A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships

KATHERINE HOUGH and JONATHAN BELL, Northeastern University, United States

Dynamic taint tracking, a technique that traces relationships between values as a program executes, has been used to support a variety of software engineering tasks. Some taint tracking systems only consider data flows and ignore control flows. As a result, relationships between some values are not reflected by the analysis. Many applications of taint tracking either benefit from or rely on these relationships being traced, but past works have found that tracking control flows resulted in over-tainting, dramatically reducing the precision of the taint tracking system. In this article, we introduce Conflux, alternative semantics for propagating taint tags along control flows. Conflux aims to reduce over-tainting by decreasing the scope of control flows and providing a heuristic for reducing loop-related over-tainting. We created a Java implementation of Conflux and performed a case study exploring the effect of Conflux on a concrete application of taint tracking, automated debugging. In addition to this case study, we evaluated Conflux's accuracy using a novel benchmark consisting of popular, real-world programs. We compared Conflux against existing taint propagation policies, including a state-of-the-art approach for reducing control-flow-related over-tainting, finding that Conflux had the highest F1 score on 43 out of the 48 total tests.

CCS Concepts: • **Software and its engineering → Dynamic analysis**; • **Security and privacy → Information flow control;**

Additional Key Words and Phrases: Taint tracking, control flow analysis, dynamic information flow

ACM Reference format:
Katherine Hough and Jonathan Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 26 (December 2021), 43 pages.
https://doi.org/10.1145/3485464

1 INTRODUCTION

Taint tracking is a technique for monitoring the flow of information through a system. Traditionally, it has been used in privacy analyses to prevent confidential data from leaking into a program's public outputs and in security analyses to detect the flow of untrusted values into sensitive program locations [60, 62]. In recent years, it has also been applied to other software development tasks, for instance, assisting automated input generation systems (fuzzers) [59], helping to

Part of this work was completed while K. Hough and J. Bell were at George Mason University.
This work was funded in part by NSF CCF-2100037, NSF CNS-2100015, and the NSA under contract number H98230-18-D-008.
Authors' address: K. Hough and J. Bell, Northeastern University, 360 Huntington Ave, Boston, MA, 02115-5005; emails: {hough.k, j.bell}@northeastern.edu.

1049-331X/2021/12-ART26 $15.00
https://doi.org/10.1145/3485464

identify poorly designed software tests [36], providing debugging guidance [7, 19], and creating performance models for configurable systems [69].

Dynamic taint tracking associates labels (also referred to as *taint tags*) with program data and propagates these labels through the system during the execution of a program. The set of rules defining how taint tags propagate when an operation executes form the tainting tracking system's propagation policy. This policy effectively describes what it means for information to "flow" through a program. Most taint tracking systems focus on tracing the flow of data through assignment, arithmetic, and logical operations, which directly pass information from their operands to their result. This direct passage of information is referred to as an *explicit* or *data* flow [60]. In a data flow, the value of the flow's target, the operation's result, is derived from the value of flow's source, the operands of the operation. For instance, in line three of Listing 1 there is a data flow from x and y to z. Thus, the labels of x and y should propagate to z.

```
1 int x = taint(4, "X");
2 int y = taint(8, "Y");
3 int z = x + y;
4 int q = 0;
5 if(y == 8) {
6       q = 1;
7 }
```

Listing 1. A basic taint tracking example.

However, tracking only these *explicit* flows can provide an incomplete picture of the flow of information through the system. For example, one might expect q to be tainted after the code in Listing 1 executes, since it is clear that q's value reveals y's value. This indirect passage of information between values can occur as a result of conditional branches, array operations, and pointer dereferencing and is referred to as an *implicit* flow [45]. Unlike a data flow, the value of an implicit flow's target is not related to the value of its source through some computation. Instead, the value of the implicit flow's source is used to "select" the value of the implicit flow's target. For example, if a tainted index is used to access an element of an array, then the retrieved element's value is not directly derived from the value of the tainted index. However, the retrieved element is selected from the collection of elements in the array as a result of the value of the tainted index. The lack of direct computational relationship between the target of an implicit flow and its source can mean that little to no information is passed along the flow. For example, consider a situation where a tainted index is used to access an element of an array containing the same item at every position. If the tainted index's value is guaranteed to be within the bounds of the array, then the value of the accessed element is not influenced by the value of the tainted index. In this case, propagating labels from the tainted index to the accessed element falsely conveys a relationship between the two values. This is typically referred to as *over-tainting*.

Implicit flows resulting from conditional branches are specifically referred to as control flows, since information is passed via the control structure of the program. The source of a control flow is a tainted branch condition that guards the execution of an assignment statement. The branch condition's value impacts whether the assignment statement executes and therefore selects whether a new value is assigned to the statement's destination storage location. Like other implicit flows, not propagating along control flows can result in critical data relationships being lost, which is referred to as *under-tainting*. For instance, when looking up the value associated with a tainted key in an associative-array data structure, there is likely a control flow, but not a data flow between

the key and the value. Thus, many existing taint tracking systems support the propagation of taint tags through control flows [10–12, 18].

