Dynamic Taint Analysis with Label-Defined Semantics

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 is a popular analysis technique which tracks the propagation of specific values while a program executes. To this end, a *taint label* is attached to these values and is dynamically propagated to any values derived from them. Frequent application of this analysis technique in many fields has led to the development of general-purpose analysis platforms with taint propagation capabilities. However, these platforms generally limit analysis developers to a specific implementation language, to specific propagation semantics or to specific taint label representations.

In this paper we present *label-defined dynamic taint analysis*, a language-agnostic approach for specifying the properties of a dynamic taint analysis in terms of propagated taint labels. This approach enables analysis platforms to support arbitrary adaptations to these properties by delegating propagation decisions to propagated taint labels and thus to provide more flexibility to analysis developers than other analysis platforms. We implemented this approach in *TruffleTaint*, a *GraalVM*-based taint analysis platform, and integrated it with GraalVM's language interoperability and tooling support. We further integrated our approach with GraalVM's performance optimizations. Our performance evaluation shows that label-defined taint analysis can reach peak performance similar to that of equivalent engine-integrated taint analyses. In addition to supporting the convenient reimplementation of existing dynamic taint analyses, our approach enables new capabilities for these analyses. It also enabled us to implement a novel tooling infrastructure for analysis developers as well as tooling support for end users.

CCS CONCEPTS

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

KEYWORDS

Dynamic Taint Analysis, GraalVM, TruffleTaint

MPLR '22, September 14–15, 2022, Brussels, Belgium
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9696-7/22/09.
https://doi.org/10.1145/3546918.3546927

ACM Reference Format:
Jacob Kreindl, Daniele Bonetta, Lukas Stadler, David Leopoldseder, and Hanspeter Mössenböck. 2022. Dynamic Taint Analysis with Label-Defined Semantics. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR '22), September 14–15, 2022, Brussels, Belgium.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3546918.3546927

1 INTRODUCTION

Dynamic taint analysis [31] is a program analysis technique in which a *taint label* is attached to specific data in order to dynamically track its propagation and detect its use at specific program locations. Such a taint label is often just a distinct flag, but may also be a custom-defined object. Dynamic taint analysis has seen applications in, among other fields, program vulnerability detection [7, 27, 29], reverse engineering [12, 35], debugging [11] and testing [8]. Because dynamic taint analysis is so widely applicable, analysis platforms with generic taint propagation capabilities emerged to support implementations of concrete applications of it [2, 15, 24, 25]. Such *dynamic taint analysis applications* select the properties of dynamic taint analysis according to their analysis goal. These properties include (1) which data to mark as tainted, (2) the representation of a taint label, (3) the semantics of taint propagation, and (4) where to check for tainted values and how to react to them there. However, current analysis platforms generally have limitations on which of these properties can be adapted by analysis developers.

In this paper, we present *label-defined dynamic taint analysis*, a new approach for implementing dynamic taint analysis. In this approach, the analysis platform provides the mechanics of attaching taint labels to values but delegates propagation decisions to the propagated taint labels. Our approach allows analysis developers detailed control over all taint analysis properties without requiring them to change the analysis engine. Any object can act as a taint label and can encode these properties by providing a particular interface. The analysis engine invokes functions of this interface with contextual information to request decisions for the label's propagation. We designed this interface to be language-agnostic with regard to both analysis implementation and targeted programs. While our approach is not limited to managed runtimes, it can particularly benefit from performance optimizations such runtimes are capable of.

We implemented label-defined dynamic taint analysis in *TruffleTaint*, a *GraalVM*-based platform with taint propagation capabilities for multiple languages [25]. By leveraging both performance optimizations that GraalVM is already capable of and speculative optimizations for taint propagation that we presented in previous work [26], TruffleTaint allows for label-defined taint analyses which achieve nearly the same performance as equivalent engine-integrated analyses. TruffleTaint can also support analysis implementations in arbitrary object-oriented languages and is integrated with GraalVM's tooling infrastructure to support such implementations, e.g., through debugging support. While performance and analysis complexity are conflicting goals, our benchmark results show that our approach supports both and leaves prioritizing betweem them to analysis developers. Based on TruffleTaint we reimplemented existing taint analysis applications, but our approach also enabled us to build novel tooling support for both analysis developers and end users.

1.1 Related Work

While many taint analysis platforms offer some degree of customization with regard to the properties of the dynamic taint analysis they implement, this customizeability can be severly limited. On some platforms only the sources of tainted values or the locations where tainted values are reacted to and the kind of reaction can be configured. Such platforms include *Minemu* [6], *LIFT* [30], *Ichnaea* [23], *TaintCheck* [29], and *jsflow* [18]. These analysis engines implement specific taint analysis applications well. However, to implement other taint analysis applications on these engines, analysis developers would need to change them. Analysis engines that support label-defined dynamic taint analysis instead allow these applications to be implemented without engine modifications.

The *LLVM DataFlowSanitizer* (DFSan) [2], *DECAF++* [13], *DyTan* [10] and *libDFT* [24] only support bit vectors of various sizes as taint labels and only support binary OR as logic for their merging. *Phosphor* [20] and *Taint Rabbit* [15] instead support custom-defined data structures as taint labels and custom logic for merging them. None of these platforms, however, allow control over the semantics of taint propagation, i.e., over when labels are merged or where they are propagated to. However, propagation semantics have a profound impact on the correctness and soundness of taint propagation, which has encouraged research into their improvement, e.g., by Slowinska et al. [32], Araujo et al. [5] and Hough et al. [19]. DFSan and DyTan offer select configuration options for their propagation semantics. DECAF++ and libDFT, which both target native code, enable analysis developers to manually specify the code that particular assembly instructions are instrumented with for taint propagation. While this enables analysis developers to arbitrarily change the propagation semantics, these engines still only support bit vectors as taint labels. Label-defined dynamic taint analysis both neccessarily supports custom-defined taint labels and provides analysis developers with full control over propagation semantics. It even enables the implementation of dynamic taint analysis applications that can be configured with a propagation semantics.

ALDA [9] is a specification language for dynamic analyses. It provides efficient shadow storage that can be used to implement arbitrary dynamic taint analysis applications. Similar to our approach, it takes advantage of joint compilation of analysis code and instrumented program, but requires explicit allocation and management of shadow storage. Our approach, in contrast, allows analysis specification in arbitrary object-oriented languages and to take advantage of rich tooling infrastructure available to these languages.

Similar to our approach, *Augur* [4] delegates propagation decisions to an analysis specification and provides context information for making these decision. However, Augur is designed specifically for JavaScript while we designed label-defined dynamic taint analysis to be language-independent. In contrast to Augur, we also encode analysis properties within taint labels rather than as a monolithic specification and we provide both performance optimizations and taint analysis applications that take advantage of this fact.

1.2 Contributions

This paper makes the following contributions:

- A language-agnostic approach for specifying the properties of a dynamic taint analysis within propagated taint labels (Section 4)
- An implementation of this approach in TruffleTaint and integration with the platform's performance optimizations, language support and tooling (Section 5)
- Adaptations of existing dynamic taint analysis applications and novel tooling infrastructure enabled by our approach (Section 6)

2 BACKGROUND

A taint analysis is characterized by its selection of *taint sources*, *taint labels*, *propagation semantics*, and *taint sinks*. A *taint analysis platform*, which we also refer to as *analysis engine*, provides taint propagation capability and implements *shadow storage*, i.e., separate memory for storing the taint labels of the values in the instrumented program's memory.

2.1 Properties of Taint Analysis

Taint sources and *sinks* are locations in a program that produce or consume values whose propagation the analysis needs to track. Taint sources are instrumented by the analysis engine to attach a taint label to any values originating from them. Common taint sources include functions that read data from the network, filesystem, or user input. However, a more specialized analysis may require more fine-grained access, e.g., to track the propagation of values resulting from an arithmetic overflow. Similarly, taint sinks are instrumented to perform certain actions when they consume tainted values. For example, *Neon* [38] taints sensitive user data and instruments system calls to automatically encrypt this data before transmitting it over a network, *TaintCheck* [29] can taint user input and instrument the *printf* function to abort execution before a user-provided format string could potentially trigger a code injection vulnerability, and *Penumbra* [11] taints data from different program inputs with different labels to report which inputs cause a software defect at a specific statement in a program. While many taint analysis platforms restrict taint sources and sinks to specific places, our approach offers greater flexibility by allowing for conditional taint sources based on run-time conditions. For example, one can implement a taint source that only marks values from arithmetic operations as tainted if they resulted from an overflow.

```
1    var a = /* tainted */;
2    var b = a + 1 // explicit data flow
3    var c = 1;
4    if (a == 0)
5      c = 2; // implicit data flow
```

Figure 1: Taint propagation example.

Taint labels are metadata attached to run-time values to mark them as *tainted*. Many analysis platforms only support single bits or fixed-size bit vectors as taint labels so that they can implement the merging of multiple labels efficiently using a binary OR. However, other platforms sacrifice this advantage in favor of arbitrary objects as taint labels with user-defined semantics for merging multiple taint labels. As shown in Section 5.1 shows that label-defined taint analysis can support both performance and analysis capability.

A *propagation semantics* defines the rules for propagating the labels of tainted values to values derived from them. For example, the result of a binary arithmetic operation, like the addition shown in line 2 of Figure 1, is usually tainted with a combination of its inputs' taint labels. However, *libdft*, for instance, extends this rule and defines the result of a subtraction as untainted if both operands are read from the same memory location and are therefore known to be equal [24]. Araujo et al. [5] similarly differ from common propagation semantics by not attaching the taint labels of memory references to values loaded via them. Some taint engines also consider *implicit data flows*, which arise from control flow, in their propagation semantics. In Figure 1 the value of b is tainted because it is computed from the tainted value stored in a. In contrast, there is no such *explicit data flow* from a to c. Instead, the value of c *implicitly* depends on the value of a because the assignment in line 5 is only executed if a holds a particular value. A purely dynamic taint engine can support implicit data flows by storing the taint labels of values used in control flow decisions and attaching them to all values produced in conditionally executed code [10].

