seen in Figure 1. The first technique is applied to *intermediate expressions*, which are expressions that only compute values from their inputs rather than storing them into memory or returning them to another function. A taint node that provides a value for such an expression stores that value's taint label into a so-called *taint slot*, i.e., a container within the node. The taint node instrumenting the intermediate expression loads the taint labels of its input values from there. In Figure 1, both TN_{read}, i.e., the $TaintNode$ instrumenting the *read* operation, and $TN_{literal}$ propagate the taint labels of the values they produce by writing them to such taint slots. The second propagation technique is applied when a taint node produces an input value for an expression which would make that value available to other statements by writing it to a scoped variable, storing it into allocated memory such as an object or array, returning it to a calling function, using it as argument to a function call, or throwing it as an exception object. When such an input value is tainted, that taint node replaces that input value with a *boxed value*, which is a container storing both the original value and its taint label, so that they can be stored together in memory. In Figure 1, boxed values are represented as tuples of value and taint label. Taint nodes that instrument the corresponding read operations may then *unbox* these values again. In Figure 1, the value of x is tainted, and so the local variable stores it as a boxed value with a taint label. Since TN_{read} produces an input value for an intermediate expression, it intercepts this boxed value, unboxes it, and stores the corresponding taint label into its taint slot before returning the unboxed value. TN_{add} loads that taint label from this taint slot and determines the sum to be tainted since one of its operands was. Since TN_{add} provides an input to an expression that writes to memory, it propagates this taint label by placing it into a boxed value and returning that boxed value instead. That boxed value is then stored into the frame. While our previous work [34] introduced a naive implementation of these techniques and evaluated them on a functional level, in this paper we describe how we optimize this implementation for peak performance.

TruffleTaint currently supports GraalVM's Truffle-based JavaScript and LLVM language runtimes. The GraalVM JavaScript runtime is an ECMAScript-compliant engine for running JavaScript and Node.js programs [3]. The GraalVM LLVM runtime supports the execution of LLVM-based languages such as C and C++ [4]. TruffleTaint only supports its *managed mode*, in which all memory allocated by user programs is backed by Java objects on the managed heap rather than by native memory [8].

2.3 Speculative Optimization

Speculative optimization describes the concept of optimizing code during dynamic compilation under *speculative assumptions* that the values the code operates on exhibit certain properties. The goal behind these assumptions is to enable opportunities for applying further, non-speculative optimizations. Since these assumptions cannot be proven already at JIT compile time, the compiled code needs to include instructions for checking whether the actual values observed at runtime do, in fact, exhibit these properties. If such a check fails and the corresponding assumption is thus disproven, the compiled code is deoptimized [51] and the same code can be recompiled under the new assumptions. Speculative optimization is only possible in a runtime capable of *dynamic compilation*, i.e., of deoptimizing and recompiling code when run-time conditions change. Managed runtimes, as we define them, further include an interpreter capable of collecting profiling information.

Value profiling [41] is a speculative optimization technique which consists of observing whether a particular expression always returns the same value during interpreted execution. If so, the compiler assumes for further optimizations that the expression always returns that value. To ensure correct execution, code compiled from this expression includes a runtime check which invalidates that profile and deoptimizes that code should the expression ever return another value. If lower and upper bound of a counted loop are profiled to be known constants, that loop may be *unrolled* [37] in order to avoid checking the loop condition at runtime. Similarly, if a conditional expression is profiled to be a constant value, the branch that is not taken need not be compiled and any side-effects of that branch need not be considered in further optimizations.

Specialization, as explained for Truffle AST nodes in Section 2.1, is another speculative optimization technique.

Speculative assumptions can create additional opportunities for applying common compiler optimizations such as *partial evaluation* [50], i.e., to partially evaluate expressions at compile time if at least one of their input values is known. Partial evaluation in turn can create opportunities for applying *constant propagation* [40], i.e., replacing accesses to a variable known to contain a constant value with that value. Together, such optimizations can significantly reduce the amount of code to be executed. Our approach is to use speculative assumptions to leverage this effect for TruffleTaint's expression-level instrumentation code for run-time taint tracking.

To give an example for our approach, consider the statement illustrated in Figure 1 which reads a variable x, increments its value, and writes the result to a local variable y. Without using static analysis, each expression in the program under analysis needs to be instrumented to perform taint propagation at runtime. This propagation involves checking whether any of the expression's inputs are tainted, deciding based on that whether its output value is also tainted, and, if so, propagating that taint label to the instrumentation of its parent expression. Expressions that read from or write to memory also need to maintain *shadow storage*, i.e., a record of which regions of memory currently contain tainted values. For our example, instrumentation could intuitively be simplified to just copying the value from the shadow storage for x to that for y. However, automating this intuition typically requires static analysis. In Section 4 we explain how we instead implement expression-level instrumentation for taint tracking such that by applying generic optimizations, a compiler can achieve the same simplification. However, speculative assumptions about the instrumented program could enable further optimizations. For example, assuming that x never holds a tainted value, maintaining or accessing shadow storage for y would not be necessary since the value of y is anyway never tainted. Any statements that read from y before any other expressions write to it could instead be optimized to consider values stored in y to be not tainted without checking the shadow storage. In Section 3 we explain how we leverage this insight.

3 Speculative Optimizations for Taint Propagation

The goal of our research is to reduce the impact of taint tracking on the peak performance of programs executed on a managed runtime. To this end we extend the instrumentation code which implements taint propagation, i.e., the *taint*

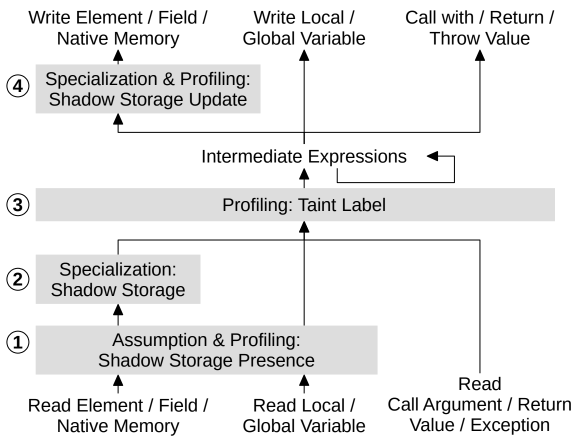
Write Element / Field / Native Memory
Write Local / Global Variable
Call with / Return / Throw Value
④ Specialization & Profiling: Shadow Storage Update
Intermediate Expressions
③ Profiling: Taint Label
② Specialization: Shadow Storage
① Assumption & Profiling: Shadow Storage Presence
Read Element / Field / Native Memory
Read Local / Global Variable
Read Call Argument / Return Value / Exception

Figure 2. Speculative optimization of taint propagation.

tracking code, to maintain speculative assumptions on how and where the instrumented program stores and propagates taint labels. These assumptions create additional opportunities for the compiler to simplify the taint tracking code by applying further optimizations to it. At runtime, compiled code checks whether these assumptions are still valid and is deoptimized when they become invalid. We thereby enable the compiler to optimize away redundant run-time taint propagation, which translates into increased peak performance. Ideally, when a program does not operate on tainted data, all taint tracking code can be optimized away and the program reaches the same peak performance as if it were not instrumented. As our evaluation in Section 5 will show, this can indeed be achieved with our approach.

Figure 2 gives an overview of the speculative optimization techniques we apply. For each memory location, such as an array or a local variable, the taint tracking code needs to maintain a corresponding shadow storage, i.e., a data structure which can store a taint label for each value stored in that location. When an expression in a statement reads from such a memory location, the taint tracking code for that expression has to load the taint label of that value from the corresponding shadow storage. Using assumptions and value profiling, we enable the compiler to speculatively optimize this access away if the memory location previously never held a tainted value (①, Section 3.1). If reading from shadow storage becomes necessary, we optimize that read operation by specializing the corresponding code to run-time properties of that shadow storage (②, Section 3.2). When an expression produces a value, the corresponding taint tracking code propagates either a taint label or a `null`-value, depending on whether that value is tainted. For any expression which produces input values for a statement, we use value profiling to determine whether these values are always tainted, never tainted, or sometimes tainted. Except in the latter case, the propagated value is treated as a constant for further optimization (③, Section 3.3). Beside memory reads, such input expressions also include function calls, function parameter reads and expressions which catch a thrown exception object. The more propagated values our optimizations can make constant for compilation, the more this code can be simplified by partial evaluation and other optimizations. When an expression writes to memory, the corresponding taint tracking code has to update shadow storage. We specialize this code to run-time properties of that shadow storage and profile whether the value to write is tainted (④, Section 3.4).

3.1 Optimized Storage for Taint Labels

As Figure 2 indicates in ①, we leverage both speculative assumptions and value profiling to optimize reading from shadow storage. To enable this approach, we create shadow storage only on-demand when a memory write expression first stores a tainted value to the corresponding memory location. When an expression reads a value from a memory location for which no shadow storage has yet been created, the corresponding taint tracking code thus knows this value to not be tainted. Using a speculative assumption or, as a fallback, value profiling we inform the compiler when this code never observes shadow storage and thus reduce the value it propagates to a constant `null` for further optimization. We also propose specialized shadow storage for various kinds of memory locations.