The standard semantics for propagating control flows propagates the taint tag of a branch's predicate to every value written by an assignment statement whose execution is controlled by that branch. However, prior works have found that using the standard control flow propagation semantics resulted in severe over-tainting, making it impractical for their applications [7, 8, 19, 66]. For example, in their tool for debugging configuration errors, Attariyan and Flinn [7] reported that "a strict definition of causal dependencies [control flows] led to our tool outputting almost all configuration values as the root cause of the problem." Clause and Orso [19] discovered that using control flow tracking in Penumbra, a tool for identifying inputs that are relevant to a failure, resulted in larger failure-relevant input sets and in the case of one application resulted in almost all of the program's roughly 15 million inputs being marked as failure-relevant. We performed a case study of Penumbra (detailed in Section 6.5) and observed a similar result: Propagating along control flows resulted in impractically large failure-relevant input sets. We also found that this over-tainting could be reduced by using alternative control flow propagation semantics. However, due to the over-tainting that occurs when using the standard control flow propagation semantics, existing software engineering tools that use taint analysis typically ignore control flows, favoring precision over recall. This over-tainting is not caused by a bug in the taint tracking system, but by a mismatch between the standard control flow propagation semantics and the expectations of downstream analyses. In particular, the standard control flow propagation semantics tend to overreport relationships between values. Downstream analyses expect information flows to be indicative of strong, causal relationships between values. If a single, specific condition results in a particular value being assigned to a particular location, then there is a strong relationship between that condition and that assignment. However, if that same assignment can be triggered by many different conditions, then the relationship between those conditions and that assignment is weaker.

Prior work has considered refinements to the standard control flow propagation policy to address control-flow-related over-tainting. For instance, Bao et al. [8] proposed a refinement to control flow tracking that only considered control flows resulting from strict equality checks rather than control flows resulting from all comparison operators. Kang et al. [41] used symbolic execution to identify and propagate along control flow paths that can only be reached by a single input value. Approaches like these, which reduce over-tainting by considering only a subset of control flows, cannot fully address under-tainting without also causing over-tainting. That is, even if it were possible to determine and propagate along the optimal, minimal subset of control flows necessary to prevent under-tainting, over-tainting could still occur.

What constitutes over-tainting is ill-defined; the types of relationships that need to be tracked varies between applications. Generally, within the context of an application of taint tracking, if a label assigned to a piece of data conveys a relationship between that data and the source of the label that is not useful in that application, then that data is said to be over-tainted. For example, a privacy analysis expects data labeled as confidential to contain enough information from the original private source that if that data were leaked publicly, then it would violate some expectation of secrecy or confidentiality. Although this is still ill-defined, it has likely motivated prior work on reducing control-flow-related over-tainting to consider the root cause of the over-tainting to be the amount of information that is transferred across a control flow. However, not all low-information flows result in over-tainting. Consider the data flow from `x` to `y` in the statement `y = x % 2`. This flow transfers very little information about the value of `x` to `y`. Half of all possible values for `x` map to 0 and the other half to 1; so, it is clearly not a one-to-one mapping. Regardless, the value of `y` is what it is because of the value of `x`, and taint tracking tools are generally expected to report such a flow. Propagating along these sorts of low information data flows does not seem to cause the same

over-tainting issues as propagating along control flows. It is our position that control-flow-related over-tainting stems from a mismatch between the nature of control and data flows.

In particular, taint tags propagated at runtime along data flows only contain information about what has actually happened during a particular execution. Code that has not executed does not impact data flows; they are determined by what has happened and not what could have happened. Dynamic taint tracking only provides insights into observed executions; unlike a static taint analysis, it cannot prove things. This is often presented as a disadvantage of dynamic taint tracking over static taint tracking. However, many software engineering tools rely upon this behavior. For example, OraclePolish [36] used dynamic tainting tracking to evaluate the quality of a test suite and was therefore only interested in code that was actually executed by the test suite. Additionally, ConfAid [7], a system for identifying the root cause of configuration errors, used dynamic taint tracking, because the system sought to identify the cause of the specific failure that actually occurred.

In contrast to data flows, control flows contain information about execution paths that did not happen; they are impacted by code that did not execute. A control flow is produced by a conditional branch splitting the flow of control into two or more paths. Some statements execute on only some and not all of those paths. These statements are therefore considered to be dependent on the branch's outcome. Thus, control flows are inherently concerned with execution paths that were not taken.

Recent applications of dynamic tainting tracking that need or benefit from precise, fine-grained tainting tracking such as OraclePolish [36], VUzzer [59], ConfAid [7], and Rivulet [35] underscore the potential benefits of bridging the gap between the existing semantics for data and control flow tracking. To that end, this article makes the following contributions:

- Alternative control flow scope semantics for reducing the amount of over-tainting;
- A heuristic that considers both dynamic and static information to reduce control-flow-related over-tainting;
- A benchmark for evaluating a control flow propagation policies' ability to precisely capture control flows in real-world Java programs.

2 BACKGROUND AND MOTIVATION

Prior works on taint tracking, information flow control, slicing, and other related topics have used a variety of terms to describe the same or similar concepts to ones discussed in this article. Thus, for the sake of clarity, the terminology used in this work is defined below:

Data flow. A data flow (also known as an *explicit* flow) occurs due to an assignment, arithmetic, or logical operation that directly passes information from its operands to its result [12, 18, 27, 45, 60]. In a data flow, the value of the flow's target (the operation's result) is derived from the value of flow's source (the operands of the operation).