2.2 Analysis Platform

GraalVM [37] is a polyglot virtual machine that can run programs implemented in, e.g., JavaScript or LLVM-based languages. It contains a framework, *Truffle*, for implementing language-specific runtimes. Truffle-based runtimes support language-level program instrumentation [14] and can interact to support interoperability between code of multiple languages in the same program [17]. GraalVM contains such runtimes for several languages as well as a debugger and a profiler based on Truffle's instrumentation support.

TruffleTaint is a GraalVM-based dynamic taint analysis platform. It leverages Truffle's instrumentation support provide capabilities for propagating taint at language-level in and between multiple languages [25]. TruffleTaint uses speculative optimization and dynamic compilation to reduce its performance overhead [26].

3 MOTIVATION

Since applications based on taint analysis can have specific requirements for particular analysis properties, a generic taint analysis platform ought to be flexible with respect to these properties. Nevertheless, most analysis platforms restrict their configurability due to performance concerns. For example, supporting only up to eight distinct taint labels enables DFSan [2] to represent such a label as a bit vector and thus to use a binary OR to merge labels efficiently. Similarly, DFSan and Taint Rabbit do not support implicit data flows. Such limitations preclude a platform from supporting taint analysis applications that require more taint labels or reduce their effectiveness for certain programs. Though analysis developers could extend these platforms, the need for work that is orthogonal to implementing analyses themselves would defeat the point of a generic taint analysis platform. Moreover, the need for platform versatility can outweigh the need for improved performance. For this reason, TruffleTaint avoids artificial limitations in favor of analysis capability support.

One such versatility is the analysis implementation language. Since TruffleTaint is built on top of a polyglot virtual machine and targets multiple languages we designed our approach for analysis specification to be language-agnostic with respect to both analysis implementation language and targeted language. Analysis developers may prefer to program in the language they target or in a high-level language more suited for rapid development. However, current taint analysis platforms typically restrict the analysis implementation language either to the language they are implemented in or to a high-level specification language.

We integrated TruffleTaint with GraalVM's tooling support to provide analysis developers with means to profile and debug their taint analysis application in the context of the program under analysis. While analysis developers can always use tooling for an analysis platform's implementation, such as low-level debuggers, doing so makes it harder to discern issues in the analysis code from unrelated implementation details of the platform. In contrast, tooling for the executed program may be oblivious to the taint analysis. However, issues in the analysis specification can become easily apparent if the analysis actions are inspected at the level of the program under analysis. In TruffleTaint we aim to support such investigations without exposing developers to implementation details of the platform.

Taint analysis poses unique challenges which can benefit from more specific tooling. One such challenge is designing a propagation semantics. Propagating taint over all possible data dependencies can lead to an overly excessive spread of taint labels that significantly impacts the usefulness of certain taint analysis applications. This problem is known as *overtainting* and is especially prone to occur when taint is propagated over implicit data flows. Conversely, ignoring certain data dependencies to avoid overtainting can lead to the analysis not detecting important data flows, which is referred to as *undertainting*. In practice, the effectivity of a particular propagation semantics depends on both the taint analysis application that uses it and the program it is used on. However, research that addresses these problems generally evaluates its approaches by just presenting improved results produced by specific taint analysis applications and specific programs. A more methodical approach to evaluation would require collecting detailed information about taint spread, which would make other approaches more comparable. The availability of such information would also aid analysis developers in selecting a suitable propagation semantics for their specific use case. We thus aimed at an approach for analysis specification that allows for taint analysis applications that can be configured with

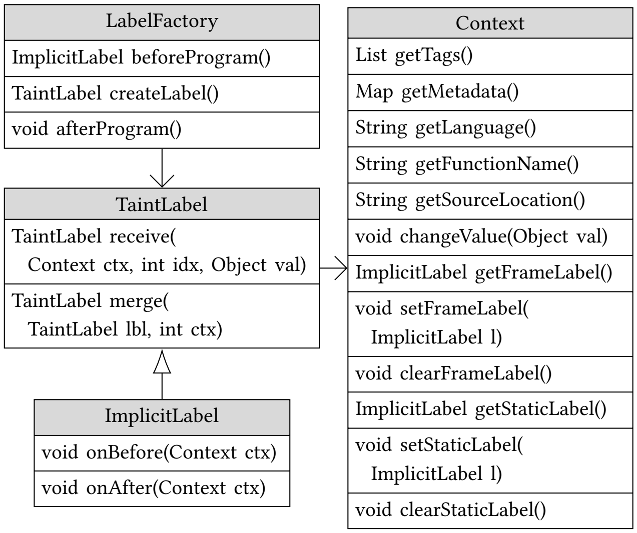


Figure 2: Interfaces for analysis specification.

```
class VanillaLabel:
  receive(ctx, idx, val): return this;
  merge(other, ctx): return this;
```

Figure 3: Taint label for basic taint tracking.

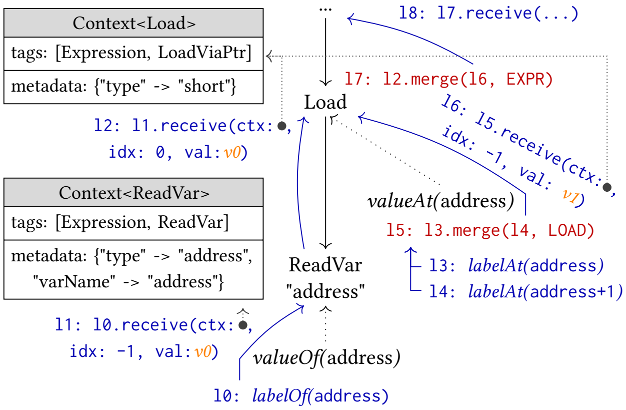


Figure 4: Label-defined taint propagation for the expression `... = *address;`, which loads a 2-byte value from a memory address stored in the local variable `address`. Intermediary labels are numbered in the order of their creation at runtime.

a propagation semantics. We also implemented tooling that can collect this information.

4 LABEL-DEFINED DYNAMIC TAINT ANALYSIS

In this section we describe our new, label-defined approach to specifying dynamic taint analysis applications. In this approach, analysis developers encode the properties of the desired taint analysis within the propagated labels. The analysis engine delegates propagation decisions to these labels by invoking specific methods on them with context information as arguments. By implementing these methods, analysis developers can make arbitrary changes to how these taint labels are propagated, define where taint labels are introduced and implement arbitrary taint sinks and sink actions. To enable this approach, we designed language-agnostic interfaces, which enable arbitrary objects to act as taint labels. In the following, we describe these interfaces, how analysis developers can use them to implement properties of a dynamic taint analysis, and how our approach allows to leverage composition and abstraction in analysis specifications.

4.1 Taint Propagation

Our approach requires taint labels to implement the `Taint- Label` interface shown in Figure 2. When a tainted value flows into an operation, its taint label is *received* in that operation, i.e., the analysis engine invokes that label's `receive` method with context information describing both the kind of input (through `idx` and `val` arguments) and the receiving operation (through the `ctx` argument). Analysis developers can implement *receive* to determine whether the operation the tainted value is received in constitutes a taint sink, if so, perform arbitrary taint sink actions. To adapt propagation semantics, analysis developers can implement `receive` to return different values based on this context information. If reception returns `null`, then the corresponding value is no longer considered tainted. `receive` can also return either the taint label itself, in order to continue its propagation, or return another taint label, e.g., to store additional data required for further taint propagation. When an operation produces a value, the analysis engine merges the taint labels received from the operation's tainted input values, if any, and attaches the resulting merged taint label to that value. The analysis engines delegates the logic for merging multiple taint labels to taint labels using their `merge` methods.

In the simplest case, a taint label is a primitive flag and it is propagated only through explicit data flows. Figure 3 shows the implementation of such a *vanilla dynamic taint analysis*, as it is sometimes called in literature [21, 22], in the `VanillaLabel` class. Figure 4 illustrates how an analysis engine invokes the `TaintLabel` methods during taint propagation when reading a value from a memory address which is stored in a local variable named `address`. The expression is shown as an *abstract syntax tree* (AST), which has the `Load` expression as a child of some other expression. `Load`'s only child expression, a read access to `address`, provides the memory address from which the target value is loaded. The Figure assumes that both the memory address and the value stored at that address are tainted with a `VanillaLabel`.

When nodes perform memory accesses, the analysis engine loads the taint labels of these values from its shadow storage. In Figure 4, the first loaded taint label is `l0`, which is the taint *label of* the memory address stored in `address`.[1]

When a tainted value flows into an operation, its taint label is received in that operation. Figure 2 shows three parameters which are provided by the analysis engine to the the `receive` method. The first argument, `ctx`, is a `Context` object, which acts as a bridge to the analysis engine. It can be used to query information about the operation and to produce certain side effects. The second argument, `idx`, describes to which input of the operation the taint label is attached to. The third argument, `val` contains the value to which the taint label is currently attached. Analysis developers can change the semantics of taint propagation by returning different taint label implementations based on this provided information.

Analysis engines using label-defined dynamic taint analysis define an indexing scheme that assigns a unique number to each kind of input for a particular kind of operation. This scheme enables analysis developers to reliably discern how an operation will use a particular input value when it is received with a particular `idx`. For example, in a binary arithmetic operation indices 0 and 1 always denote the left and right operand, respectively, and in a memory store operation index 0 denotes the target memory address while index 1 denotes the value to store. An `idx` of -1 always indicates that the tainted value was read from memory by the operation itself rather than being provided to it by a subexpression. In Figure 4 the read access to the variable `address` loads the memory address from the variable `address`, i.e., from memory. `l0` is thus received with an `idx` of -1 in the `ReadVar` expression. This reception results in `l1`. Since no further tainted values flowed into `ReadVar`, `l1` is directly applied to `v0`, i.e., the memory address read from `address` and provided to `Load`. Since `ReadVar` is the only child of `Load`, the address’s taint label is *received* with index 0 in the `Load` expression, which results in `l2`.