3.1.1 Memory Read Access Optimization.

To avoid compiled code having to check for the presence of shadow storage at runtime we maintain speculative assumptions for the compiler that none has yet been created. For scoped variables, i.e., local and global variables, we can maintain one assumption per variable since their number is known at compile time. For dynamically allocated memory locations such as arrays, objects and native allocations we instead maintain one assumption per kind of location. While these assumptions are valid, taint tracking code for expressions that read from the corresponding memory locations does not check for the existence or content of shadow storage at runtime. The compiler instead assumes this code to propagate a constant `null`-value. Also as long as such an assumption is valid, taint tracking code for an expression that writes to the corresponding (kind of) memory location checks whether it writes a tainted value and, if so, invalidates that assumption. This invalidation triggers all code compiled under the assumption to be deoptimized to ensure proper taint propagation. For programs not operating on tainted data, the compiler can often safely optimize these checks away during JIT compilation. When statements do not have tainted inputs, they typically do not produce tainted values. Consider the example in Figure 1. If the local variable `x` is assumed to not hold a tainted number, then by partially evaluating the check for whether the addition has a tainted operand, the compiler can determine that the sum is not tainted. In turn, the compiler can optimize away the check whether the value to be

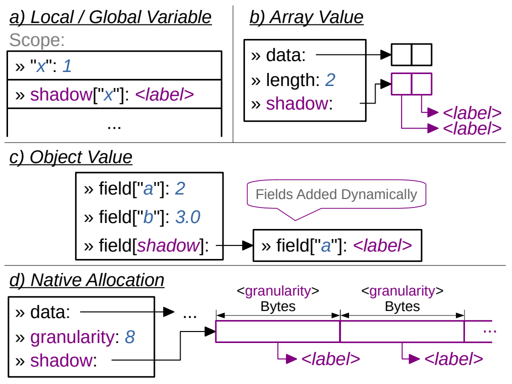

Figure 3. Taint label storage optimized for the common case of no data being tainted.

stored in y is tainted in the same manner. As a result of this optimization, the code compiled from this example need not contain any instructions for checking or propagating taint labels or for invalidating an assumption.

We maintain separate speculative assumptions for different memory locations to access shadow storage only in those parts of the program that actually operate on tainted data. For example, if a program propagates tainted values only through local variables, then only the assumptions regarding these specific variables become invalid. For expressions which read from other variables, shadow storage still does not need to be accessed.

Our evaluation in Section 5.2 shows that our taint tracking engine can often avoid deterioration of peak performance in programs that never operate on tainted data, which is solely the effect of this first optimization. To leverage this effect, a program need not lack taint sources. Consider a program which introduces tainted values only in a conditional branch. If that branch is never executed in practice, none of our speculative assumptions are invalidated. Dynamic compilation enables the program to be recompiled if that branch is eventually executed, hence our optimizations do not sacrifice correctness of dynamic taint analysis for performance.

When a speculative assumption about shadow storage for a kind of dynamically allocated memory location is invalidated, we use value profiling as a fallback optimization. By profiling whether a specific memory read expression ever observes shadow storage to access, the value propagated by the corresponding taint tracking code can still potentially be reduced to a constant `null`-value for further optimization. To enable this strategy we represent shadow storage for various kinds of dynamically allocated memory locations as shown in Figure 3. The Figure shows that we strictly separate the data structures used for representing memory allocations from the shadow storage. The former data structures each contain a field that stores a pointer to potential shadow storage. In taint tracking code, we profile for each memory read expression whether the value of this field in the memory locations it reads from is `null`. As long as it is, this code can be compiled under the assumption that it will always be, and thus that the value it propagates is a constant `null`-value as well. When such compiled code is executed, it checks whether the expression still only reads from memory without corresponding shadow storage. If such a check finds shadow storage present, the compiled code is deoptimized. When that taint tracking code is later recompiled, the compiler does not assume the propagated value a constant anymore, and the newly compiled code dynamically accesses shadow storage as necessary.

Shadow storage must be visible and accessible only to instrumentation code in order to prevent the instrumented program from detecting or manipulating taint propagation. Another reason for separating it from value storage is that managed runtimes can specialize the latter to the actually stored values to improve performance [25, 49]. The separation leaves this ability unaffected by whether values are tainted, and allows for separate optimizations specific to shadow storage.

3.1.2 Shadow Storage Representation.

Shadow Variable. Shadow storage for a scoped variable, i.e., a local or global variable, is allocated in the form of a *shadow variable* in the same scope. In Figure 3a, the scope contains both a variable `x` and a corresponding shadow variable, named `shadow[x]`, which holds a taint label for the value stored in `x`.

Shadow Array. Shadow storage for an array value, as Figure 3b shows, takes the form of a separate array, a so-called *shadow array*, of the same length. Managed runtimes typically represent array values as special objects that store metadata such as the array length or element type in addition to a pointer to the actual array storage. We extend this metadata with a pointer to the shadow array. Figure 3b shows this pointer held in a field named `shadow`.

Shadow Object. For storing the taint labels of the values that are stored in the fields of an object, as Figure 3c shows, we allocate a separate object value, which we refer to as a *shadow object*. If the language runtime supports adding fields to objects dynamically, the reference to a corresponding shadow object can be stored in such a dynamically added field if that field can be hidden from the instrumented program. In Figure 3c, the reference to the shadow object is stored in such a dynamic field, which is named `shadow`. The shadow object in that example only contains a field named `a`, but the corresponding object also has a field named `b`. This is because fields of a shadow object are also allocated on-demand, i.e., when the equally named field in the corresponding object first receives a tainted value. The object shown in Figure 3c never stored a tainted value in its `b` field, and thus no such field was added to the shadow object.

Shadow Allocation. Native allocations are regions of allocated memory that can be accessed at byte-level, but accesses to it typically involve multiple bytes. For example, all data structures created by C code can be cast to and accessed as byte arrays. However, accesses to them are typically performed at a higher granularity. An array of `long` values will typically only be accessed at intervals of 8 bytes, i.e., the size of a `long` value, and each time also 8 bytes will be accessed simultaneously. In order to avoid having to store the same taint label multiple times, and thus having to check whether any of the multiple bytes are tainted in a read access, we specialize the shadow storage for a native allocation, which we refer to as a *shadow allocation*, to the granularity to which the respective allocation stores data at runtime. This approach is especially suitable to managed runtimes like the one implemented by Rigger et al. [45] which already track metadata for each allocation. For such runtimes, a link to the shadow allocation and its current granularity can simply be added to that metadata, which the runtime anyways needs to retrieve upon each access to a native allocation. Other analysis platforms need to maintain a mapping between allocations and metadata manually. Previous research, e.g., Lam et al. [35], has described ways of doing so efficiently.

The granularity to which a shadow allocation stores taint labels is initially determined as the highest power of 2 that evenly divides both the size in bytes of the tainted value and the byte offset used by the memory store operation that required the creation of the shadow allocation. Figure 3d shows a shadow allocation with a granularity of 8 for an allocation representing a two-element array of `long` values, i.e., one taint label is stored for every 8 bytes of data. If this array is only accessed as an array of `long` values, each access thus requires only a single load or store operation for the taint label of the written value, rather than 8.

When a shadow allocation is accessed at a byte offset or with a byte-size which is not evenly divided by its current level of granularity, the shadow allocation needs to be transitioned to a finer granularity, namely the next coarsest granularity that would support the access. For example, if one were to cast the array of `long` values from Figure 3d to an array of `byte` values and proceed to overwrite each allocated byte individually, the corresponding shadow allocation would be transitioned to storing taint labels at a granularity of one taint label per allocated byte. Granularity is not immediately reduced to byte-level per default to account for different sizes of the fields of structured values. Consider, for example, a `struct` value in C code which contains both `long` and `int` fields. If a `long` field is stored a tainted value first, the shadow allocation would specialize to a granularity of 1 taint label per 8 bytes. When an `int` field then receives a tainted value, the shadow allocation is transitioned to a granularity of 1 taint label per 4 bytes, which requires storing 2 taint labels for each `long` field. An access to a `long` field then still needs to access only 2 taint labels rather than 8.

3.2 Specialized Access to Taint Label Storage

An expression that reads from an object or native allocation needs to access shadow storage to determine if a value that was read is tainted. We optimize such expressions by specializing the code for accessing shadow storage to the properties of the objects, native allocations and corresponding shadow storages they access. During interpreted execution, such expressions collect information about these properties. Code compiled from these expressions is optimized as discussed in the following paragraphs. This code checks whether the storages and shadow storages it reads from still exhibit these properties before a such optimized expression is executed. If such a check fails, the compiled code is deoptimized and replaced with an unspecialized version to guarantee correct taint propagation.