Implicit flow. An implicit flow is the indirect passage of information between values typically as a result of conditional branches, array operations, or pointer dereferencing [45]. In an implicit flow, the value of the flow's source is used to "select" the value of the flow's target. This definition of implicit flows is broader than the one used by Chandra and Franz [12], Sabelfeld and Myers [60], and Enck et al. [27], which includes only the indirect passage of information as a result of control structures.

Control flow. A control flow is an implicit flow resulting from a conditional branch [18, 27].

Dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ dominates a node $v_j \in V$ if all paths in G from v_{entry} to v_j contain v_i.

Post-dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ post-dominates a node $v_j \in V$ if all paths in G from v_j to v_{exit} contain v_i. Note that by this definition every node post-dominates itself and the designated exit node post-dominates every node.
Immediate post-dominance. Let $G = [V, E]$ be a control flow graph with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$. A node $v_i \in V$ is the immediate post-dominator of a node $v_j \in V$ if $v_i \neq v_j$; v_i post-dominates v_j; and there does not exist some $v_k \in V$ such that $v_k \neq v_i$, $v_k \neq v_j$, v_k post-dominates v_j, and v_k does not post-dominate v_i.
Scope of influence of a branch. The scope of influence of the execution of a conditional branching statement is the set of statements that execute after that execution of the conditional branching statement but before the next execution of the first statement in the immediate post-dominator of the basic block containing the branching statement. This definition is equivalent to the range of influence of a branch used by Weiser [71] and Denning and Denning [26].
Control flow scope. The scope of a control flow is the dynamic set of instruction executions during which any taint tags associated with the source of the flow propagate to written values.
Standard control flow scope. We use the term standard control flow scope to refer to the typical definition for the scope of a control flow, which is defined with respect to the post-dominance relation. In particular, the standard scope of a control flow introduced by some conditional branch is defined as the set of instructions that execute after the flow of control splits at the branch but before the flow of control rejoins at the immediate post-dominator of the basic block containing that branch [26].
Over-tainting. Over-tainting is when a taint tag assigned to a value falsely conveys a relationship between that value and the source of the taint tag. While conceptually, over-tainting can be caused by an imprecision in the underlying program analysis, this article focuses on over-tainting caused by a mismatch between a taint tracking system's propagation rules and the expectations of analyses built on top of that taint tracking system. This type of over-tainting behavior was described by Clause and Orso [19], Staicu et al. [66], and Attariyan and Flinn [7].
Under-tainting. Under-tainting is when a value has not been assigned a particular taint tag, falsely conveying a lack of relationship between the value and the source of the taint tag.
Propagation policy. A tainting tracking system's propagation policy is the set of rules defining how taint tags should propagate when an operation executes.

The typical approach to control flow tracking considers there to be a control flow from the predicate of a conditional branch to any values written within the control flow's "scope." Many dynamic taint analysis systems track these scopes using a stack [10–12, 18]. The taint tag of a branch's predicate is pushed onto this stack at the start of a control flow's scope and popped at the end of its scope. Traditionally, this scope is defined with respect to the post-dominance relation. Specifically, the standard definition used for the scope of a control flow introduced by some conditional branch is defined as the set of instructions that execute after the flow of control splits at that branch but before the flow of control rejoins at the immediate post-dominator of the basic block containing that branch [26].

Bao et al. [8] and Kang et al. [41] propose propagating taint tags along a subset of control flows based on the "syntax of [the] comparison expression" using the standard, post-dominator-based definition for control flows' scopes. Additionally, Kang et al. [41] attempt to identify "culprit" flows,

control flows along which taint tags need be propagated to avoid under-tainting. However, even if propagation occurs only along the optimal, minimal subset of control flows necessary to prevent under-tainting (i.e., culprit flows), the standard control flow scope definition can cause over-tainting. Consider the code in Figure 1(a). If taint tags are not propagated along control flows, then under-tainting can occur because the relationship between a plus sign in the input and a space in the output is missed. The minimal subset of control flows needed to correct this under-tainting contains only the flow introduced by the branch from the `switch` statement's case on line 8. Figure 1(d) shows the labels expected to propagate to the output produced by `spaceDecode` when presented with an input array with each of its characters tainted with its position in the array (as shown in Figure 1(c)). However, the control flow graph for `spaceDecode` (depicted in Figure 1(c)) shows that the immediate post-dominator of the basic block that contains `switch(input[i])` is the exit node. Thus, once the branch associated with the case on line 8 is traversed, the label for the predicate of that branch will be pushed onto the taint stack and impact all subsequent instructions until the method is exited. This causes the output of `spaceDecode` to be over-tainted, as described in Figure 1(d).