`Load` reads a 2-byte value, `v1`, from a memory that is addressable at byte-level. Since shadow storage needs to store taint labels at the granularity of addressable memory, this read access requires the analysis engine to load the taint labels for each of these bytes, which are shown as `l3` and `l4`. To derive a single taint label of a multi-byte value, the analysis engine invokes `merge` on the taint label of the first tainted byte with that of the second tainted byte as argument. This is repeated with the aggregated labels as receivers until the taint labels of all tainted bytes are consumed. `l3` thus receives the `merge` method call with `l4` as argument and returns `l5` as the aggregated taint label. In order to reduce the number of intermediary taint labels, only the merged taint label is received in the expression loading the value.[2] Thus `l6` is received from `l5` while `l3` and `l4` are not received.

Taint labels of input values to an operation are merged in the order of the *idx* they were received with. If a value loaded from memory contributes to a data flow, the analysis engine merges its received taint label into those of the other inputs. In Figure 4, `l6` is thus passed as an argument to `l2`’s `merge` method. This merge produces `l7`, which is finally attached to the value returned from `Load` and received as `l8` in `Load`’s parent expression. Taint analyses that use multiple kinds of taint labels cannot predict which of them will be the receivers of the `merge` method invocations, so they must ensure that each label combination’s `merge` semantics is commutative.

```
class PointerLabel extends VanillaLabel:
  receive(ctx, idx, val):
    if (idx == 0):
      if (ctx.getTags().contains('Load')
       || ctx.getTags().contains('Store')):
        return null;
    return this;
```

Figure 5: Propagation semantics following Araujo et al. [5].

While the first argument to `merge` is the taint label to merge with, the second argument describes the context in which the taint labels are merged. Some taint analyses for low-level languages merge taint labels of multiple bytes of the same value differently than taint labels of separate values [15]. The analysis engine invokes `merge` with different numeric values for the `context` argument in these cases, as can be seen in the two merges in Figure 4, so a propagation semantics implemented on this engine can make this distinction. Which merge contexts exist depends on the operations of the instrumented language. For example, an assignment operation to an array at a particular index in JavaScript produces two data flows. The first one propagates the taint labels of the array object, array index and value to store to the value stored within the array at that index. The second data flow propagates the taint labels of the same values to the value returned from the assignment expression. Since these are separate data flows, the analysis engine needs to merge the taint labels that contribute to the corresponding values separately and provide according merge context values to enable propagation semantics implementations to distinguish these cases.

4.2 Context-Aware Propagation Semantics

While some analysis engines already enable analysis developers to define custom taint label representations and to specify *how* such taint labels are to be merged, our approach additionally enables analysis developers to define *which* taint labels should be merged. To this end, our approach provides the `receive` method for taint labels, which provides detailed context information based on which analysis developers can execute the propagation semantics they require. This context information is provided in the form of arguments to `receive` and contains details of both the operation itself and the context it is used in.

Figure 4 illustrates how `Context` objects represent details for the operations used in our running example in the form of semantic *tags* and additional *metadata*. *Tags* are literals that define the *kind* of the operation. All tags that apply to the current operation can be queried using the `getTags` method. Tags can represent abstract information such as whether the operation is a value-producing *Expression* or a *Statement* that only consumes values for languages that make

[1] In our approach, shadow storage is not directly accessible to analyses. To explicitly set the taint label of the value stored at a particular memory location, they can execute instrumented code of the analysis’ target language from the propagation semantics.

[2] If needed, one could change this semantics using a special *AggregatedLabel* that is returned from merges in such cases and internally stores the taint labels of the individual bytes. When this label is eventually *received*, it would relay this reception to its stored labels, merge the such received labels and finally receive the resulting taint label.

such a distinction. More importantly though, the analysis engine must provide a specific tag for each kind of operation defined by the targeted language. These tags, which may be language-specific, enable analysis developers to adapt taint propagation on the level of individual language features. Some operations require additional information to fully describe their semantics. This information is provided in the form of named metadata through the mapping returned from getMetadata. Figure 4 exemplary shows that for operations of a typed language that are tagged *Expression*, the *type* of that operation's return value can be provided, while for an access to a local variable additionally the *name* of that variable be can provided. We used this approach of describing operations, which was taken from Van de Vanter et al. [14], already in our previous work for implementing a multi-language dynamic taint analysis engine with a fixed, language-level propagation semantics for each language [25].

While the idx and val arguments to receive provide details on a specific taint propagation, the context argument also provides details on the context of the operation that receives the taint propagation. This context includes the name of the function and the location in source code where the operation is used. It also includes the implementation language of the operation in order to enable the implementation of language-specific propagation semantics on a multi-language analysis engine.

The context information provided to the receive method enables detailed modifications to the respective taint labels' propagation semantics. For example, Figure 5 shows the code of a receive method that uses the provided context information to implement the propagation semantics optimization that Araujo et al. [5] proposed, i.e., disregarding the taint label of the memory address in memory load and store expressions. Both memory loads and memory stores receive the memory address to access as their first argument, which is denoted by an idx of 0 in our approach. The code in Figure 5 thus compares the idx argument to 0 and searches the context object's list of tags for ones that indicate a memory load or a memory store in order to determine whether the tainted value is currently propagated as the memory address argument for a memory access. If so, the code returns null so that the analysis engine considers the memory address to not be tainted and thus does not propagate the address's taint label to a value being stored to memory or being loaded from memory. Otherwise, i.e., for all other explicit data flows, the taint label returns itself so that it continues to be propagated. If the expression shown in Figure 4 propagated only taint labels of this kind, the reception of l1 with index 0 in the Load operation, which is tagged as *Load*, would return null. Therefore, l6 would be the only taint label affecting the data flow in that operation and, because of this, would be propagated without further taint label merging. Likewise, in an expression that stores a value to memory the taint label of the memory address could not spread to the stored value.

4.3 Taint Propagation in Implicit Data Flows

To support taint analyses that require instrumentation beyond explicit data flows, we define *implicit labels*. As Figure 2 shows, implicit labels are an extension to regular taint labels that can be *set* using Context objects as either *frame label* or *static label*. Once an implicit label is set, it is notified whenever an expressions starts or ceases to be executed. It is furthermore received in all expressions that produce values. Such received labels are then attached to these values or, if the values were already tainted, merged with their taint labels. Frame labels are set in the scope of the currently executing function and therefore have no impact on that function's callers or callees. Static labels, in contrast, apply to the whole program and remain active until they are either cleared or replaced. Implicit labels are received with an idx of either −2 or −3, depending on the scope they are set in. When set implicit labels are merged, they are always the receivers of the merge method calls.

Implicit labels can be used to implement taint propagation over implicit data flows. The generic algorithm for doing so is as follows. (1) When an expression that changes control flow does so based on a tainted condition value, an implicit label is set. Since this implicit label will be received in all subsequent expressions, all values which are propagated because of that condition value will be tainted, which accounts for the implicit data flow. (2) When execution reaches code that would be executed regardless of the control flow decision that depended on a tainted condition value,

```
class Implicit extends VanillaLabel:
  constructor(src): this.src = src;
  receive(...): return new Explicit();
  onBefore(ctx): {}
  onAfter(ctx):
    if (sameSource(ctx, this))
      ctx.clearFrameLabel();
class Explicit extends VanillaLabel:
  receive(ctx, idx, val):
    if (isBranch(ctx, idx)
        && !ctx.getFrameLabel())
      ctx.setFrameLabel(new Implicit(
        ctx.getSourceLocation()));
    return this;
```

Figure 6: Propagation semantics for implicit data flows.

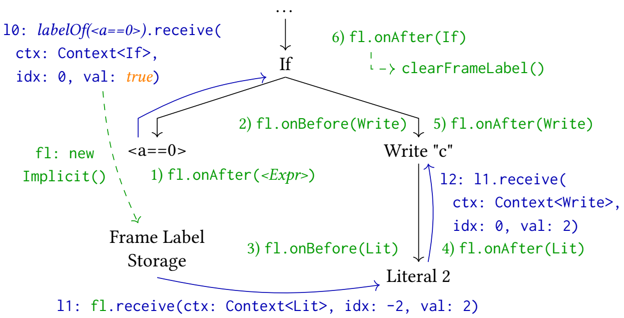


Figure 7: Propagation of the taint labels shown in Figure 6 for the expression if (a==0) { c = 2; }. Propagation of Explicit label is shown in blue. Propagation of Implicit is shown in green, with numbers indicating execution order.

the implicit label is cleared since the implicit data flow has thus ended. Figure 6 sketches an implementation of this algorithm for a programming language with only structured control flow. Figure 7 illustrates how this implementation works using the implicit flow example from Figure 1. In that Figure, the `If` expression receives a condition value from its left subexpression and either executes an assignment of the constant 2 to the variable `c` or not, depending on that condition value.

In the propagation semantics of Figure 6, taint labels of type `Explicit` are propagated over explicit data flows. To implement step (1) of the algorithm mentioned above, they detect when they are being received as the condition value for a control flow branch and, if so, set a new `Implicit` as frame label. The first taint propagation in Figure 7 is the reception of the condition value's taint label in the `If` expression. Since the conditions are met, that taint label sets a new frame label, which is called `fl`. `fl` is then notified that the expression `<a==0>` has finished executing.

Figure 7 shows the tainted condition value as `true`, hence the conditional assignment is executed. Since our example assumes a programming language with only structured control flow, the conditionally executed code can be represented as a proper subexpression of `If`. To implement step (2) of the algorithm mentioned above each implicit label removes itself as frame label after the expression that originally set it as such, which it identifies by its source code position, has finished its execution, since that implies that the conditionally executed code has also finished its execution. Figure 7 shows this removal in the sixth invocation of an execution event notification. `Explicit` does not overwrite already set implicit labels because in a language with only structured control flow nested control flow decisions cannot extend the scope of an implicit flow.