3.2.1 Shadow Objects. Managed runtimes for languages such as JavaScript, which support dynamically adding fields to objects, often optimize field read expressions by specializing them to the layouts of the objects they read from [49]. For example, consider an expression that reads the value stored in a field named `b`. If that expression always reads from objects with the same layout, this expression is specialized to objects of that layout. When this expression is compiled at runtime, it is known at which offset in such objects `b` is located, so the compiled code need not determine this offset at runtime. Instead, the compiled code only needs to check that the objects it reads from still have the same layout, which typically only involves a single pointer comparison [49]. When an object with a different layout is observed, the compiled code is deoptimized and replaced with a version that determines the offset at runtime.

We leverage the runtime's ability to optimize object lookups to speed up accessing shadow objects. To determine whether a value stored in a field of an object is tainted, an expression reading from that field first has to check if that object references a shadow object. As stated in Section 3.1, we store this reference in a special field, which is allocated on-demand. In Figure 3c, this field is named `shadow`. If an expression reading from a field `b` is anyways specialized to a layout, it can be determined at compile time whether objects of that layout contain a `shadow` field. If not, then for the purpose of further optimizations any value returned from this read expression is known to not be tainted. It is thus not necessary to access shadow storage at runtime, and the taint label to propagate can be a constant null-value. The field read expression is anyways deoptimized if it encounters an object with a different layout.

If an expression reading from a field `b` is specialized to a layout which does include a `shadow` field, we additionally attempt to specialize the code for reading from the shadow objects to their layouts. If at runtime all these shadow objects have the same layout, and if that layout does not contain a field named `b`, as is shown for the shadow object in Figure 3c, then values read from that expression can still be assumed to not be tainted. Code compiled from such an expression checks at runtime that the actually observed shadow objects still have the same layout, and is deoptimized when this check fails. Code compiled from a specialized field read expression thus only loads a dynamic taint label from shadow storage if it both reads from objects with associated shadow objects and those shadow objects also have a field named `b`.

3.2.2 Shadow Allocations. Accessing a native allocation requires computing the number of slots in the shadow allocation and their offsets that are affected by the access depending on the shadow allocation's current granularity and the offset and byte size of the value access. Additionally, the access may require transitioning the shadow allocation to a finer granularity. To alleviate this cost, we specialize read operations to the native allocations they observe at runtime. In each specialization, the granularity to which these shadow allocations store metadata and the size and offset of the corresponding access to the native allocation is assumed constant. During optimization, slow math operations for, e.g., checking whether the current granularity supports a particular read operation can thus be partially evaluated at compile time. For example, since the granularity is always a power of 2, math operations to check whether an offset or byte-size is evenly divided by a constant granularity value can be replaced by the compiler with faster bit-wise operations.

3.3 Profiling Taint Labels of Statement Inputs

We extended taint tracking code for each kind of expression that introduces external data to a statement to profile the values it propagates. Such *input expressions* include reading from memory or function parameters, calling another function for a value and catching an exception. If such an expression never produces tainted values, the corresponding taint tracking code always propagates `null`. Similarly, when the produced values are always tainted, the taint tracking code always propagates the same taint label. In these cases, the propagated value is effectively constant, but the compiler cannot know that. Profiling this value discovers these cases and, if they occur, instructs the compiler to speculatively assume the propagated value to be that constant to enable additional opportunities for applying further optimizations. To verify this assumption, the compiler inserts a run-time check into the compiled code to verify whether each value to propagate is still that constant. Note that if the optimizations of Sections 3.1 and 3.2 already reduced the propagated value to a constant `null`, this check is trivially optimized away during partial evaluation. If a different value than the profiled constant has to be propagated, the compiled code is deoptimized. When the corresponding expression is later recompiled, no more assumptions about the propagated value will be made.

```
v = g();
for (i = v; i > 0; i--) /* large body */
return v;
```

Figure 4. Example for an optimization opportunity.

Advanced compilers such as GraalVM's JIT compiler have advanced data- and control-flow optimization capabilities, which we leverage to speedup taint tracking. Such a compiler can use insights derived from our speculative assumptions for one statement to further optimize others. In Section 3.1, for example, we explained how these insights enable the compiler to optimize away run-time checks for the validity of our speculative assumptions in some cases. By profiling whether the inputs of statements are tainted, we achieve a similar effect. Consider the code shown in Figure 4. It stores the number returned from calling function `g` into a local variable `v`. After executing this number of iterations of a loop with a large body, the value of `v` is returned from the current function. If the values returned by `g` are always tainted, then by profiling we reduce their taint label to a constant for further optimization. During this optimization, the compiler can propagate that constant to the taint tracking code for the expressions initializing the loop variable `i` and returning the value of `v`. This code thus needs to neither load it from shadow storage at runtime nor check whether it matches a profiled value. Since the shadow variable of `v` does not escape the compilation unit, the compiler can also remove the write access to it since there is no more code that would read the written value. The same constant propagation can be applied to the shadow variable of `i` since after partial evaluation both the initialization and decrement assign only the same constant taint label to it. As a result, the loop body can be compiled knowing the taint label of `i`, which also avoids having to load that taint label from shadow memory each time the loop body is executed.

3.4 Specialized Storage Update

Similar to memory reads, we also use profiling and specialization to optimize taint tracking code for memory writes.

First, we aim to reduce the size of compiled code by speculatively excluding logic for allocating or updating shadow storage from it. For scoped variables it is known at compile time whether corresponding shadow storage exists. For other kinds of memory writes we use the speculative assumptions discussed in Section 3.1 or, after these assumptions have been invalidated, profiling to determine whether taint propagation logic can be specialized to the target not having shadow storage. When a memory write operation is specialized to the target not having shadow storage, we also profile whether the value to write is tainted. If that value is never tainted the logic for manipulating shadow storage need not be optimized or contained in compiled code. A similar effect is achieved if the memory write operation can be specialized to the target having shadow storage, since the instructions for allocating shadow storage need not be included in the compiled code. Common compiler optimizations such as *code duplication* [38] or *inlining* can significantly increase the size of a compilation unit, so dynamic compilers apply them only if the resulting compilation unit does not become too large to process. Reducing the size of taint propagation code reduces the threat that the compiler cannot perform such optimizations and thus loses on peak performance.

Second, we optimize taint tracking code for writing to object fields by profiling the layout of the accessed shadow objects. When shadow objects do not yet contain the field to be written, they need to be transitioned to a new layout. Through our profiling, the base layout can become a constant, which enables the compiler to compute the new layout already at compile time, and to optimize the compiled code for the transition between these known layouts. Similarly, if taint tracking code needs to allocate new shadow objects, we also compute their initial layout at compile time.

Third, we specialize taint tracking code for writing to native allocations to the byte size (S) of the value to write and the granularity (G) at which the observed shadow allocations store taint labels. If both S and the offsets to which values are written are aligned to[1] G, then no transitioning of shadow allocations to finer granularity is required. In each instance of these specializations we profile whether transitioning is required to exclude the code for performing this transition in specializations where it is not. If a write access does require transitioning, the code for storing the taint label performs this transition and then recursively invokes itself in order to store the taint label to the transitioned shadow allocation. The number of slots (U) in the shadow allocation that need to be updated is the quotient of S and G. Since these numbers are constant in each specialization, U can be partially evaluated during compilation. Thus, the loop in each specialization that iterates U times to update these slots has a constant bound, which enables the compiler to *unroll* it. Code that is compiled from a specialization in which S is equal to G is thus reduced to only a single write access to the shadow allocation after checking whether the offset that is currently being written to is aligned to G. This check can be done efficiently with bit operations since G is always a power of 2 [2]. If the offset to write to is a constant, for example when the native allocation contains an object whose field is being accessed, this check falls away during partial evaluation. Note that while the number of specializations that can be active at the same time is usually limited, we do not consider this to be an issue in practice since S is typically a constant and G is restricted to powers of 2.

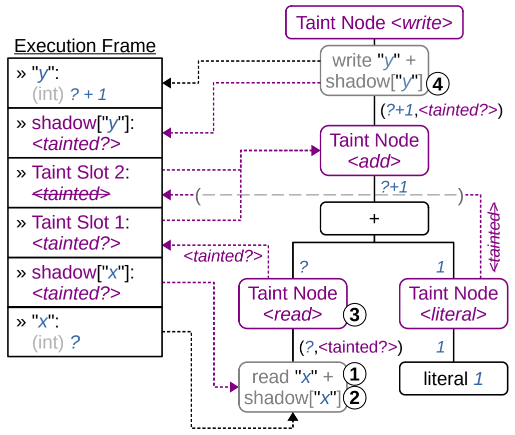

Figure 5. Taint propagation with optimized TruffleTaint for the example of Figure 1. Shadow variables for `x` (`shadow["x"]`) and `y` are allocated. The values stored in `x` and its shadow variable are not known at compile time, but `x` and `y` are known to store integers. Edges are annotated with the values that flow along them when the AST is executed. Encircled numbers indicate where the optimizations from Section 3 are implemented.