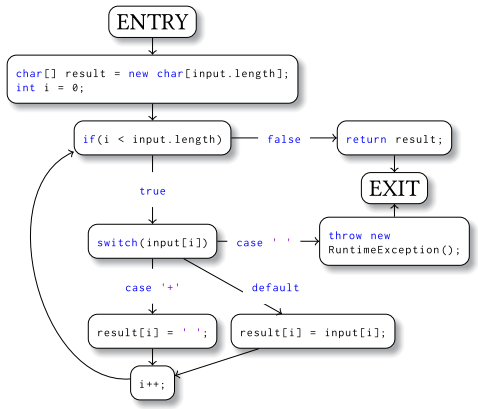
This over-tainting can be fixed by reducing the scope of the control flow to include only the basic block that contains the instructions on lines 9 and 10. However, even if control flow propagation occurs only along the minimal subset of control flows necessary to prevent under-tainting with the minimal, basic-block level scopes necessary to prevent under-tainting for each control flow, over-tainting can still occur. Consider the code in Figure 2(a). If taint tags are not propagated along control flows, then under-tainting can occur because the relationship between a percent sign in the input and a decoded character in the output would be missed. The minimal subset of control flows needed to correct this under-tainting contains only the branch on line 5. As shown in the control flow graph for `percentDecode` depicted in Figure 2(b), the minimal scope for that control flow includes only the basic block that contains the instructions on lines 6 and 7. Figure 2(d) shows the labels expected to propagate to the output produced by `percentDecode` when presented with an input array with each of its characters tainted with its position in the array (as shown in Figure 2(c)). When a percent sign is encountered in the input and the branch on line 5 evaluates to true, the label for that input is pushed onto the taint stack. That label then correctly propagates to the element of the `result` array that is assigned a value on line 6. But, it also incorrectly propagates to the variable `size` and the looping variable `i` when their values are incremented on lines 6 and 7. The end of the control flow's scope is then hit and its label is popped from the taint stack. On subsequent iterations of the loop, when `i` is used to access elements of the input array, `i`'s taint tag propagates to the accessed element. Additionally, when `size` is used to select where in the output array to store a value, `size`'s taint tag propagates to the stored value. This causes the output of `percentDecode` to be over-tainted, as described in Figure 2(d).

3 APPROACH

Our approach, Conflux, aims to precisely propagate taint tags through information-preserving transformations in programs. Conflux leverages novel heuristics to identify control flows that are likely to correspond to information-preserving transformations and ignore taint tag propagation from others. Like Bao et al. [8]'s approach, Conflux considers only a subset of control flows for propagation based on the comparative operator of the control flow's branch. Specifically, Conflux includes only control flows introduced by equality checks. However, even when considering only control flows introduced by equality checks, significant over-tainting can still occur. As discussed in Section 2, the standard, post-dominator-based definition for the scope of a control flow is overly conservative. Thus, Conflux does not use the standard scope definition and instead introduces the notion of a "binding" scope, which includes a subset of the basic blocks contained in the control

```
1 static char[] spaceDecode(char[] input) {
2   char[] result = new char[input.length];
3   for(int i = 0; i < input.length; i++) {
4     switch(input[i]) {
5       case ' ':
6         throw new RuntimeException();
7       case '+':
8         result[i] = ' ';
9         break;
10      default:
11        result[i] = input[i];
12    }
13  }
14  return result;
15 }
```

(a) A simple Java method for decoding spaces encoded as plus signs.



(b) Control flow graph for the spaceDecode method (Figure 1(a)).

Input Value	H	e	l	l	o	+	W	o	r	l	d
Applied Tags	0	1	2	3	4	5	6	7	8	9	10

(c) Sample tainted input for the spaceDecode method (Figure 1(a)). Each input character is tainted with its position in the input array (*e.g.,* the first input character, "H", is tainted with the tag "0").

Output Value	H	e	l	l	o		W	o	r	l	d
Expected Tags	0	1	2	3	4	5	6	7	8	9	10
Propagated Tags	0	1	2	3	4	5	5, 6	5, 7	5, 8	5, 9	5, 10

(d) Expected tainted output from the spaceDecode method (Figure 1(a)) when presented with the sample input from Figure 1(c) compared with the actual tainted output when propagating along the minimal subset of control flows necessary to prevent under-tainting using the standard scope definition.

Fig. 1. An example of a program in which using the standard control flow propagation semantics results in over-tainting. The method spaceDecode in Figure 1(a) takes a sequence of characters that are not spaces. When a plus sign is encountered, a space is added to the output. Every other input character is copied to the output. When spaceDecode is executed with the tainted input displayed in Figure 1(c), the taint tag of every input plus sign ("+") is expected to flow to the output with the produced space character. Additionally, the taint tag of every other input character is expected to flow to the output with the character. However, as shown in Figure 1(d), the standard control flow propagation semantics over-taint the output.

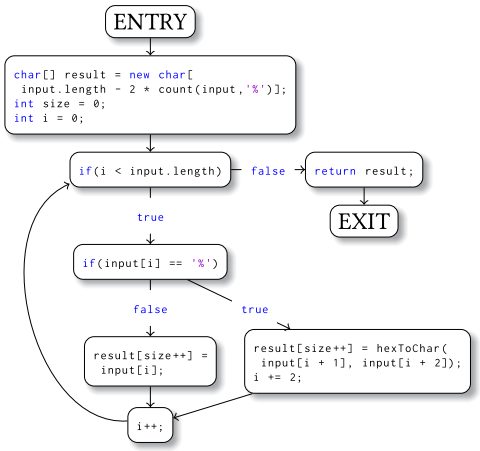
flows' standard scope. However, this alone is not sufficient to produce the expected tainted output in Figure 2(a), since the loop index `i` will still become tainted with the taint tag of `input[i]` on line 5. To address this and similar over-tainting, Conflux introduces a dynamic heuristic, "loop-relative stability," which reasons about the strength of the relationship between a conditional branch and an assignment statement by considering the impact of executing loops on program semantics.