Our approach does not require implicit labels to be separate from regularly propagated taint labels. However, if `Implicit` could be propagated and thus merged with other labels from outside the current function, the additional data it needs to store would require more complicated logic in `merge`. To avoid the associated decrease in run-time performance, `Implicit` instead returns an `Explicit` in its `receive` method. `fl` is thus received as an `Explicit` in Figure 7's `Literal` expression and is propagated as such to `Write`.

Label-defined dynamic taint analyses can also support taint propagation over implicit flows for languages with unstructured control flow. In such languages, control flow expressions can arbitrarily dispatch between code blocks rather than containing the conditionally executed code as a subexpression. Dytan uses a postdominator tree to detect the code block at which control flow merges again after a conditional branch and removes its equivalent of an implicit label just before that block is executed [10]. To support this concept, an analysis engine that supports label-defined taint analyses needs only provide block identifiers as metadata for *Code Block* expressions and the according merge block's identifier as metadata to each control flow expression.

We present this semantics for propagating taint over implicit data flows only to show that it can be achieved using our approach to taint analysis specification. More advanced programming languages such as JavaScript and LLVM IR typically contain multiple kinds of expressions that affect control flow in different ways. Supporting such a language requires implementing special cases for each of these expressions in potentially both implicit labels and regularly propagated taint labels. However, as we have shown, label-defined dynamic taint analysis allows fine-grained control over taint propagation across explicit data flows as well as receiving execution events beyond explicit data flow. Combined, these capabilities enable analysis developers to support the effects of any kind of control flow in their propagation semantics. More importantly, our approach does not restrict analysis developers to a single semantics for implicit flows. Decisions such as whether a tainted loop condition value causes an implicit data flow to all subsequent loop iterations or only to the next one can be modeled in separate propagation semantics without having to adapt the analysis engine itself.

```
class SourceLabel extends VanillaLabel:
  receive(ctx, idx, val):
    if (ctx.getTags().contains('Return') &&
        ctx.getFunctionName() == '...')
      return new VanillaLabel();
    return null;
  onBefore(ctx) {} onAfter(ctx) {}
```

Figure 8: Programmatically defining the return value of a particular function as a taint source.

4.4 Taint Sources and Taint Sinks

The `LabelFactory` interface, shown in Figure 2, is used to implement taint sources. An analysis engine that supports label-defined dynamic taint analysis would receive a script or a path to a script that returns an object implementing this interface. If this analysis engine provides a function to instrumented programs with which they can manually taint a value, the analysis engine creates the taint labels for these values using the factory's `createLabel` method. Such a function is required for analysis engines targeting languages which offer taint tracking as a feature, such as older versions of Ruby [34]. `LabelFactory` also provides methods which enable analyses to perform setup and cleanup actions before and after the instrumented program is executed.

The label factory's `beforeProgram` method is invoked by the analysis engine and the implicit label returned from that invocation is set as static label before the instrumented program starts to be executed. This can be leveraged to implement arbitrary taint sources. For example, consider an analysis specification which returns an object of the `SourceLabel` class shown in Figure 8 in `beforeProgram`. Set as static label, this label would then be received in all executed program expressions and thus attached to all values produced by them. However, `SourceLabel` implements its reception to only return a taint label for the return value of a particular function. Thereby this function is instrumented as a taint source. The same approach of using static labels enables arbitrary run-time conditions to be defined as taint sources. For example, to define the occurrence of an integer overflow as a taint source the implicit label would return itself if received as static label, but implement its `merge` method to never replace an existing label. The reception of this implicit label in an arithmetic operation returns a

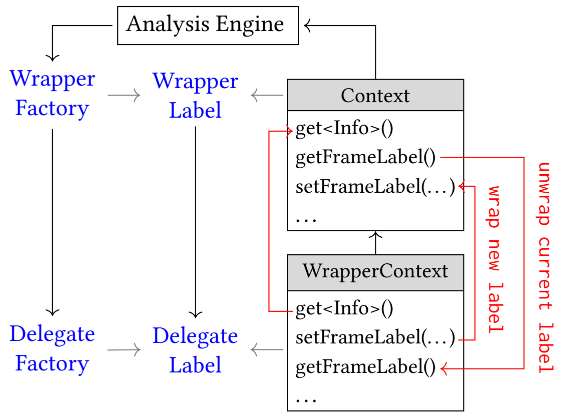


Figure 9: Delegation of concerns via taint label composition.

new *ValueLabel*, which stores the operand and the kind of operation, while other receptions return null. With this strategy, all operands to an arithmetic operation are tainted. If ValueLabel is merged with another ValueLabel, it uses the information both labels store to check whether an overflow occured. If so, the merge results in a regular taint label, otherwise it results in null. In this system, both ValueLabel and regular taint labels return the regular label if only one of the merge operands is a ValueLabel, which happens when an already overflown value flows into another arithmetic operation. Thus, ValueLabel is not propagated outside of arithmetic expressions but causes a regular taint label to be attached to all values created from an arithmetic overflow. Figure 11 in Appendix A sketches an implementation of this concept.

Each taint label can implement its receive method to detect the run-time conditions it considers to be a taint sink and perform arbitrary actions there. For example, the taint label could check if it is being read from the actual arguments of a sensitive function, such as C's printf function, and terminate the program like *TaintCheck*. To implement *Neon*'s automatic encryption of sensitive data before the network boundary the taint label could detect when it is received as an argument to an according system call and use the Context interface's changeValue function to replace the value it is attached to with an encrypted one.

4.5 Analysis Definition through Composition

In addition to supporting precise and flexible definition of taint analysis components, label-defined dynamic taint analysis enables composition as a tool to both simplify and enrich taint analysis implementations. Since label-defined analysis implementations are object-oriented, inheritance simplifies the implementation of new analyses by reusing parts of existing ones. We have done this in Figures 5, 6 and 8, where we extended VanillaLabel with various new semantics. One could similarly extend an existing taint label that provides a specific propagation semantics with additional taint sinks or additional information to propagate. Furthermore, existing taint label implementations can be reused as part of new analyses. As an example of this, in Figure 8 we defined a new taint analysis with a particular taint source, but reused VanillaLabel as the actual label to propagate.

Since label-defined taint analyses have a known interface, one analysis implementation can *wrap* another. Figure 9 illustrates this concept, which follows the object-oriented design pattern of *decorators* [16]. The Figure shows that a *wrapper analysis* consists of a label factory and taint labels which each refer to a separate *delegate analysis*. The wrapper analysis' factory and labels delegate all invocations of the methods of their respective interfaces to these *delegates*. In this manner, the wrapper labels and factory essentially instrument the taint analysis implemented by the delegates. For this instrumentation to be sound, all taint labels produced by the wrapped taint analysis must also be wrapped, so that the analysis engine only propagates WrapperLabel. This wrapping is done by the wrapper labels and factory in two ways. First, they wrap the labels returned from delegated method calls. Second, they wrap the Context object provided to the delegated method calls. Such a WrapperContext delegates most methods to the context object provided to the WrapperLabel by the analysis engine, but wraps all implicit labels set through that context. In contrast to soundness, all taint labels passed to or accessed by the delegate analysis must be unwrapped in order to not interfere with the implementation of the delegate taint labels, which may involve dynamic type checks. For this reason, wrapper labels unwrap the labels passed to the merge method and wrapper contexts unwrap implicit labels before providing them to the delegate labels. We provide an implementation of such a *wrapper analysis* in Figure 12 of Appendix A.

Wrapper analyses have multiple applications. One application is the implementation of *configurable dynamic taint analyses*, that is, applications of dynamic taint analysis that can be configured with a propagation semantics. Here, the wrapper analysis would consume the propagation semantics implemented by a separate taint analysis but track additional information or implement specific taint sources or sinks. Some analysis engines, e.g., DFSan [2], provide run-time flags to configure some aspects of the semantics with which taint labels are propagated. Configurable dynamic taint analyses using our approach provide more flexibility since they are not limited by predefined configuration choices.

Wrapper analyses can also be used to add new features to the analysis engine. For example, the ability to run multiple taint analyses in parallel for the same program execution, as it is supported by ALDA [9], can be implemented using a wrapper analysis that delegates to multiple analyses rather than just one. Such a *combined analysis* assigns a unique identifier to each delegate analysis. The *combined labels* it propagates may contain any number of delegates and ensure isolation between the analyses when delegating method calls. For example, they implement merge to only merge delegate labels with the same identifier and include any delegate labels that have no suitable merge partner in the resulting combined label without merge. They also implement context wrappers such that each analysis can only retrieve the implicit labels it set itself and that multiple implicit labels can be set in parallel. We sketch an implementation of such a combined analysis in Figures 13 and 14 of Appendix A.

In the following sections, we will provide additional uses for the wrapper analysis concept.

5 IMPLEMENTATION IN TRUFFLETAINT

We implemented label-defined dynamic taint analysis in *TruffleTaint*, our GraalVM-based dynamic taint analysis platform. In previous work we presented TruffleTaint's capability to propagate taint in and across multiple programming languages [25] and how we leveraged speculative optimization and dynamic compilation to reduce TruffleTaint's run-time overhead [26]. Label-defined dynamic taint analysis is uniquely suited to managed runtimes as it allows for both leveraging the managed language with its associated tooling and for taking advantage of performance optimizations the managed runtime is already capable of. It both complements TruffleTaint's language support and integrates well with TruffleTaint's performance optimization strategies. The result is a highly configurable, polyglot dynamic taint analysis platform which enables analysis developers to choose between priorities such as analysis performance, analysis complexity and tooling support.

5.1 Performance Optimizations

Since GraalVM and TruffleTaint are implemented in Java, we also chose Java as the principal implementation language for taint analyses on our platform. To execute programs, GraalVM language runtimes parse them into an abstract syntax tree whose nodes are implemented as Java objects. Instrumentations such as TruffleTaint are also inserted as nodes into these *Truffle ASTs*. Truffle is integrated with the GraalVM JIT compiler, a dynamic compiler for Java, so that these ASTs can be dynamically optimized and compiled, including inserted instrumentation. For this reason, taint analyses implemented on TruffleTaint can reach performance close to that of fully engine-integrated analyses.