4 Implementation in TruffleTaint

We implemented our optimizations in TruffleTaint and additionally optimized its techniques for taint propagation to be amenable to the optimizations implemented by the GraalVM JIT compiler. This required modifications in both the taint tracking engine and the supported language runtimes. Figure 5 shows the instrumented Truffle AST of Figure 1 under the optimized TruffleTaint and will serve as an example. In the following, individual taint nodes in that Figure will be referred to as $TN_{<operation>}$.

We implemented and optimized shadow storage as discussed in Section 3.1. Shadow variables, as Figure 5 shows, are allocated as entries of the *frame*, Truffle's container for local variables which is specially optimized by the GraalVM JIT compiler. All frames of the same method share a common layout. Changing this layout by, e.g., dynamically adding a shadow variable triggers recompilation of all code that uses it. At compile time it is thus known whether the frame currently contains a shadow variable for a particular local variable. We implemented shadow objects and shadow arrays in GraalVM's JavaScript runtime and we implemented shadow allocations in GraalVM's LLVM runtime in order to support efficiently storing metadata in memory allocated by each language.

Since TruffleTaint by design delegates the task of storing boxed values to language runtimes, we also implemented the speculative optimizations related to memory access there. To this end we implemented *shadow read nodes* and *shadow*

[1]0, equal to or a multiple of

write nodes, which extend the implementation of a runtime's existing read and write nodes, respectively, in order to read from, allocate, or write to the appropriate shadow storage with the optimizations introduced in Section 3. Some such nodes only intercept the value that was read or is to be written, respectively, in order to load or store the corresponding taint label and reuse the existing nodes for reading or writing the actual value. For other nodes, however, we found that an efficient implementation required a more integrated approach, so we partly duplicated the existing code. Separating extended nodes from the original ones enabled us to insert modified nodes into the Truffle AST only when it is instrumented, which enabled us to collect an accurate baseline of the peak performance of our benchmarks.

Figure 5 shows a shadow read node, which reads the current value of `x` and loads its taint label from the corresponding shadow variable. If a shadow read node finds a taint label for a value it read, it allocates a boxed value containing both that taint label and that value. The taint node for the read operation, TN_{read}, profiles whether it receives boxed values from the read node. If this condition is profiled to be *false*, values returned from that node can be optimized as constantly not tainted and the compiled code need not contain instructions for propagating their taint labels. When TN_{read} receives boxed values, it unboxes them to extract the taint label for further propagation. That taint label is also profiled at that point as discussed in Section 3.3. The same profiling is done also in taint nodes for, e.g., function calls or argument accesses.

When shadow read nodes or taint nodes need to allocate a boxed value for the first time they specialize themselves to doing this allocation always, i.e., without checking. Doing so enables them to benefit from *partial escape analysis* (PEA) [47], an optimization in which the allocated object is decomposed into its fields. All accesses to these fields are replaced with accesses to the proper locations in the frame, and the allocation of the object is removed[2]. For the example in Figure 5, PEA removes the allocation of the boxed value in the shadow read node and replaces the read access to the taint label it stores in TN_{read} with a read access to the location where that taint label was stored, namely the shadow variable of `x`. Taint nodes further specialize themselves to receiving or not receiving boxed values in order to exclude code corresponding to the respective other branch from compilation.

Like shadow variables, taint slots are implemented as entries in the frame, which the GraalVM JIT compiler has special knowledge of to enable optimizations such as partial evaluation and constant propagation across read and write accesses to it. When the example AST in Figure 5 is compiled, the taint propagation logic for the intermediate expressions is partially evaluated. During this process, the assignment of the taint label to Taint Slot 1 and the subsequent reading of that slot in the TN_{Add} is removed by the compiler and replaced by accessing the value returned from TN_{Read} in TN_{Add} directly. The taint label of the increment value is a constant `null` value and it is moved into TN_{Add} in the same way. The implementation of TN_{Add} checks if any of the addition's operands had a taint label to determine whether one needs to be placed on the sum as well. During partial evaluation this determination is simplified to the taint label of the left operand, i.e., whatever is currently stored in the shadow variable of `x`, since the right operand is constantly not tainted.

Each taint node clears the taint slots it reads from after it has determined which value to propagate. This practice restricts the range of code in which a taint slot can have a value that is not known at compile time. Because of this practice, JIT-compiled code can avoid keeping track of the last stored value at runtime outside of that range, which it would otherwise have to do so the state of the frame object can be reconstructed during deoptimization.

As proposed in Section 3.4, shadow write nodes specialize on both whether the value to write is tainted and whether the target already has shadow storage. Figure 5 shows a shadow write node, which writes the new value of `y` and stores its taint label to the corresponding shadow variable, whose existence and location is known at compile time.

TruffleTaint requires that taint nodes box tainted values if their parent node, i.e., a shadow write node, implements a write operation. We further optimized taint nodes so that each specializes to returning boxed values after it had to return a tainted value once. Each shadow write node also checks at runtime whether the speculative assumption about the target to which it writes not having shadow memory (c.f. Section 3.1) is still valid. The compiler can remove this run-time check if at compile time either the assumption has already been invalidated or if the taint node has not yet specialized to returning boxed values. A specialization of the taint node would require deoptimizing the compiled code, which in turn would reenable the check and thus ensure proper taint propagation. Similarly, the run-time check within the taint node on whether to specialize can be optimized away as well if the value to write is known to not be tainted, e.g., as a result of the other optimizations we introduced in Section 3.

As explained previously, optimizations reduce the taint label of the value to be stored into `y` to the value currently stored in the shadow variable of `x`. Since `x` contains a tainted value TN_{Add} specializes on always returning boxed values, and the shadow write node consuming those boxed values immediately unboxes them to store the contained values into `y` while storing the contained taint labels into the shadow variable of `y`. Partial escape analysis removes the allocation

[2]The GraalVM JIT compiler can be configured to apply multiple iterations of PEA [10] instead of its regular single iteration. We enabled this configuration when running programs with TruffleTaint so that unboxed values were escape analyzed like in uninstrumented execution.

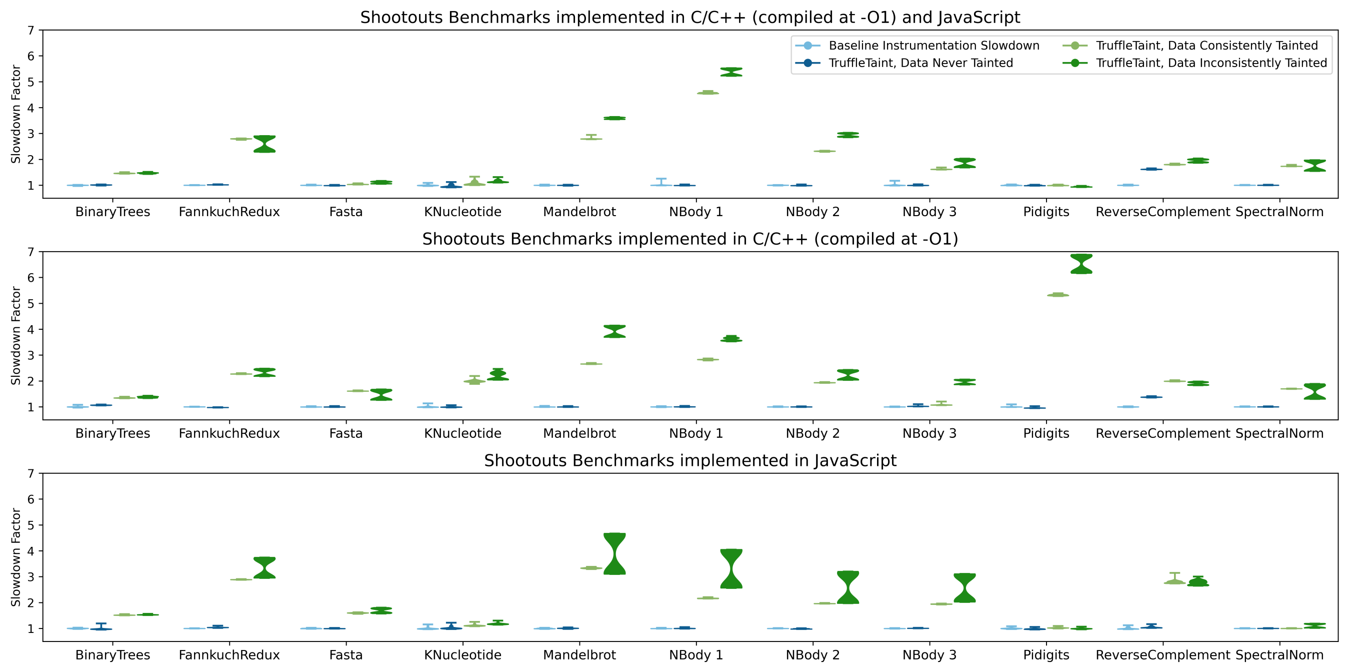

Figure 6. TruffleTaint Slowdown on Shootouts benchmarks.

of these boxed values, which causes the sum and its taint label to be stored directly to y and its shadow variable, respectively. Since the taint label of the sum is, due to partial escape analysis, the value currently stored in the shadow variable of x, the JIT compiler thus reduces the taint propagation logic for this statement to a single copy operation without TruffleTaint itself implementing any static analysis or compiler optimization.