3.1 Binding Scope

Overall, Conflux aims to identify conditional branch executions that are not strictly information-preserving and avoid propagating along the resulting control flows. To achieve this, Conflux uses an alternative control flow scope definition that distinguishes between statements that can execute only as a result of a single, specific condition and statements that can execute under multiple conditions. The traditional definition for control flows' scopes is highly conservative; it considers every basic block between a branch in the flow of control and where that branch rejoins to be within the scope of the branch. Unlike traditional control flow scopes, which are defined with respect to

```java
static char[] percentDecode(char[] input) {
  char[] result = new char[input.length - 2
      * count(input, '%')];
  int size = 0;
  for(int i = 0; i < input.length; i++) {
    if(input[i] == '%') {
      result[size++] = hexToChar(input[i +
     1], input[i + 2]);
      i += 2;
    } else {
      result[size++] = input[i];
    }
  }
  return result;
}
```

(a) A simple Java method for decoding percent-encoded values.



(b) Control flow graph for the `percentDecode` method (Figure 2(a)).

Input Value	%	4	8	%	6	9	%	2	1
Applied Label	0	1	2	3	4	5	6	7	8

(c) Sample tainted input for the `percentDecode` method (Figure 2(a)). Each input character is tainted with its position in the input array (*e.g.,* the first input character, "%", is tainted with the tag "0").

Output Value	H	i	!
Expected Labels	{0, 1, 2}	{3, 4, 5}	{6, 7, 8}
Propagated Labels	{0, 1, 2}	{0, 3, 4, 5}	{0, 3, 6, 7, 8}

(d) Expected tainted output from the `percentDecode` method (Figure 2(a)) when presented with the sample input from Figure 2(c) compared with the actual tainted output when propagating along the minimal subset of control flows with the minimal, basic-block level scopes necessary to prevent under-tainting.

Fig. 2. An example of a program in which using the standard control flow propagation semantics results in over-tainting. The method `percentDecode` in Figure 2(a) takes a sequence of characters and percent-encoded octets. Each input character that is not part of a percent-encoded octet is copied to the output. Each input percent-encoded octet is decoded into a character and that character is copied to the output. When `percentDecode` is executed with the tainted input displayed in Figure 2(c), the taint tag of an input character that is not part of a percent-encoded octet is expected to flow to the output with the character. Additionally, the union of the taint tags of an input percent-encoded octet is expected to flow to the output with the character decoded from the octet. However, as shown in Figure 2(d), the standard control flow propagation semantics over-taint the output.

nodes in the control flow graph, "binding" scopes are defined with respect to edges. In particular, binding scopes are defined with respect to the edges that are traversed when a conditional branching statement is "taken" or "not taken" (or for `switch` statements the edges associated with the cases of the `switch`). The binding scope of a branch edge includes instructions that only execute if the edge is traversed, i.e., an instruction is included if every path from the distinguished entry node of the control flow graph to that instruction contains the branch edge. Since Conflux only considers branch edges corresponding to equality checks (e.g., the branch taken side of an equality check or the branch not taken side of an inequality check), the execution of an instruction within the scope of a branch edge occurs only if some value was equal or "bound" to another value.

Binding scopes can be calculated for the branch edges in a method by constructing its control flow graph, $G = [V, E]$, with designated entry and exit nodes denoted by $v_{entry} \in V$ and $v_{exit} \in V$,

respectively. Each node in V other than v_{entry} and v_{exit} represents a basic block and consists of a sequence of instructions. The successors of a node $u \in V$ is defined as $succ(u) = \{v \mid (u, v) \in E\}$. A node $v \in V$ is said to be a branch node if it has more than one successor. The last instruction of a branch node is its conditional branch instruction. We refer to the set of outgoing edges of a branch node as its branch edges. Each branch edge is associated with a set of conditions under which the edge is traversed. For example, an `if` statement will have two branch edges: one edge corresponding to the branch being taken with a singleton condition set corresponding to the statement's predicate evaluating to true and one edge corresponding to the branch not being taken with a singleton condition set corresponding to the statement's predicate evaluating to false. Whereas, a `switch` statement's branch edges' condition sets will partition the cases of the `switch` statement.

An instruction i is within the binding scope of a branch edge e if i can only execute if e has been traversed. By definition [2], all of the instructions within a basic block execute if and only if the first instruction in the basic block executes. Therefore, we can define binding scopes with respect to basic blocks instead of individual instructions. Thus, the binding scope of a branch edge e is the set of all basic blocks v such that all paths in G from v_{entry} to v contain e.