Taint analysis engines often restrict their configuration options in favor of performance. However, by taking advantage of dynamic optimizations typically performed by managed runtimes that target object-oriented languages, the configurability afforded by label-defined taint propagation need not lead to poor performance. The performance impact of a taint analysis rather depends on the complexity of the analysis and the quality of its implementation. We implemented the `VanillaLabel` from Figure 3, `PointerLabel` from Figure 5 and a wrapper analysis in Java. Table 1 shows the increase in normalized run time of each of these label-defined analyses over an engine-integrated vanilla dynamic taint analysis for the benchmarks we used in our earlier work on TruffleTaint's performance[3]. More details on our benchmarks and further results can be found in Appendix B. In the following, we explain these results and which optimization strategies we used to achieve them.

Managed runtimes often use *inline caches*, i.e., at a particular call site of an interface method they observe the concrete types of the objects whose methods are called and store a fast path to the implementations of those methods for the most frequently observed types. During dynamic compilation, if an inline cache contains only one entry then its targeted method can be inlined. If it is, then the type of label returned from the inlined `TaintLabel` method may become known to the compiler, which can thus inline that type's methods that are called during further taint propagation. To take advantage of such optimizations, we implemented our interfaces for TruffleTaint as *Truffle libraries*, which is a feature of GraalVM's *Truffle* framework that enables the definition of interfaces and the automatic optimization of calls to their implementations using, for example, inline caches [3]. Due to this optimization, our benchmark results in Table 1 show that for most benchmarks a label-defined vanilla dynamic taint analysis increases the benchmark's normalized run time by less than 20% compared to an equivalent engine-integrated taint analysis, although the increase in run time can be considerable for other benchmarks.

The context information provided through `ctx` and `idx` parameters in `TaintLabel` methods is constant at each call site. During dynamic compilation, it can thus be *constant-propagated* [28] and can enable *partial evaluation* [36], i.e., simplification of inlined propagation semantics based on known operands. This can, in turn, enable other optimizations such as *partial escape analysis* [33], which can optimize away the allocation of short-lived intermediate `Context` and `TaintLabel` objects. As a result of such optimizations, the benchmark results of Table 1 show that a label-defined taint analysis which adapts its propagation based on context information as shown in Figure 5 can incur almost the same slowdown as a `VanillaLabel`[4].

The column *WrapperLabel* of Table 1 shows that a wrapper analysis which delegates to `VanillaLabel` usually increases the normalized benchmark run time up to 3 times as much as the `VanillaLabel` alone does, though higher increases are also possible. This illustrates the tradeoff between performance and functionality. The advantage of label-defined taint propagation, however, is that it leaves the choice between these priorities to analysis developers. Conversely, engine implementers can focus on performance optimizations, such as fast and efficient shadow memory, without having to care about analysis semantics.

While our benchmark results show dynamic compilation and common optimizations to be effective at improving the performance of label-defined taint analyses, their effectivity depends on the compiler's ability to inline the propagation semantics. The more complex that semantics is, the faster its repeated inlining consumes the compiler's *inlining budget*, i.e., the maximum amount of code allowed for inlining. The same problem applies to other approaches that compile the propagation semantics, such as ALDA and Phosphor. In earlier work, we described how TruffleTaint enables the dynamic JIT compiler to optimize away instructions for taint propagation in program parts that are not reached by tainted data [26], which mitigates this problem by reducing the amount of code to compile. We reused this optimization technique to optimize away instructions for attaching implicit labels as long as none have been set. As a result, TruffleTaint incurs almost no slowdown as long as no taint labels are propagated, regardless of the propagation semantics' complexity. Due to space restrictions, the benchmark results that show this can be found in Appendix B in Table 2. Label-defined

[3] Available at https://github.com/jkreindl/taint-benchmarks

[4] Since array objects in Java lack a `contains` function, our Java implementation of the taint label shown in Figure 5 instead uses a loop over the context object's array of tags to determine whether the array contains a particular tag. However, in some instances, the GraalVM JIT compiler would not automatically unroll this loop even though that array and its elements were known constants during compilation. Not doing so resulted in the compiler not being able to evaluate such determinations at compiletime. In order to ensure that these determinations could be fully evaluated during compilation whenever possible, we annotated this loop with a Truffle compiler directive forcing its unrolling in inlined propagation semantics.

Table 1: Increase in normalized runtime of various label-defined propagation semantics vs. vanilla dynamic taint analysis hardcoded in the analysis engine.

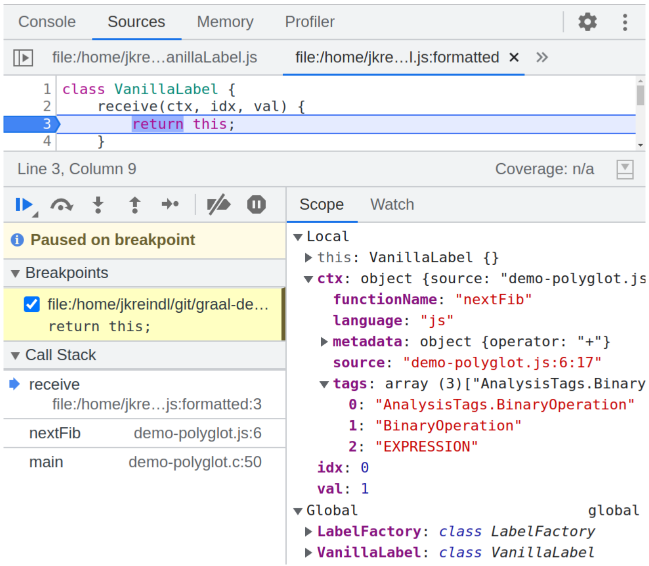
Language	Benchmark	Vanilla	Vanilla Label		Pointer Arg Label		Vanilla Wrapper Label	
		Engine	Runtime	Change	Runtime	Change	Runtime	Change
C/C++-JS	BinaryTrees	1.59	1.68	+5.6%	1.65	+3.8%	1.70	+6.9%
C/C++-JS	Fannkuch	1.93	2.68	+38.8%	2.66	+37.7%	3.17	+64.1%
C/C++-JS	Fasta	1.05	1.06	+1.0%	1.07	+1.9%	1.09	+3.8%
C/C++-JS	Mandelbrot	3.25	4.69	+44.2%	4.72	+45.2%	7.04	+116.4%
C/C++-JS	NBody 1	4.52	5.00	+10.6%	4.98	+10.2%	5.47	+21.0%
C/C++-JS	NBody 3	1.71	1.75	+2.3%	1.75	+2.3%	2.02	+18.1%
C/C++-JS	NBody 2	1.58	1.93	+22.2%	2.09	+32.4%	2.49	+57.8%
C/C++-JS	Pidigits	0.99	0.99	+0.0%	0.97	-2.0%	1.00	+1.0%
C/C++-JS	ReverseComplement	1.64	1.63	-0.6%	1.93	+17.7%	1.99	+21.3%
C/C++-JS	SpectralNorm	1.49	1.83	+22.8%	1.98	+32.9%	2.61	+75.1%
C/C++	BinaryTrees	1.58	1.65	+4.4%	1.68	+6.3%	1.77	+12.0%
C/C++	Fannkuch	2.01	3.01	+49.8%	2.80	+39.4%	4.51	+124.6%
C/C++	Fasta	1.71	1.80	+5.3%	1.80	+5.3%	1.94	+13.4%
C/C++	Mandelbrot	2.47	3.62	+46.6%	3.59	+45.4%	5.61	+127.2%
C/C++	NBody 1	2.14	3.53	+64.9%	3.47	+62.1%	4.27	+99.4%
C/C++	NBody 3	1.13	1.34	+18.6%	1.34	+18.6%	1.71	+51.4%
C/C++	NBody 2	1.99	2.33	+17.1%	2.32	+16.6%	2.80	+40.6%
C/C++	Pidigits	6.53	6.98	+6.9%	7.64	+17.0%	11.19	+71.4%
C/C++	ReverseComplement	1.93	1.94	+0.5%	1.96	+1.6%	1.85	-4.1%
C/C++	SpectralNorm	1.24	2.18	+75.5%	2.21	+77.9%	2.88	+131.8%
JS	BinaryTrees	2.31	2.64	+14.3%	2.69	+16.4%	2.77	+19.9%
JS	Fannkuch	2.30	2.95	+28.2%	2.89	+25.6%	3.46	+50.4%
JS	Fasta	1.50	1.51	+0.7%	1.50	+0.0%	1.71	+14.0%
JS	Mandelbrot	2.37	4.15	+75.0%	4.31	+81.7%	7.54	+217.8%
JS	NBody 1	1.87	6.52	+248.7%	6.46	+245.5%	14.55	+678.3%
JS	NBody 3	1.82	3.06	+68.1%	3.07	+68.7%	4.29	+135.7%
JS	NBody 2	2.20	2.70	+22.7%	2.69	+22.3%	2.10	-4.5%
JS	Pidigits	0.99	1.01	+2.0%	1.03	+4.0%	1.01	+2.0%
JS	ReverseComplement	4.65	5.48	+17.8%	5.27	+13.3%	9.94	+113.7%
JS	SpectralNorm	1.00	1.11	+11.0%	1.11	+11.0%	1.88	+88.1%

taint analysis further complements our previous optimization strategy since propagation semantics can only be inlined in code that is reached by tainted values. For our benchmarks we still doubled the GraalVM JIT compiler's inlining budget. While a separate inlining policy for taint analysis could mitigate the code size issue further, we consider tuning a compiler's optimization heuristics for specific programs or taint analyses to be orthogonal to this paper.