The example in Figure 5 would typically be part of a larger compilation unit. Profiling the taint label of the value stored in x adds a check to the compiled code whether the profile is still accurate. However, it also enables other statements that read from y before any other statement writes to it to be partially evaluated knowing its taint label. This avoids accessing shadow storage, which offsets the performance cost of that additional check.

5 Evaluation

We evaluated our optimizations implemented in TruffleTaint using benchmarks from the *Computer Language Benchmarks Game* [1] (often called the *Shootouts*), the *SPECint 2017* benchmark suite [9] and a multi-language benchmark based on LLHTTP [5], i.e., the internal HTTP header parser used by Node.js [7].[3] Figures 6, 7 and 8 show how much peak performance slowdown these benchmarks incur from taint propagation with TruffleTaint relative to running them on GraalVM without instrumentation. The results show that TruffleTaint can exhibit slowdown as low as 0% when benchmark programs do not operate on tainted data, and that it exhibits up to ~5.52x when they do. The absolute numbers are given in Appendix A, important ones are also given in this Section. We use these benchmarks to answer the following research questions:

RQ1: Can TruffleTaint avoid slowdown when programs do not operate on tainted data?
RQ2: How much slowdown does TruffleTaint incur when programs do operate on tainted data?
RQ3: Does program size affect slowdown incurred by TruffleTaint?
RQ4: Does the total amount of tainted data a benchmark operates on affect slowdown incurred by TruffleTaint?
RQ5: Do implementation language or language boundaries affect slowdown incurred by TruffleTaint?

To answer RQ1, RQ2, and RQ5 we instrumented implementations of the Shootouts benchmarks with TruffleTaint and compared their peak performance to that of the same benchmarks without this instrumentation. The Shootouts benchmarks are comprised of several well-defined workloads exercising different properties and features of the implementation language. We evaluated implementations of these workloads in C or C++, in JavaScript, and in a version where both languages interact. We extended these implementations to include taint sources and taint sinks which

[3]We excluded benchmarks from these suites from our evaluation when, independent of TruffleTaint, GraalVM did not support compiling or executing them.

introduce tainted values into the benchmarks' main workloads and verify that taint labels are propagated correctly.[4]

5.1 Experimental Methodology

We collected the numbers for the Shootouts and LLHTTP-based benchmark on a system with an Intel Core i7-3770 processor and 16GB RAM. The numbers for the SPECint 2017 benchmarks were collected on a system with an Intel Core i7-8700 and 32GB RAM to account for the higher memory requirements of these benchmarks. In either case the corresponding Linux systems were configured to collect consistent benchmark results by, e.g., disabling CPU features such as Intel Turbo Boost and Hyper-Threading as well as using a performance-oriented CPU governor.

The collected numbers reflect the slowdown on the peak performance of these benchmarks. To collect these numbers we executed each benchmark with enough warmup iterations to allow the compiled code to stabilize, i.e., no more compilation occurred during benchmark iterations. The baseline for each number is the mean execution time of the benchmark iterations of the corresponding benchmark on GraalVM without any instrumentation. We also repeated each experiment multiple times to verify the results we collected were reasonably stable and noise-free.

5.2 Slowdown without Tainted Data

To answer RQ1 we executed the Shootouts benchmarks on GraalVM with TruffleTaint's instrumentation, but configured all taint sources to mark the data they produce as not tainted. Thus the benchmark programs were fully instrumented to perform taint propagation, but the values they operated on were never tainted. If the optimizations we introduced in Section 3 were effective, then the code produced by the GraalVM JIT compiler should not include any instructions for taint propagation, since there is no taint to propagate. Our results as given by Figure 6 show that TruffleTaint incurs a slowdown of only up to 2% compared to execution on GraalVM without instrumentation for 24 of the 33 individual benchmark programs when the *Data* they operate on is *Never Tainted*. This compares favourably to DECAF++[22], which to our knowledge is the dynamic taint analysis system with the least slowdown on programs that do not propagate taint, but still incurs at least 4% slowdown.

Out of all 33 benchmark programs of the Shootouts suite, only 4 benchmarks showed a slowdown of over 7% (c.f. Figure 6), namely up to 40%. This slowdown, however, can also be a result of the compiler making different optimization decisions, rather than by taint propagation. Optimization decisions made by the GraalVM JIT compiler are based on timing and on various metrics of the code to be compiled. As a result, modifying the code to invoke instrumentation

[4]The implementations are available at https://github.com/jkreindl/taint-benchmarks

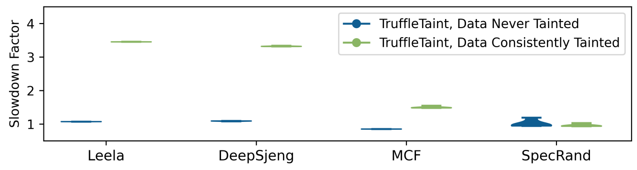

Figure 7. TruffleTaint Slowdown on Spec benchmarks.

callbacks can affect optimization decisions, leading to different compiled code for the same workload [23]. Figure 6 includes the *Baseline Instrumentation Slowdown*, which is the slowdown incurred by instrumenting the program to invoke empty callbacks whenever an expression is executed. This data shows a slowdown of ~35.78% by such empty instrumentations on the *FannkuchRedux* benchmark implemented in a mix of C and JavaScript. This baseline slowdown is incurred from Truffle instrumentation itself, not added by TruffleTaint. On the other hand, the *ReverseComplement* benchmark implemented in C++ exhibits ~38.06% slowdown with TruffleTaint, while instrumentation alone shows no slowdown. Increasing the amount of code to compile for a particular method, for example by adding taint tracking code or any other instrumentation code, can particularly impact the compiler's decisions on whether to inline a particular method at a specific call site, even if the taint tracking code in both methods could be removed completely during subsequent optimization. The *MCF* benchmark from the SPEC benchmark suite, as shown in Figure 7, instead experiences a ~14.87% speedup under TruffleTaint when there is no taint to propagate. We manually inspected compilation graphs to verify that our optimizations, rather than external factors influencing compilation decisions, were generally the main cause for a lack of or reduction in slowdown from taint propagation in our benchmarks.

5.3 Slowdown with Tainted Data

To answer RQ2 we executed our benchmarks with TruffleTaint's instrumentation enabled and tainting all values originating from a taint source. Although taint sources are not required to produce such *Consistently Tainted Data*, e.g., in case not all data returned from a particular system call or only data read from specific files is sensitive, we believe this to be the prevalent configuration. As explained in Section 3.3, when the values propagated as taint labels of an expression's inputs are known constants, then the value propagated for the expression's output value is too. Since statements consist of expressions and compilers perform constant propagation, the same also holds true for statements. Since compilers typically perform constant propagation also through assignments to local variables and other memory writes, that concept can be extended to sequences of statements too when each statement consumes only values produced by a preceding statement. Thus, if all inputs to the first statement are each either consistently tainted or consistently not tainted, the compiler can optimize away all taint tracking code for those statements. The results of our benchmarks suggest, and we verified this by manually inspecting compilation graphs for select benchmarks, that due to our optimizations GraalVM's JIT compiler is indeed able to optimize away such redundant taint propagation. Thus benchmarks in which fewer statements operate on inconsistently tainted data tend to incur less slowdown. 7 of our 33 Shootouts benchmarks even exhibit under 20% slowdown despite heavily operating on tainted data. The average slowdown across the Shootouts benchmarks is ~2.10x, and the highest slowdown we observed was ~5.52x.

The benchmark which incurred the highest slowdown, as Figure 6 shows, is the *Pidigits* benchmark implemented in C++. This benchmark heavily relies on function calls in which heap-allocated objects are passed as arguments. We observed, in general, that higher slowdown corresponds to data flowing into expressions not being consistently tainted, tainted data traversing a call boundary and a higher number of allocated memory with shadow storage.

In some programs, the main workload may operate on both tainted and untainted data in the same statements to a significant degree, which prevents our profiling for data being consistently tainted. To evaluate TruffleTaint's performance in such programs we executed the Shootouts benchmarks with TruffleTaint's instrumentation enabled, but prevented any specialization on data being consistently tainted by only enabling taint sources and sinks in every other benchmark iteration. Specialization on inputs being consistently not tainted, however, was not affected to account for the insight that tainted data usually spreads to only a part of the whole program. In this configuration, as illustrated in Figure 6, our benchmarks exhibited a slowdown between 0% and ~6.76x.

Lack of consistency in whether data was tainted generally lead to increased slowdown. However, the *Pidigits* benchmark implemented in JavaScript did not exhibit slowdown. This benchmark, in contrast to its C++ implementation, consists of a single compilation unit, which gave the compiler access to the flag enabling taint sources and sinks during optimization.