To calculate the binding scope of branch edges, we first construct a modified control flow graph, G', from G by replacing each branch edge $(u, v) \in E$ with a new node $b_{u,v}$ and a pair of edges $(u, b_{u,v})$ and $(b_{u,v}, v)$. Given this construction, the binding scope of a branch edge e is the set of all $v \in V$ such that b_e dominates v in G'. Proof of the correctness of this calculation is as follows:

Proposition 1. *Given a branch edge $e \in E$ and b_e, its node replacement in G', b_e dominates a node $v \in G'$ if and only if all paths in G from v_{entry} to v contain e.*

Proof. Suppose that b_e dominates v in G'. Assume that there exists a path, $P = [v_{entry}, v_0, \ldots, v_n, v]$, in G that does not contain e. Since b_e dominates v in G', every path in G' from v_{entry} to v must go through b_e. Given that $b_e \notin V$ and therefore $b_e \notin P$, P must contain at least one edge, (v_i, v_{i+1}), that is not in E' such that all paths from v_i to v_{i+1} in G' contain b_e. Furthermore, $(v_i, v_{i+1}) \in E$ and $(v_i, v_{i+1}) \notin E'$ implies that (v_i, v_{i+1}) was a branch edge that was replaced during the construction of G'. Thus, $b_{v_i,v_{i+1}} \in V'$, $(v_i, b_{v_i,v_{i+1}}) \in E'$, and $(b_{v_i,v_{i+1}}, v_{i+1}) \in E'$. As a result, there is a path $[v_i, b_{v_i,v_{i+1}}, v_{i+1}]$ in G'. This path must contain b_e, therefore $b_{v_i,v_{i+1}} = b_e$. Therefore, given the construction process of G', $(v_i, v_{i+1}) = e$. This contradicts the assumption P does not contain e. Thus, b_e dominates v in G' implies that all paths in G from v_{entry} to v contain e.

Now suppose that all paths in G from v_{entry} to v contain a branch edge e. Assume that b_e does not dominate v in G' because there exists some path, $P = [v_{entry}, v_0, \ldots, v_n, v]$ in G' that does not contain b_e. Given a sequential pair of nodes v_i and v_{i+1}, in P, the edge (v_i, v_{i+1}) is either present in E or it was added when a branch edge in E was replaced. If $(v_i, v_{i+1}) \notin E$, then either $v_i \notin V$ or $v_{i+1} \notin V$ This is a consequence of the construction procedure for G', since each edge added to G' is between an element of the original set of nodes and a node not in the original set of nodes. Furthermore, as a result of this, every edge in G' uses at least one node that is an element of V. Since P begins at a node in V, if $v_i \notin V$, there must be a node, $v_{i-1} \in V$ immediately before v_i in P. Since P ends at a node in V, if $v_{i+1} \notin V$, then there must be a node, $v_{i+2} \in V$ immediately after v_{i+1} in P. Thus, P can be broken into a sequence of sub-paths either of the form $[v_i, v_{i+1}]$ where $(v_i, v_{i+1}) \in E$ or the form $[v_i, v_{i+1}, v_{i+2}]$ where $v_i \in V$, $v_{i+1} \notin V$, and $v_{i+2} \in V$. For each such sub-path, there is an edge in E that does not equal e that connects the start of the sub-path directly to the end. Proof is as follows for each of two cases:

Case 1. $[v_i, v_{i+1}]$ *where* $(v_i, v_{i+1}) \in E$
Since $e \notin E'$, P contains (v_i, v_{i+1}), and P is a path in G', $(v_i, v_{i+1}) \neq e$.

Case 2. $[v_i, v_{i+1}, v_{i+2}]$ *where* $v_i \in V$, $v_{i+1} \notin V$, *and* $v_{i+2} \in V$
The edges (v_i, v_{i+1}) *and* (v_{i+1}, v_{i+2}) *must have been added to* E' *when the node* v_{i+1} *was added to* V' *as a replacement for the edge* (v_i, v_{i+2}). *Since* b_e *is not in* P *and* v_{i+1} *is in* P, $v_{i+1} \neq b_e$. *Therefore,* $(v_i, v_{i+2}) \in E$ *and* $(v_i, v_{i+2}) \neq e$.

These edges form a path in G from v_{entry} to v not containing e contradicting the assumption that all paths in G from v_{entry} to v contain the branch edge e. Thus, if all paths in G from v_{entry} to v contain a branch edge e, then b_e dominates v in G'. □

The binding scope of a branch edge is "scope-like"; all nodes within the scope lie on paths between the edge and its dominance frontier. Thus, the taint tag of a branch's predicate need only be pushed onto a taint stack once at the start of the branch edge's scope and can be safely popped at the ends of its scope. Another desirable property of binding scopes is that the set of basic blocks within a branch edge's binding scope is a subset of the basic blocks within its branch's standard (post-dominator based) scope. The immediate post-dominator of a branch node is reachable from all of its branch edges and therefore either part of or beyond the dominance frontier of its node replacement in the modified graph.

We can now apply the binding scope definition to the `spaceDecode` method in Figure 1(a) using its control flow graph shown in Figure 1(b). There are two branch edges corresponding to equality checks: the one associated with `case '+'` and the one associated with `case ' '`. The `case '+'`'s edge's binding scope contains only the basic block with the instruction `result[i] = ' '`. The `case ' '`'s edge's binding scope contains only the basic block with the instruction `throw new RuntimeException()`. In this case, using binding scopes allows the relationship between a plus sign in the input and a space in the output to be reflected without introducing over-tainting.