5.2 Polyglot Features

Label-defined taint analyses can be implemented in any programming language that supports object-orientation. Using the analysis engine's implementation language for analysis specifications allows for using that engine's APIs from these specifications. For example, on TruffleTaint, analysis developers could use Truffle's compiler directives for performance improvement. Besides this, analysis engines will typically provide an API for executing code of their targeted language. This API can be used to implement a wrapper analysis which delegates to objects of that language. Such analyses require frequent crossing of the language boundary, which can significantly reduce performance. However, this approach gives analysis developers the option of using the targeted language for analysis specification, which can provide benefits such as tooling support for development. We implemented such a wrapper analysis on TruffleTaint to take advantage of GraalVM's extensive language support and its associated tooling.

Figure 10 illustrates the capability of the *polyglot wrapper analysis* we implemented on TruffleTaint. It shows GraalVM's language-agnostic debugger suspended at a breakpoint in the `receive` method of a `VanillaLabel` implemented in JavaScript. In addition to that method, the call stack also contains the executing methods of the instrumented program, enabling analysis developers to debug a taint analysis both in the context of and at the same level as the program under analysis, regardless of which languages either is implemented in. We further specifically integrated TruffleTaint with GraalVM's debugging support such that, even though the taint label of a value is transparent for the program under analysis, tainted values are displayed as such by the debugger and their taint labels can be inspected as well. We used this debugging support extensively during the implementation of label-defined taint analysis. It significantly simplified diagnosing propagation issues both of the



```
class VanillaLabel {
    receive(ctx, idx, val) {
        return this;
    }
```

☑ file:/home/jkreindl/git/graal-de…
return this;

receive	file:/home/jkre…js:formatted:3
nextFib	demo-polyglot.js:6
main	demo-polyglot.c:50

```
▼Local
 ▶this: VanillaLabel {}
 ▼ctx: object {source: "demo-polyglot.js
    functionName: "nextFib"
    language: "js"
   ▶metadata: object {operator: "+"}
    source: "demo-polyglot.js:6:17"
   ▼tags: array (3)["AnalysisTags.Binary
      0: "AnalysisTags.BinaryOperation"
      1: "BinaryOperation"
      2: "EXPRESSION"
  idx: 0
  val: 1
▼Global                                global
 ▶LabelFactory: class LabelFactory
 ▶VanillaLabel: class VanillaLabel
```

Figure 10: Integrated debugging of polyglot taint analyses and polyglot applications using TruffleTaint.

engine and in propagation semantics we implemented in JavaScript on top of it, and also proved invaluable during the implementation of taint analysis applications such as the ones we present in Section 6. Besides debuggers, a polyglot wrapper analysis also gives analysis developers access to other useful tooling such as profilers or code coverage instruments.

6 APPLICATIONS

In this section we highlight some applications of dynamic taint analysis that can benefit from being implemented using our label-defined approach.

6.1 Existing Application

Label-defined taint analysis allows the reimplementation of even complex existing applications of dynamic taint analysis. An example of such an analysis is *Penumbra* [11], which tracks the program locations from which tainted values originate and prints these locations when those values flow into a taint sink. By defining taint sinks as program locations at which faults are known to occur, Penumbra gives a list of sources that contribute to a particular fault. The original Penumbra targets native code and instruments various kinds of program inputs as taint sources. We implemented Penumbra's core logic in JavaScript in a PenumbraLabel, which is provided in Figure 15 in Appendix C. This label propagates a set of source locations, which it initializes with the location of its first reception. Since we wanted our implementation to be language-independent we did not implement language-specific taint sources like the original Penumbra, but rather rely on user-level functions to explicitly declare taint sources (through LabelFactory's createLabel) and sinks (through checking for reception as argument in an arbitrary specific function) in the instrumented program. Note that PenumbraLabel could be reused in other language-specific taint analyses which do implement label-defined taint sources. Similarly, the original Penumbra inherits support for implicit flows from the analysis engine it is implemented upon. Such support is necessarily language-specific, thus we implemented PenumbraLabel as a wrapper label which can be configured with a propagation semantics. However, we only tested this implementation using a VanillaLabel as a delegate. We also implemented a full information flow analysis, which tracks the full propagation path of a tainted value in addition to its sources. Its JavaScript implementation is provided in Figure 16 in Appendix C. LLVM DFSan provides similar path tracking functionality, but this functionality is integrated into the engine, separate from its general taint propagation and supports only limited configuration of its propagation and semantics.

6.2 Taint Profiling Infrastructure

Label-defined taint analysis enables instrumentation of taint analyses through wrapper analyses. Based on such instrumentation, we implemented a taint profiling infrastructure. This infrastructure records the frequency of taint propagation per program operation as well as, optionally, how often each operation was executed. This information provides deep insights into the behaviour of a particular taint analysis for a particular program execution, which are useful in a variety of ways. For example, engine developers can use them to detect the kinds of expressions for which taint is propagated most frequently and which would thus be the most beneficial targets for optimizations. Using our profiling infrastructure, we found that for most of our C/C++ benchmarks, the largest subset of propagations pertained to LLVM's *Phi* instructions and thus paid particular attention that they were not a performance bottleneck. Similarly, the collected data provides a metric for taint spread in the number and ratio of operations in which taint propagation occurred, i.e., that were reached by tainted data. When changing a propagation semantics, for example with the goal of avoiding excessive taint spread, analysis developers can use this metric to evaluate the effects of that change and can both assess and empirically argue its suitability. Analysis users may similarly care to know to which parts of an instrumented program tainted values spread. If the used analysis unexpectedly fails to propagate taint to a particular place, this information can help diagnose where the taint was lost. We used this tool in concert with another instrumentation label that records the invocation and result of TaintLabel methods into a trace file for offline analysis to debug issues like this in TruffleTaint.

6.3 Data-Flow Enabled Applications

Label-defined dynamic taint analysis also enables the development of applications that instrument data flow to provide benefits to end users rather than just to analysis developers. One such application is value-based breakpoints. As we described earlier, taint analysis applications can use arbitrary APIs provided by the engine to integrate with its functionality. We leveraged such integration with Truffle's debugging framework to implement *value breakpoints*. In contrast to traditional source-based breakpoints or watchpoints, which suspend program execution when the result of a particular value or expression changes, value breakpoints are attached to a particular value and are triggered when that value or a value derived from it are used by the running program. Value breakpoints enable users to step through program execution based on data flow rather than control flow, which is useful, for example, to diagnose why a particular value flows to a location where it is known to trigger a bug. Note that this application can be used in concert with the information flow analysis we mentioned in Section 6.1. TruffleTaint's general integration with Truffle's debugging support enables attached taint labels to be visible and inspectable in the debugger. By using value breakpoints as delegates of the information flow analysis, each value a breakpoint is attached to also records the previously visited locations, providing an experience similar to time-travel debugging.

7 CONCLUSION

In this paper, we presented label-defined dynamic taint analysis, a novel approach to supporting dynamic taint analysis in analysis runtimes. We showed how this approach enables analyses to specify and influence all aspects of their semantics without requiring analysis developers to modify the analysis engine or limiting them to predefined customization options as it is common in other approaches. We further evaluated an implementation of our approach in TruffleTaint, a polyglot dynamic taint analysis platform we developed in earlier work. This evaluation shows that taint analyses implemented with our approach can be just as fast as equivalent engine-integrated analyses, but that our approach also enables analysis developers to trade off performance in favor of analysis power. As an example, we presented leveraging GraalVM's polyglot capabilities in analysis specifications to lift implementation language restrictions and gain rich, integrated tooling support. Furthermore, we showed how existing applications of dynamic taint analysis can take advantage of capabilities enabled by label-defined taint analysis and presented novel applications such as taint-specific tooling enabled by our approach and value-based breakpoints.

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's Institute for System Software for their support of and feedback on our work.

A TAINT PROPAGATION

In this section we sketch label-defined implementations of various propagation semantics in pseudocode. Figure 11 sketches a label-defined dynamic taint analysis that tracks values created from arithmetic overflows as explained in Section 4.4. Figure 12 sketches the implementation of a generic wrapper analysis as introduced in Section 4.5. Figures 13 and 14 sketch a wrapper analysis that facilitates the execution of multiple independent label-defined dynamic taint analyses in parallel as introduced in Section 4.5.

B FULL BENCHMARK RESULTS

We evaluated the performance of Java implementations of the `VanillaLabel` shown in Figure 3, the `PointerLabel` shown in Figure 5 and a wrapper analysis using the Java implementation of the `VanillaLabel` as a delegate. This evaluation compared the peak performance impact of taint propagation with each of these label-defined taint analyses against the peak performance impact of a vanilla dynamic taint analysis whose propagation semantics was hardcoded in the analysis engine. In this evaluation we used the same benchmarks[5] as we did in our previous work on performance optimizations for TruffleTaint [26]. These benchmarks are reimplementations of benchmark problems defined by the *Computer Language Benchmarks Game* [1], which are commonly referred to as *Shootouts* benchmarks. These benchmarks are designed to introduce tainted values and propagate them as part of their main workload. We evaluated implementations of these benchmarks in either C or C++[6], in JavaScript, and in a combination of of these languages. We ran these benchmarks on a system with an Intel Core i7-3770 processor and 16GB RAM. The Linux-based operating system on this PC was further 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 impact of taint propagation 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. We also repeated each experiment multiple times to verify that the results we collected were reasonably stable and noise-free.

Tables 1 and 2 provide the results of executing our benchmarks with the various dynamic taint analyses we mentioned before. For each benchmark, the tables provide the normalized run time of the benchmark under an engine-integrated vanilla dynamic taint analysis as well as the normalized run time of the benchmark for each of the label-defined dynamic taint analyses we mentioned above. The table further provides for each benchmark and each of these label-defined analyses their increase in normalized run time over the engine-integrated vanilla dynamic taint analysis. Table 2 shows these results when we disabled the taint sources in our benchmarks and therefore no taint was being propagated, even though the instrumentation performing taint propagation was fully enabled. Table 1 shows the results of our benchmarks when the taint sources we defined in them were active.