Figure 6 shows a high standard deviation in the *Data Inconsistently Tainted* configuration for some benchmarks. While code compiled from these benchmarks includes instructions for allocating and accessing shadow storage on-demand, these instructions only needed to be executed in iterations with enabled taint sources. These iterations were thus consistently slower, and the difference in slowdown between both kinds of iterations is larger for benchmarks whose workload is dominated by memory accesses.

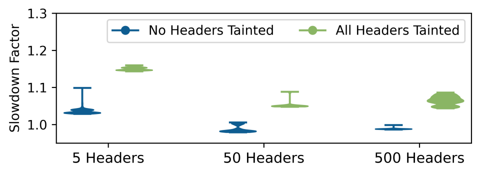

Figure 8. TruffleTaint Slowdown on LLHTTP benchmarks.

5.4 Performance Impact of Program Size

To answer RQ3 we instrumented several programs of the SPECint 2017 benchmark suite. In contrast to the smaller Shootouts programs, SPECint includes real-world C/C++ applications consisting of several thousand lines of code which perform common workloads. We adapted these benchmarks to mark all input data as tainted and to make sure that this taint is propagated. As Figure 7 shows, these benchmarks incur slowdown ranging from 0% to ~3.45x for consistently tainted data, which confirms that our optimizations also support larger programs. However, taint propagation itself significantly increases the size of the code to be compiled, which can become a problem for applications whose peak performance depends on very large methods. To be able to run all programs shown in Figure 7 we needed to increase the compiler's upper limit for the size of methods it would compile by a factor of 20. We also increased the maximum amount of memory available to the compiler to 24GB.

5.5 Performance Impact of Data Amount

To answer RQ4 we implemented a benchmark based on *llhttp*, the HTTP header parser of the Node.js framework. While LLHTTP itself is implemented in C code, it is invoked by JavaScript code via Node.js' native interface. Parsers are often of interest for taint tracking applications, for example when reverse engineering input formats or guiding Fuzzers, so this is a realistic workload. Our benchmark assembles an HTTP request in JavaScript code by writing a fixed number of headers and corresponding values, which consists of random characters, to a buffer allocated in native memory by C code. The parser, which is implemented in C, parses this request, converts the parsed headers and their values into JavaScript String values and passes these String values back into JavaScript code via a callback, which stores these Strings into a JavaScript array. In this benchmark, the values of the individual headers are tainted, so after the entire request has been parsed, the String values stored in the JavaScript array are checked to be tainted. By varying the amount of headers per request we can illustrate the impact of the amount of tainted data a benchmark processes on the slowdown it incurs from taint propagation by TruffleTaint. Moreover, the implementation of the benchmark utilizes both JavaScript and C code, which validates our results for both languages. The results of executing this benchmark multiple times with a successively increasing number of headers per request are shown in Figure 8, the concrete numbers are given in Appendix A. Running this benchmark with 50 or 500 headers per request showed no significant difference in slowdown for both runs, both when all headers were tainted and when none were. Running the benchmark with only 5 headers per request instead shows higher slowdown from instrumentation with TruffleTaint. The parser contains a separate, faster code path for requests containing only a small number of headers, which seems to incur more slowdown from taint propagation. Since the GraalVM LLVM runtime aims to exclude unexecuted branches from compilation, different code paths being executed also lead to a different Truffle AST to compile. The runs with the same branches taken, in contrast, did not exhibit a significant difference in slowdown, which indicates no direct correlation between the total amount of tainted data a benchmark operates on and the slowdown it incurs from taint propagation.

5.6 Language Independence

Figure 6 shows that each implementation of the Shootouts suite incurred slowdowns ranging from 0% to ~6x in its benchmarks and thus answers RQ5.

Differences in slowdown for the various implementations of the same benchmark arise from the design of the individual benchmark programs, rather than the choice of the implementation language. Concretely, the more a benchmark accesses allocated memory with shadow storage the more slowdown it incurs from TruffleTaint. This is because the JIT compiler cannot remove read or write accesses to shadow storage if it may escape the compilation unit, and accessing shadow storage can involve multiple checks and offset calculations. Another factor for increased slowdown is propagating tainted values via function calls to another compilation unit. This slowdown stems from partial escape analysis not being able to remove the allocation of boxed values, which we use to propagate the taint labels of function arguments, when boxing and unboxing do not happen in the same compilation unit. The more the performance of a benchmark depends on these features, the more slowdown it will incur from taint propagation. To give an example, the JavaScript implementation of the *Pidigits* benchmark propagates tainted data only through local variables within a single compilation unit. *Pidigits*' C++ implementation, in contrast, incurs more slowdown since it frequently propagates tainted values through memory allocations[5] and through multiple compilation units. The JavaScript and C implementations of the *NBody 2* benchmark, in contrast, both have the same structure and both store tainted values into allocated memory, i.e., into a JavaScript array and a C native allocation, respectively. These benchmarks exhibit only a ~5.22% difference in slowdown from taint tracking between them.

The GraalVM LLVM runtime uses shadow allocations to store taint labels, since all memory allocated by programs it executes can be accessed at byte-level. In contrast, GraalVM's JavaScript runtime allocates shadow objects and shadow arrays. However, Figure 6 shows the same range of slowdown for both C/C++ and JavaScript benchmarks, which indicates that neither is necessarily faster. This further indicates that implementation language is not a dominant factor in how much slowdown a benchmark exhibits.

Figure 6 also shows the language boundary to not be a barrier to our optimizations. Namely, it shows that the Shootouts benchmarks in which C/C++ code interacts with JavaScript code incur the same range of slowdown as the single-language implementations.

6 Related Work

In this section we compare our approach to related work.

6.1 Conditional Taint Propagation

DECAF++ [22], an optimized version of the *DECAF* [27] whole-system taint tracking engine, switches between a *track mode*, in which full taint propagation is performed, and a *check mode*, in which the instrumentation only checks whether memory loads or stores involve tainted data; if they do, it switches to track mode. In check mode, DECAF++ incurs only 4% slowdown in addition to the 5x to 10x slowdown incurred by the underlying emulator [22]. *Track mode* corresponds to the original DECAF, which incurs additional slowdown of up to 8.15x [27]. Ho et al. [28] implement their version of check mode on top of a fast hypervisor, while their version of track mode is implemented on top of an emulator. Their modified hypervisor is up to 90% slower than its original implementation due to the need for checking whether to switch to the emulator.

While Ho et al. and DECAF++ are based on whole-system taint tracking, LIFT [44] relies on application-level taint tracking. Before executing a block of instructions, LIFT checks each memory location read or written by these instructions and executes the block either with or without taint propagation depending on whether this memory currently contains tainted data. LIFT also applies other optimizations, but still incurs up to 7.9x slowdown. The *Taint Rabbit* [24] dynamically creates separate versions for each basic block that are each optimized for a specific combination of taint states of the block's inputs and outputs and dispatches between them.

TruffleTaint replaces run-time checks whether executed code requires taint propagation in compiled code with speculative assumptions. When nothing in a program actually introduces tainted data, these assumptions are never disproven, and so TruffleTaint avoids slowdown from taint propagation entirely for some programs. Moreover, TruffleTaint decides

[5]JavaScript provides a native data type for high-bit-width numbers, while C++ code instead has to represent them by heap-allocated data structures. The JavaScript benchmark is faster since it thus accesses only atomic values rather than memory allocations.

for each individual input of a statement whether taint needs to be propagated for it, while DECAF++, LIFT, and Ho et al.'s approaches can only decide this for an entire sequence of instructions. In contrast to the Taint Rabbit, TruffleTaint does not require dispatching between different basic blocks at each control flow decision. Unlike TruffleTaint, these approaches also do not optimize the taint propagation logic itself, they only try to avoid executing it entirely.

6.2 Static Optimizations for Taint Propagation

Some dynamic taint tracking systems use static analysis to avoid run-time taint propagation [30, 35, 52]. For example, Kang et al. [30] and Zhang et al. [52] employ static taint analysis in addition to dynamic taint analysis in order to limit run-time taint propagation to those parts of an instrumented program where the static taint analysis cannot preclude the presence of tainted data. Lam et al. [35] use another static analysis technique, *program slicing*, to detect all paths between taint sources and taint sinks, and only perform run-time taint tracking there. These approaches require a separate static analysis engine implementing the same propagation semantics as the dynamic analysis engine. TruffleTaint, in contrast, is a purely dynamic analysis, but still achieves much of the same effect. Since TruffleTaint profiles for each location at which values enter a compilation unit whether those values are tainted, it also avoids run-time taint propagation where no tainted data arrives. It is also more precise in doing so due to its knowledge of which operations are reached by tainted data at runtime, while static analysis only determines which operations *could* be reached.

Jee et al. [29] manually implemented common compiler optimizations to apply them to just the instrumentation code for taint tracking. We instead leverage an existing compiler that is capable of these and other optimizations, and additionally create further opportunities for speculative optimization.