3.2 Loop-relative Stability Heuristic

When propagating along control flows, taint tags often accumulate on program data during the execution of a loop leading to an "explosion" of taint tags. For example, regardless of whether taint tags are applied in standard control flow scopes or only in binding scopes, during the execution of the loop in the `percentDecode` method (Figure 2(a)), taint tags build up on the looping variable, `i`, resulting in progressively larger label sets for each successive output. Conflux mitigates this accumulation of taint tags by making special considerations when determining whether to propagate taint tags between the branch of a control flow and a statement within the control flow's scope. This process is guided by the novel "loop-relative stability" heuristic.

The underlying idea for loop-relative stability is that loops introduce alternative paths to statements within the scope of a control flow. For instance, within a single call to the `percentDecode` method (Figure 2(a)), the instruction on line 7, `i += 2`, can execute even if the branch on line 5 is not taken on a particular iteration of the loop. That is, on subsequent iterations of the loop, the branch on line 5 could evaluate to true, causing the statement `i += 2` to execute, thereby producing the same effect that would have happened had the branch been taken on the earlier iteration. This weakens the relationship between the value of `i` and the element of the `input` array that caused the branch on line 5 to be taken. However, when a value is stored to `result[size++]` on line 6, the storage location to which that value is stored is different on each iteration of loop. Like before, on subsequent iterations of the loop, the branch on line 5 could evaluate to true causing the statement on line 6 to execute. However, unlike the statement on line 7 that updates the same local variable `i` on different iterations, the statement on line 6 updates a *different* element in the `result` array on different iterations. This produces a stronger relationship between the value of `input[i]` on line 5 and the value written on line 6 to `result[size++]` than the relationship between the value of `input[i]` on line 5 and the value written on line 7 to `i`.

The loop-relative stability heuristic aims to identify these cases in which a loop introduces multiple conditions under which the same location could be assigned the same value. For this to occur, there must be a conditional branching statement that is contained within a loop and an assignment statement within the scope of that branch's control flow. If the values used by the branch change on different iterations of the loop but the values used by the assignment statement stay the same, then on each iteration of the loop a different condition could cause the same effect. Since the conditional branching statement might occur in a different method than the loop and assignment statement, the loop-relative stability heuristic is defined in terms of the dynamic execution of statements. To capture these ideas, we define an execution of a program statement as being "stable" relative to an executing loop if the values used by that statement are the same on every iteration of that loop. The loop-relative stability heuristic determines whether to propagate along a particular control flow based on the stabilities of statement executions. More specifically, let b be an execution of conditional branching statement and a be an execution of an assignment statement that is within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if b is stable relative to every loop to which a is relatively stable.

The loop-relative stability heuristic considers only "natural" loops as defined by Reference [2]. In particular, a natural loop is a single-point-of-entry cycle in the control flow graph defined with respect to a back edge, i.e., an edge whose target dominates its source. The natural loop of some back edge (u, v) consists of all nodes x such that v dominates x and there exists a path from x to u not containing v. The node v is said to be the header of the natural loop defined by the back edge (u, v). Any two natural loops with the same header are combined and treated as a single loop. This definition of loops ensures that any two loops are either disjoint or nested, i.e., one loop is fully contained within the other. The use of arbitrary `GOTO` statements may in some cases produce control flow graphs that contain cycles that do not correspond to natural loops. However, most structured programming languages do not allow programmers to produce control flow graphs that contain cycles that do not correspond to natural loops. Thus, we feel that it is appropriate for the loop-relative stability heuristic to only consider natural loops.

3.2.1 Instability Levels. Conceptually, the loop-relative stability heuristic could be expressed in terms of "stability sets"; a stability set is the set of loops with respect to which a statement execution is relatively stable. Let S_b be the stability set of an execution of conditional branching statement b and S_a be the stability set of an execution of an assignment statement within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if S_b is a superset of S_a. However, instead of tracking these stability sets, it is possible to express the same concept using a single number, an "instability level." An instability level is a numeric value between zero and the number of loops containing the statement currently executing. This number represents the "depth" of the innermost loop relative to which a statement is not stable. The loop-relative stability heuristic propagates along the control flow from an execution of a conditional branching statement b to an execution of an assignment statement a within the scope of b's control flow only if b's instability level is less than or equal to a's instability level. An explanation of why this simplification is possible is provided below.

Given a statement execution contained within two nested loops, if the values used by the statement change over the duration of the inner loop, then they must also change over the duration of the outer loop, since the inner loop is fully contained within the outer loop. Thus, the stability set of a statement execution is defined by the innermost loop relative to which it is not stable. If an execution of a program statement occurs outside of a loop, then that execution must be stable relative to that loop, since it is not possible for the values used by the statement to change over the duration of the loop. As a result, when calculating the loop-relative stability for the statement

currently executing, only the set of loops containing that statement needs to be considered in the calculation.