C APPLICATIONS

In this section we provide the JavaScript code of label-defined implementations of example applications of dynamic taint analysis. The analysis shown in Figure 15 tracks from which taint sources a tainted value originated and prints the source code locations of

[5] We chose to omit running the *KNucleotide* benchmark since its C implementation depends on taint propagation via tainted pointer values. Since the *PointerLabel*, as explained in Section 4.2, ignores the taint status of pointer values, this benchmark experiences different taint spread with this taint label. Due to TruffleTaint's speculative optimization, performance results with different taint spreads are not comparable.

[6] GraalVM supports the execution of C/C++ code through its LLVM IR runtime, which in turn supports taint propagation at the level of LLVM IR via TruffleTaint. We thus compiled the C/C++ implementations of our benchmarks to LLVM IR for execution and TruffleTaint instrumentation.

```
class OverflowLabel extends VanillaLabel {
  constructor(location) { this.location = location; }
  receive(ctx, idx, val) {
    if (isSink(ctx)) {
      print("Value originates from arithmetic overflow that occured at: " + this.location);
    }
    return this;
  }
  merge(other, ctx) {
    if (other instanceof OverflowLabel) {
      return new OverflowLabel(combineLocations(this, other));
    }
    return this;
  }
}
class ValueLabel {
  constructor(val, kind) {
    this.val = val;
    this.kind = kind;
  }
  receive(ctx, idx, val) { return null; }
  merge(other, ctx) {
    if (other instanceof ValueLabel && isOverflow(ctx, this.value, other.value)) {
      return new OverflowLabel(ctx.getSourceLocation());
    }
    return other;
  }
}
class ImplicitLabel extends VanillaLabel {
  receive(ctx, idx, val) {
    if (idx == -3) { return this; }
    if (isArithmeticOp(ctx)) { return new ValueLabel(val, kind); }
    return null;
  }
  merge(other, ctx) { return other; }
}
class LabelFactory {
  createLabel() { return null; }
  beforeProgram() { return new ImplicitLabel(); }
  afterProgram() {}
}
```

Figure 11: Pseudocode sketching a label-defined dynamic taint analysis that tracks values created from arithmetic overflows.

these taint sources when the tainted value reaches a taint sink. The analysis shown in Figure 16 tracks the path a tainted value took from its taint sources to each taint sinks it reaches. When the program has finished its execution, this analysis prints these paths for all tainted values that reached a taint sink.

```
class WrapperLabel {
  constructor(delegate) { this.delegate = delegate; }
  wrap(label) {
    if (label != null) { return new WrapperLabel(label); }
    return null;
  }
  unwrap(label) {
    if (label != null) { return label.delegate; }
    return null;
  }
  receive(ctx, idx, val) {
    return this.wrap(this.delegate.receive(new WrapperContext(this, ctx), idx, val));
  }
  merge(other, ctx) {
    label = this.delegate.merge(other.delegate, ctx);
    return this.wrap(label);
  }
  onBefore(ctx) { this.delegate.onBefore(new WrapperContext(this, ctx)); }
  onAfter(ctx) { this.delegate.onAfter(new WrapperContext(this, ctx)); }
}
class WrapperContext extends Context {
  constructor(label, delegate) { this.label = label; this.delegate = delegate; }
  getFrameLabel() { return this.label.unwrap(this.delegate.getFrameLabel()); }
  setFrameLabel(label) { this.delegate.setFrameLabel(this.label.wrap(label)); }
  clearFrameLabel() { this.delegate.clearFrameLabel() }
  getStaticLabel() { return this.label.unwrap(this.delegate.getStaticLabel()); }
  setStaticLabel(newLabel) { this.delegate.setStaticLabel(this.label.wrap(newLabel)); }
  clearStaticLabel() { this.delegate.clearStaticLabel() }
  changeValue() { this.delegate.changeValue(); }
  get<...>() { return this.delegate.get <...> (); }
}
class WrapperFactory {
  constructor(delegateFactory) { this.delegateFactory = delegateFactory; }
  createLabel() { return new WrapperLabel(this.delegateFactory.createLabel()); }
  beforeProgram() {
    implicitLabel = this.delegateFactory.beforeProgram();
    if (implicitLabel != null) {
      return new WrapperLabel(implicitLabel);
    }
    return null;
  }
  afterProgram() { this.delegateFactory.afterProgram(); }
}
```

Figure 12: Pseudocode sketching a wrapper analysis.

```
function asMap(label) {
  labels = new Map();
  for (label = this; label != null; label = label.next) {
    labels.put(label.id, label.delegate);
  }
  return labels;
}
class CombinedLabel {
  constructor(id, delegate, next) {
    this.id = id;
    this.delegate = delegate;
    this.next = next;
  }
  onBefore(ctx) {
    this.delegate.onBefore(new CombinedWrapperContext(ctx, this.id));
    if (this.next != null) { this.next.onBefore(ctx); }
  }
  onAfter(ctx) {
    this.delegate.onAfter(new CombinedWrapperContext(ctx, this.id));
    if (this.next != null) { this.next.onAfter(ctx); }
  }
  receive(ctx, idx, val) {
    newDelegate = this.delegate.receive(
      new CombinedWrapperContext(ctx, this.id), idx, val);
    if (this.next != null) {
      newNext = this.next.receive(ctx, idx, val);
    } else {
      newNext = null;
    }
    if (newDelegate == null) { return newNext; }
    return new CombinedLabel(this.id, newDelegate, newNext);
  }
  merge(other, ctx) {
    rhs = asMap(other);
    combined = new Map();
    for (label = this; label != null; label = label.next) {
      mergePartner = rhs.remove(label.id);
      if (mergePartner != null) {
        combined.put(label.id, label.delegate.merge(mergePartner, ctx));
      } else {
        combined.put(label.id, label.delegate);
      }
    }
    newLabel = null;
    for (id, label in rhs) { newLabel = new CombinedLabel(id, label, newLabel); }
    for (id, label in combined) { newLabel = new CombinedLabel(id, label, newLabel); }
    return newLabel;
  }
}
```

Figure 13: Pseudocode sketching a wrapper analysis that facilitates the execution of multiple analyses in parallel. Part (1/2)

```
class CombinedFactory {
  constructor(delegateFactories) { this.delegateFactories = delegateFactories; }
  createLabel() {
    newLabel = null;
    for (factory in this.delegateFactories) {
      label = factory.createLabel();
      if (label != null) { newLabel = new CombinedLabel(id, label, newLabel); }
    }
    return newLabel;
  }
  beforeProgram() {
    newLabel = null;
    for (factory in this.delegateFactories) {
      label = factory.beforeProgram();
      if (label != null) { newLabel = new CombinedLabel(id, label, newLabel); }
    }
    return newLabel;
  }
  afterProgram() { for (factory in this.delegateFactories) { factory.afterProgram(); } }
}
function getFrom(combinedLabel, id) {
  for (label = combinedLabel; label != null; label = label.next) {
    if (label.id == id) { return label.delegate; }
  }
  return null;
}
function setLabel(combinedLabel, id, toAdd) {
  newLabel = null;
  for (label = combinedLabel; label != null; label = label.next) {
    if (label.id == id) {
      if (toAdd != null) { newLabel = new CombinedLabel(id, toAdd, newLabel); }
    } else {
      newLabel = new CombinedLabel(label.id, label.delegate, newLabel);
    }
  }
  return newLabel;
}
class CombinedContext {
  constructor(id, delegateContext) { this.id = id; this.delegateContext = delegateContext; }
  getFrameLabel() { return getFrom(this.delegateContext.getFrameLabel(), this.id); }
  setFrameLabel(label) {
    this.delegateContext.setFrameLabel(
      setLabel(this.delegateContext.getFrameLabel(), this.id, label));
  }
  clearFrameLabel() { this.setFrameLabel(null); }
  getStaticLabel() { return getFrom(this.delegateContext.getStaticLabel(), this.id); }
  setStaticLabel(label) {
    this.delegateContext.setStaticLabel(
      setLabel(this.delegateContext.getStaticLabel(), this.id, label));
  }
  clearStaticLabel() { this.setStaticLabel(null); }
  changeValue() { this.delegateContext.changeValue(); }
  get<...>() { return this.delegateContext.get<...>(); }
}
```

Figure 14: Pseudocode sketching a wrapper analysis that facilitates the execution of multiple analyses in parallel. Part (2/2)

Table 2: Change in normalized runtime with various taint label implementations when no taint is introduced.