Phosphor [13] is a taint tracking engine for Java that is based on bytecode instrumentation and achieves an average slowdown of 53%, but up to 2.2x, on the DaCapo benchmark suite. Similar to TruffleTaint, Phosphor uses shadow arrays and shadow variables. However, it allocates them eagerly instead of on-demand. Phosphor also uses boxed values, but limits this method of taint propagation to primitive values and arrays thereof. However, Phosphor does not use speculative optimizations to avoid slowdown in application runs that do not propagate taint. Additionally, Phosphor manually applies simplifications to the inserted taint tracking code after instrumentation. While Phosphor is limited to Java bytecode, TruffleTaint supports multiple languages.

Any implementation of DTA which modifies the source code or the intermediate representation of an application to analyze in order to perform taint propagation, e.g., the LLVM Data-Flow Sanitizer [6] or *Phosphor* [13], can trivially benefit from an optimizing compiler. TruffleTaint adds to this by directing this compiler to make speculative assumptions about the instrumented program in order to enable it to perform additional optimizations for the taint tracking code.

libDFT [32] and Minemu [14] reduce slowdown from taint tracking in native code to ~3x by exploiting characteristics of the x86 platform. The Taint Rabbit additionally performs conditional taint propagation and achieves a slowdown of 1.7x for some benchmarks when taint is propagated. Our approach can achieve similar average slowdown, but is platform-independent.

Chin et al. [18] add character-level shadow storage to String values on-demand to be able to quickly check whether a specific character could be tainted. We apply that same concept more generally to dynamic objects, arrays and native allocations and additionally specialize shadow storage to how the corresponding data structures store tainted values at runtime. DECAF++ also maintains a flag for each memory page whether it contains tainted data, while our approach maintains such flags on a finer per-allocation granularity.

7 Conclusion

In this paper, we presented how TruffleTaint, a polyglot dynamic taint analysis platform, leverages speculative optimization in concert with dynamic compilation to achieve low overhead taint propagation. We showed that our optimization strategies are applicable to both dynamic, object-oriented languages like JavaScript and languages like C and C++ which are typically compiled statically. Our evaluation showed that using these optimization strategies TruffleTaint incurs a peak performance slowdown between 0% and 40% when instrumented programs do not operate on tainted data. Our evaluation further showed that TruffleTaint incurred up to ~5.52x slowdown, with an average slowdown of ~2.10x, when instrumented programs do operate on tainted data.

Acknowledgments

This research project was partially funded by Oracle Labs. We thank all members of the Virtual Machine Research 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 feedback on our work.

A Benchmark Results

Tables 1, 2, and 3 provide absolute numbers for the slowdown measurements discussed in Section 5.

Table 1. TruffleTaint slowdown factor on the Shootouts benchmarks.

Language	Benchmark	Empty Instrumentation	TruffleTaint Never Tainted	TruffleTaint Consistently Tainted	TruffleTaint Inconsistently Tainted
C/C++ & JavaScript	BinaryTrees	0.98 ± 0.01	0.98 ± 0.01	1.43 ± 0.02	1.43 ± 0.02
	FannkuchRedux	1.36 ± 0.01	1.38 ± 0.01	3.78 ± 0.02	3.53 ± 0.39
	Fasta	1.01 ± 0.01	1.00 ± 0.01	1.05 ± 0.01	1.11 ± 0.04
	KNucleotide	0.90 ± 0.07	0.87 ± 0.08	0.96 ± 0.10	1.04 ± 0.09
	Mandelbrot	1.00 ± 0.01	1.00 ± 0.01	2.79 ± 0.02	3.58 ± 0.04
	NBody 1	1.01 ± 0.02	1.00 ± 0.01	4.60 ± 0.04	5.42 ± 0.15
	NBody 2	0.92 ± 0.01	0.91 ± 0.01	2.13 ± 0.01	2.71 ± 0.06
	NBody 3	1.00 ± 0.02	1.00 ± 0.01	1.62 ± 0.01	1.85 ± 0.15
	Pidigits	1.02 ± 0.02	1.01 ± 0.02	1.01 ± 0.02	0.96 ± 0.02
	ReverseComplement	0.94 ± 0.01	1.53 ± 0.01	1.70 ± 0.01	1.83 ± 0.06
	SpectralNorm	1.00 ± 0.00	1.00 ± 0.00	1.73 ± 0.01	1.75 ± 0.20
C/C++	BinaryTrees	0.99 ± 0.02	1.06 ± 0.01	1.34 ± 0.02	1.37 ± 0.03
	FannkuchRedux	1.00 ± 0.00	0.98 ± 0.00	2.27 ± 0.01	2.32 ± 0.13
	Fasta	1.00 ± 0.01	1.00 ± 0.01	1.61 ± 0.01	1.46 ± 0.19
	KNucleotide	0.96 ± 0.03	0.95 ± 0.02	1.90 ± 0.05	2.09 ± 0.13
	Mandelbrot	1.00 ± 0.01	1.00 ± 0.01	2.66 ± 0.02	3.91 ± 0.22
	NBody 1	0.99 ± 0.01	1.00 ± 0.01	2.81 ± 0.02	3.59 ± 0.06
	NBody 2	0.99 ± 0.01	1.00 ± 0.01	1.93 ± 0.02	2.21 ± 0.17
	NBody 3	1.00 ± 0.01	1.02 ± 0.02	1.07 ± 0.02	1.95 ± 0.09
	Pidigits	1.04 ± 0.02	0.99 ± 0.02	5.52 ± 0.07	6.76 ± 0.34
	ReverseComplement	1.00 ± 0.01	1.38 ± 0.02	2.00 ± 0.02	1.91 ± 0.06
	SpectralNorm	1.00 ± 0.00	1.00 ± 0.00	1.70 ± 0.01	1.57 ± 0.27
JavaScript	BinaryTrees	1.03 ± 0.01	1.01 ± 0.02	1.56 ± 0.02	1.58 ± 0.02
	FannkuchRedux	0.99 ± 0.00	1.03 ± 0.01	2.87 ± 0.01	3.30 ± 0.37
	Fasta	1.01 ± 0.01	1.01 ± 0.01	1.61 ± 0.02	1.70 ± 0.09
	KNucleotide	1.01 ± 0.05	1.04 ± 0.06	1.13 ± 0.05	1.20 ± 0.05
	Mandelbrot	1.02 ± 0.02	1.03 ± 0.02	3.40 ± 0.05	3.96 ± 0.77
	NBody 1	0.99 ± 0.01	1.00 ± 0.02	2.14 ± 0.02	3.26 ± 0.70
	NBody 2	1.01 ± 0.01	0.99 ± 0.01	1.98 ± 0.01	2.58 ± 0.58
	NBody 3	1.04 ± 0.01	1.05 ± 0.01	2.02 ± 0.01	2.66 ± 0.52
	Pidigits	1.00 ± 0.03	0.98 ± 0.03	1.02 ± 0.03	0.99 ± 0.03
	ReverseComplement	1.03 ± 0.05	1.08 ± 0.06	2.89 ± 0.12	2.86 ± 0.14
	SpectralNorm	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.10 ± 0.08

Table 2. TruffleTaint slowdown factor on the SPECint 2017 benchmarks.

Benchmark	TruffleTaint Never Tainted	TruffleTaint Consistently Tainted
DeepSjeng	1.09 ± 0.01	3.32 ± 0.01
Leela	1.07 ± 0.00	3.45 ± 0.00
MCF	0.85 ± 0.01	1.50 ± 0.03
SpecRand	1.01 ± 0.11	0.96 ± 0.06

Table 3. TruffleTaint slowdown factor on the LLHTTP benchmark.

# Headers Per Request	TruffleTaint No Headers Tainted	TruffleTaint All Headers Tainted
5	1.04 ± 0.01	1.15 ± 0.01
50	0.99 ± 0.01	1.05 ± 0.01
500	0.99 ± 0.00	1.06 ± 0.01

References

[1] 2021. The Computer Language Benchmarks Game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Accessed: 2021-05-09.

[2] 2021. GeeksforGeeks: Check if n is divisible by power of 2 without using arithmetic operators. https://www.geeksforgeeks.org/check-n-divisible-power-2-without-using-arithmetic-operators/. Accessed: 2021-05-22.

[3] 2021. GraalVM JavaScript Runtime. https://www.graalvm.org/reference-manual/js/. Accessed: 2021-05-09.

[4] 2021. GraalVM LLVM Runtime. https://www.graalvm.org/reference-manual/llvm/. Accessed: 2021-05-09.

[5] 2021. The LLHTTP parser for HTTP headers. https://github.com/nodejs/llhttp. Accessed: 2021-05-18.

[6] 2021. LLVM Data-Flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed: 2021-05-17.

[7] 2021. Node.js. http://www.nodejs.org/. Accessed: 2021-05-09.

[8] 2021. Safe and Sandboxed Execution of Native Code. https://medium.com/graalvm/safe-and-sandboxed-execution-of-native-code-f6096b35c360. Accessed: 2021-05-09.