Language	Benchmark	Vanilla	Vanilla Label		Pointer Arg Label		Vanilla Wrapper Label	
		Engine	Runtime	Change	Runtime	Change	Runtime	Change
C/C++-JS	BinaryTrees	1.18	1.19	+0.8%	1.27	+7.6%	1.20	+1.7%
C/C++-JS	Fannkuch	0.96	0.95	-1.0%	0.96	+0.0%	1.00	+4.1%
C/C++-JS	Fasta	0.98	0.98	+0.0%	0.97	-1.0%	0.96	-2.0%
C/C++-JS	Mandelbrot	1.16	1.16	+0.0%	1.16	+0.0%	1.16	+0.0%
C/C++-JS	NBody 1	1.04	1.02	-1.9%	1.02	-1.9%	1.02	-1.9%
C/C++-JS	NBody 3	1.02	1.02	+0.0%	1.01	-1.0%	1.02	+0.0%
C/C++-JS	NBody 2	0.98	0.91	-7.1%	0.98	+0.0%	0.98	+0.0%
C/C++-JS	Pidigits	1.00	0.99	-1.0%	1.01	+1.0%	1.00	+0.0%
C/C++-JS	ReverseComplement	1.70	1.60	-5.9%	1.71	+0.6%	1.51	-11.2%
C/C++-JS	SpectralNorm	1.07	1.05	-1.9%	1.06	-0.9%	1.05	-1.9%
C/C++	BinaryTrees	1.24	1.24	+0.0%	1.21	-2.4%	1.22	-1.6%
C/C++	Fannkuch	1.04	1.06	+1.9%	1.06	+1.9%	1.04	+0.0%
C/C++	Fasta	1.08	1.09	+0.9%	1.08	+0.0%	1.12	+3.7%
C/C++	Mandelbrot	1.09	1.10	+0.9%	1.09	+0.0%	1.09	+0.0%
C/C++	NBody 1	1.02	1.02	+0.0%	1.02	+0.0%	1.02	+0.0%
C/C++	NBody 3	1.05	1.05	+0.0%	1.04	-1.0%	1.05	+0.0%
C/C++	NBody 2	1.39	1.39	+0.0%	1.39	+0.0%	1.39	+0.0%
C/C++	Pidigits	1.26	1.34	+6.3%	1.35	+7.1%	1.33	+5.6%
C/C++	ReverseComplement	1.63	1.50	-8.0%	1.54	-5.5%	1.52	-6.8%
C/C++	SpectralNorm	1.01	1.01	+0.0%	1.01	+0.0%	1.01	+0.0%
JS	BinaryTrees	1.10	1.15	+4.6%	1.19	+8.2%	1.19	+8.2%
JS	Fannkuch	0.96	0.96	+0.0%	0.94	-2.1%	0.96	+0.0%
JS	Fasta	0.98	0.98	+0.0%	0.97	-1.0%	0.97	-1.0%
JS	Mandelbrot	1.01	1.01	+0.0%	1.02	+1.0%	1.02	+1.0%
JS	NBody 1	1.00	1.01	+1.0%	1.01	+1.0%	1.00	+0.0%
JS	NBody 3	1.04	1.04	+0.0%	1.04	+0.0%	1.04	+0.0%
JS	NBody 2	1.46	1.55	+6.1%	1.54	+5.5%	1.03	-29.4%
JS	Pidigits	1.00	0.99	-1.0%	1.00	+0.0%	1.00	+0.0%
JS	ReverseComplement	1.23	1.22	-0.8%	1.25	+1.6%	1.25	+1.6%
JS	SpectralNorm	1.00	1.00	+0.0%	1.00	+0.0%	1.00	+0.0%

```
class PenumbraLabel {
  constructor(delegate, sources) { this.delegate = delegate; this.sources = new Set([...sources]); }
  isSink(ctx) {
    for (let tag of ctx.tags) {
      switch (tag) {
        case 'LLVMTags.Intrinsic':
          if (ctx.metadata.name.startsWith('__truffletaint_assert_')) { return true; }
          break;
        case 'AnalysisTags.Builtin':
          if ('Taint#assertTainted' == ctx.source) { return true; }
          break;
      }
    }
    return false;
  }
  wrap(label) {
    if (label != null) { return new PenumbraLabel(label, [...this.sources]); }
    return null;
  }
  unwrap(label) { if (label != null) { return label.delegate; } else { return null; } }
  receive(ctx, idx, val) {
    if (this.sources.size == 0) {
      // initialize the source of a new label
      this.sources.add(ctx.source);
    }
    if (this.isSink(ctx)) {
      console.log('Values from these sources arrived at ' + ctx.source + ':');
      for (let source of this.sources) { console.log('-> ' + source); }
    }
    return this.wrap(this.delegate.receive(new WrapperContext(this, ctx), idx, val));
  }
  merge(other, ctx) {
    const mergedSemantics = this.delegate.merge(other.delegate, ctx);
    if (mergedSemantics == null) { return null; }
    return new PenumbraLabel(mergedSemantics, [...this.sources, ...other.sources]);
  }
  onBefore(ctx) { this.delegate.onBefore(new WrapperContext(this, ctx)); }
  onAfter(ctx) { this.delegate.onAfter(new WrapperContext(this, ctx)); }
}
class PenumbraFactory {
  constructor(delegateFactory) { this.delegateFactory = delegateFactory; }
  createLabel() { return new PenumbraLabel(this.delegateFactory.createLabel(), []); }
  beforeProgram() {
    const implicitLabel = this.delegateFactory.beforeProgram();
    if (implicitLabel != null) { return new PenumbraLabel(implicitLabel, []); }
    return null;
  }
  afterProgram() { this.delegateFactory.afterProgram(); }
}
```

Figure 15: JavaScript implementation of a dynamic taint analysis inspired by Penumbra [11]. This implementation is a wrapper analysis similar to the one shown in Figure 12. The `WrapperContext` is intended as shown in Figure 12, but omitted here for brevity.

```
const chainStart = "<start of location chain>";
class PathLabel {
  constructor(flows, location, previous, mergedPrevious) {
    this.flows = flows; this.location = location;
    this.previous = previous; this.mergedPrevious = mergedPrevious;
  }
  static isSink(ctx) {
    for (let tag of ctx.tags) { switch (tag) {
        case 'LLVMTags.Intrinsic':
          if (ctx.metadata.name.startsWith('__truffletaint_assert_')) {
            return true; } break;
        case 'AnalysisTags.Builtin':
          if ('Taint#assertTainted' == ctx.source) { return true; } break; }  }
    return false;
  }
  receive(ctx, idx, val) {
    if (PathLabel.isSink(ctx)) { this.flows.push(this); }
    if (this.location == ctx.source) { return this; }
    return new PathLabel(ctx.source, this, null); }
  merge(other, ctx) { return new PathLabel(this.flows, this.location, this, other); }
}
class GraphvizGraph {
  constructor() { this.nodeId = 0; }
  printHeader() {
    console.log(); console.log("/****************** recorded flows ******************/");
    console.log("digraph Flows {"); console.log("node [shape=record]"); }
  printSubgraph(label) {
    const curId = this.nodeId++; let txt = `n${curId} [label="${label.location}\\l`;
    while (label.previous && label.mergedPrevious == null) {
      label = label.previous;
      if (label.location != chainStart) { txt = `${txt}${label.location}\\l`; } }
    txt = `${txt}"]`; console.log(txt);
    if (label.previous) {
      const leftId = this.printSubgraph(label.previous);
      const rightId = this.printSubgraph(label.mergedPrevious);
      console.log(`n${curId} -> {n${leftId}, n${rightId}}`); }
    return curId; }
  closeGraph() { console.log("}"); }
}
class LabelFactory {
  constructor() {
    this.flows = [];
    this.startLabel = new PathLabel(this.flows, chainStart, null, null); }
  beforeProgram() { return null; }
  createLabel() { return this.startLabel; }
  afterProgram() {
    if (this.flows.length > 0) {
      const graph = new GraphvizGraph(); graph.printHeader();
      for (let label of this.flows) { graph.printSubgraph(label); }
      graph.closeGraph(); } }
}
```

Figure 16: JavaScript implementation of a dynamic taint analysis that tracks the path of tainted values from taint source to taint sink and prints it as *graphviz* plot upon program termination.

REFERENCES

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

[2] 2022. LLVM Data-Flow Sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed: 2022-03-04.

[3] 2022. Truffle Libraries Documentation. https://www.graalvm.org/22.1/graalvm-as-a-platform/language-implementation-framework/TruffleLibraries. Accessed: 2022-05-21.

[4] M. Aldrich, E. Shi, A. Turcotte, and F. Tip. 2022. Augur. https://github.com/nuprl/augur. Accessed: 2022-05-21.

[5] 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

[6] 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

[7] 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

[8] P. Chen and H. Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046

[9] X. Cheng and D. Devecsery. 2022. Creating concise and efficient dynamic analyses with ALDA. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch (Eds.). ACM, 740–752. https://doi.org/10.1145/3503222.3507760

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] E. Gamma. 1995. *Design Patterns*. Addison-Wesley Publishing Co.

[17] 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

[18] D. Hedin, L. Bello, and A. Sabelfeld. 2016. Information-flow security for JavaScript and its APIs. *J. Comput. Secur.* 24, 2 (2016), 181–234. https://doi.org/10.3233/JCS-160544

[19] K. Hough and J. Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 26 (dec 2021), 43 pages. https://doi.org/10.1145/3485464

[20] K. Hough, J. Bell, and G. Kaiser. 2022. Phosphor: Dynamic Taint Tracking for the JVM. https://github.com/gmu-swe/phosphor. Accessed: 2022-05-21.

[21] B. Kang, T. Kim, B. Kang, E. G. Im, and M. Ryu. 2014. TASEL: dynamic taint analysis with selective control dependency. In *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, RACS 2014, Towson, Maryland, USA, October 5-8, 2014*, C. Lu, E. S. Nadimi, S. Kim, and W. Wang (Eds.). ACM, 272–277. https://doi.org/10.1145/2663761.2664219

[22] 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.

[23] 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

[24] 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

[25] 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

[26] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck. 2021. Low-overhead multi-language dynamic taint analysis on managed runtimes through speculative optimization. In *MPLR '21: 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, Münster, Germany, September 29-30, 2021*, H. Kuchen and J. Singer (Eds.). ACM, 70–87. https://doi.org/10.1145/3475738.3480939

[27] 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

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

[29] 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/

[30] 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

[31] 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

[32] A. Slowinska and H. Bos. 2009. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*, W. Schröder-Preikschat, J. Wilkes, and R. Isaacs (Eds.). ACM, 61–74. https://doi.org/10.1145/1519065.1519073

[33] 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

[34] D. Thomas, C. Fowler, and A. Hunt. 2005. *Programming Ruby - the pragmatic programmer's guide (2. ed.)*. O'Reilly.

[35] 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/

[36] 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

[37] 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

[38] Q. Zhang, J. McCullough, J. Ma, N. Schear, M. Vrable, A. Vahdat, A. C. Snoeren, G. M. Voelker, and S. Savage. 2010. Neon: system support for derived data management. In *Proceedings of the 6th International Conference on Virtual Execution Environments, VEE 2010, Pittsburgh, Pennsylvania, USA, March 17-19, 2010*, M. E. Fiuczynski, E. D. Berger, and A. Warfield (Eds.). ACM, 63–74. https://doi.org/10.1145/1735997.1736008