[9] 2021. SPEC CPU 2017 Benchmark Suite. https://www.spec.org/cpu2017/. Accessed: 2021-05-13.

[10] 2021. Truffle Compiler Flags, Including *–engine.IterativePartialEscape*. https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/Options. Accessed: 2021-06-04.

[11] 2021. The Truffle Framework. https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/. Accessed: 2021-05-15.

[12] F. Araujo and K. 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*, J. Jung and T. Holz (Eds.). USENIX Association, 145–159. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/araujo

[13] J. Bell and G. E. 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, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein (Eds.). ACM, 83–101. https://doi.org/10.1145/2660193.2660212

[14] E. Bosman, A. Slowinska, and H. Bos. 2011. Minemu: The World's Fastest Taint Tracker. In *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6961)*, R. Sommer, D. Balzarotti, and G. Maier (Eds.). Springer, 1–20. https://doi.org/10.1007/978-3-642-23644-0_1

[15] J. Cai, P. Zou, J. Ma, and J. He. 2016. SwordDTA: A dynamic taint analysis tool for software vulnerability detection. *Wuhan University Journal of Natural Sciences* 21 (02 2016), 10–20. https://doi.org/10.1007/s11859-016-1133-1

[16] 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

[17] B. Chess and J. West. 2008. Dynamic taint propagation: Finding vulnerabilities without attacking. *Inf. Secur. Tech. Rep.* 13, 1 (2008), 33–39. https://doi.org/10.1016/j.istr.2008.02.003

[18] E. Chin and D. A. Wagner. 2009. Efficient character-level taint tracking for Java. In *Proceedings of the 6th ACM Workshop On Secure Web Services, SWS 2009, Chicago, Illinois, USA, November 13, 2009*, E. Damiani, S. Proctor, and A. Singhal (Eds.). ACM, 3–12. https://doi.org/10.1145/1655121.1655125

[19] J. A. Clause, W. Li, and A. Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, D. S. Rosenblum and S. G. Elbaum (Eds.). ACM, 196–206. https://doi.org/10.1145/1273463.1273490

[20] J. A. Clause and A. Orso. 2009. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, G. Rothermel and L. K. Dillon (Eds.). ACM, 249–260. https://doi.org/10.1145/1572272.1572301

[21] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irún-Briz. 2008. Tupni: automatic reverse engineering of input formats. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, P. Ning, P. F. Syverson, and S. Jha (Eds.). ACM, 391–402. https://doi.org/10.1145/1455770.1455820

[22] A. Davanian, Z. Qi, Y. Qu, and H. Yin. 2019. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*. USENIX Association, 31–45. https://www.usenix.org/conference/raid2019/presentation/davanian

[23] M. L. Van de Vanter, C. Seaton, M. Haupt, C. Humer, and T. Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *Art Sci. Eng. Program.* 2, 3 (2018), 14. https://doi.org/10.22152/programming-journal.org/2018/2/14

[24] J. Galea and D. Kroening. 2020. The Taint Rabbit: Optimizing Generic Taint Analysis with Dynamic Fast Path Generation. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, H. Sun, S. Shieh, G. Gu, and G. Ateniese (Eds.). ACM, 622–636. https://doi.org/10.1145/3320269.3384764

[25] M. Grimmer, S. Marr, M. Kahlhofer, C. Wimmer, T. Würthinger, and H. Mössenböck. 2017. Applying Optimizations for Dynamically-typed Languages to Java. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes, ManLang 2017, Prague, Czech Republic, September 27 - 29, 2017*. ACM, 12–22. https://doi.org/10.1145/3132190.3132202

[26] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*, Y. Cho, S. Y. Shin, S. Kim, C. Hung, and J. Hong (Eds.). ACM, 1663–1671. https://doi.org/10.1145/2554850.2554909

[27] A. Henderson, A. Prakash, L. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, C. S. Pasareanu and D. Marinov (Eds.). ACM, 248–258. https://doi.org/10.1145/2610384.2610407

[28] A. Ho, M. A. Fetterman, C. Clark, A. Warfield, and S. Hand. 2006. Practical taint-based protection using demand emulation. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006*, Y. Berbers and W. Zwaenepoel (Eds.). ACM, 29–41. https://doi.org/10.1145/1217935.1217939

[29] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. 2012. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society. https://www.ndss-symposium.org/ndss2012/general-approach-efficiently-accelerating-software-based-dynamic-data-flow-tracking-commodity

[30] M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song. 2011. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society.

[31] R. Karim, F. Tip, A. Sochurková, and K. Sen. 2020. Platform-Independent Dynamic Taint Analysis for JavaScript. *IEEE Trans. Software Eng.* 46, 12 (2020), 1364–1379. https://doi.org/10.1109/TSE.2018.2878020

[32] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. 2012. libdft: practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th International Conference on Virtual Execution Environments, VEE 2012, London, UK, March 3-4, 2012 (co-located with ASPLOS 2012)*, S. Hand and D. Da Silva (Eds.). ACM, 121–132. https://doi.org/10.1145/2151024.2151042

[33] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz. 2013. Information flow tracking meets just-in-time compilation. *ACM Trans. Archit. Code Optim.* 10, 4 (2013), 38:1–38:25. https://doi.org/10.1145/2541228.2555295

[34] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck. 2020. Multi-language dynamic taint analysis in a polyglot virtual machine. In *MPLR '20: 17th International Conference on Managed Programming Languages and Runtimes, Virtual Event, UK, November 4-6, 2020*, S. Marr (Ed.). ACM, 15–29. https://doi.org/10.1145/3426182.3426184

[35] L. Lam and T. Chiueh. 2006. A General Dynamic Information Flow Tracking Framework for Security Applications. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*. IEEE Computer Society, 463–472. https://doi.org/10.1109/ACSAC.2006.6

[36] S. Lekies, B. Stock, and M. Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and Moti Yung (Eds.). ACM, 1193–1204. https://doi.org/10.1145/2508859.2516703

[37] D. Leopoldseder, R. Schatz, L. Stadler, M. Rigger, T. Würthinger, and H. Mössenböck. 2018. Fast-path loop unrolling of non-counted loops to enable subsequent compiler optimizations. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*, E. Tilevich and H. Mössenböck (Eds.). ACM, 2:1–2:13. https://doi.org/10.1145/3237009.3237013

[38] D. Leopoldseder, L. Stadler, T. Würthinger, J. Eisl, D. Simon, and H. Mössenböck. 2018. Dominance-based duplication simulation (DBDS): code duplication to enable compiler optimizations. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*, J. Knoop, M. Schordan, T. Johnson, and M. F. P. O'Boyle (Eds.). ACM, 126–137. https://doi.org/10.1145/3168811

[39] B. Livshits. 2012. *Dynamic Taint Tracking in Managed Runtimes.* Technical Report MSR-TR-2012-114. Microsoft Research.

[40] S. Muchnick. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Francisco, Calif.

[41] R. Muth, S. A. Watterson, and S. K. Debray. 2000. Code Specialization Based on Value Profiles. In *Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1824)*, J. Palsberg (Ed.). Springer, 340–359. https://doi.org/10.1007/978-3-540-45099-3_18

[42] J. Newsome and D. X. 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/

[43] A. Prokopec, G. Duboscq, D. Leopoldseder, and T. Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, M. T. Kandemir, A. Jimborean, and T. Moseley (Eds.). IEEE, 164–179. https://doi.org/10.1109/CGO.2019.8661171

[44] 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-39 2006), 9-13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 135–148. https://doi.org/10.1109/MICRO.2006.29

[45] M. Rigger, R. Schatz, R. Mayrhofer, M. Grimmer, and H. Mössenböck. 2018. Sulong, and Thanks for All the Bugs: Finding Errors in C Programs by Abstracting from the Native Execution Model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar (Eds.). ACM, 377–391. https://doi.org/10.1145/3173162.3173174

[46] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *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

[47] L. Stadler, T. Würthinger, and H. Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, D. R. Kaeli and T. Moseley (Eds.). ACM, 165. https://dl.acm.org/citation.cfm?id=2544157

[48] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda. 2008. Automatic Network Protocol Analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society. https://www.ndss-symposium.org/ndss2008/automatic-network-protocol-analysis/

[49] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Cracow, Poland) *(PPPJ '14)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2647508.2647517

[50] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, A. Cohen and M. T. Vechev (Eds.). ACM, 662–676. https://doi.org/10.1145/3062341.3062381

[51] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. 2013. One VM to rule them all. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, A. L. Hosking, P. Th. Eugster, and R. Hirschfeld (Eds.). ACM, 187–204. https://doi.org/10.1145/2509578.2509581

[52] R. Zhang, S. Huang, and Z. Qi. 2011. Efficient Taint Analysis with Taint Behavior Summary. In *Third International Conference on Communications and Mobile Computing, CMC 2011, Qingdao, China, 18-20 April 2011*, D. Yuan, M. Cao, C. Wang, and H. Huang (Eds.). IEEE Computer Society, 11–14. https://doi.org/10.1109/CMC.2011.76