However, this does not by itself guarantee that all comparisons for the loop-relative stability heuristic made at runtime only need to consider the set of loops containing the statement currently executing. The stability set of the execution of a conditional branching statement needs to be used before the execution of any assignment statement within the scope of the branch's control flow. This means that the stability set of an execution of a conditional branching statement may be considered by the loop-relative stability heuristic after the innermost loop relative to which the execution was not stable was exited. However, once the innermost loop relative to which an execution of a conditional branching statement is not stable is exited, all subsequent statements will execute outside of that loop and therefore be stable relative to it. Since these executions are stable relative to a loop that the branch execution was not, the loop-relative stability heuristic will prevent the branch's predicate from propagating to these statements as a result of the control flow. Thus, Conflux stops all propagation along the control flow from the execution of a conditional branching statement as soon as the innermost loop relative to which that branch execution is not stable is exited. As a result, all comparisons for the loop-relative stability heuristic made at runtime only need to consider the set of loops containing the statement currently executing. Given this and the fact that the stability set of a statement execution can be defined by the innermost loop relative to which it is not stable, loop-relative stabilities can be specified at runtime as a single number, an instability level, between zero and the number of loops containing the statement currently executing.

These instability levels are calculated at runtime as a function of both static information, the stability "classifier" of a statement (Section 3.2.2), and dynamic information, the context of the statement's execution (Section 3.2.3). A purely dynamic approach could only consider instructions that have already executed. This is problematic because it is possible for the predicate of a conditional branching statement to be the same on all but the last iteration of a loop. By the time that last iteration occurs the loop-relative stability heuristic may have already been applied, causing Conflux to incorrectly propagate along a control flow from that branch. Furthermore, the loop-relative stability is concerned with the presence of alternative conditions that could have produced the same outcome. It does not matter whether those conditions were met on a particular execution. For example, consider the `any` method on line 3 of Listing 2. The for-loop on lines 4–8 introduces multiple conditions under which the value of `x` is set to `true`, specifically if any of the elements of the array `z` is `true`. If `any` is passed an array `new boolean[]{false, false, false, true}`, then there happened to be one condition, `z[3] == true` that was satisfied and caused `x` to be assigned the value `true`. However, there were still possible alternative conditions under which that assignment could have occurred. Therefore, propagation should not occur from the branch on line 5 to the assignment statement on line 6 according to the loop-relative stability heuristic. To handle this, Conflux uses static information to reason about possible executions and assign stability classifiers to program elements.

Conflux also relies upon dynamic information about calling contexts. Consider the `any_` method on line 15 of Listing 2. The `any_` method is functionally equivalent to the `any` method on line 3 of Listing 2, but this functionality is split across two methods, `any_` and `setX`. Because `any_` is functionally equivalent to `any`, taint tag propagation for the two methods should ideally be the same, meaning that propagation should not occur during the execution of the assignment statement on line 12 of the `setX` method. However, `setX` is also called by the method `checkFirst` on line 25 of Listing 2. The conditional branching statement on line 24 of `checkFirst` does not necessarily occur within a loop. Therefore, propagation may need to occur during the execution of the assignment statement on line 12 of the `setX` method when called from the `checkFirst` method.

```
1 static boolean x = false;
2
3 static void any(boolean[] z) {
4     for (int i = 0; i < z.length; i++) {
5         if (z[i]) {
6             x = true;
7         }
8     }
9 }
10
11 static void setX(boolean value) {
12     x = value;
13 }
14
15 static void any_(boolean[] z) {
16     for (int i = 0; i < z.length; i++) {
17         if (z[i]) {
18            setX(true);
19         }
20     }
21 }
22
23 static void checkFirst(boolean[] z) {
24     if (z[0]) {
25         setX(true);
26     }
27 }
```

Listing 2. Java methods to demonstrate why both static and dynamic information is used to calculate loop-relative stabilities.

Thus, it is not possible to determine whether propagation should occur during the execution of the assignment statement on line 12 without considering the dynamic execution context of the call to `setX`. Thus, Conflux constructs and passes between methods information about execution contexts at runtime. These execution contexts are used to calculate instability levels.

3.2.2 Stability Classifiers. Conflux statically assigns stability classifiers to program elements; these classifiers encode information about possible program executions. There are three types of stability classifiers: stable, dependent, and unstable. The stable classifier type indicates that all executions of a program element are stable relative to all loops regardless of the dynamic execution context. We use Stable to denote a stable-type classifier. The dependent classifier type indicates that the instability level of an execution of a program element depends on the instability level of one or more arguments passed to the method call that contains the execution. We refer to the set of parameters corresponding to these arguments as the dependency set of the classifier. We use Dependent$\langle d \rangle$ to denote a dependent-type classifier with a dependency set d. In addition to dependencies on arguments, a dependent-type classifier can also indicate a dependency on the execution context's return value location. We use α to denote a dependency on the execution context's return value location.

The unstable classifier type indicates that the instability level of an execution of a program element depends on the loops that contain the method call that contains the execution. In this case, the element is not stable with respect to all loops containing the method call and possibly

additional loops within the method. We use Unstable$\langle l \rangle$ to denote an unstable-type classifier for a program element that is not stable relative to the elements of l, a set of loops within the method.

At runtime, Conflux determines instability levels based on stability classifiers and execution contexts. The instability level of a stable-type classifier is always zero. The instability level of a dependent-type classifier is dynamically calculated as the maximum instability level of the arguments corresponding to the parameters in the program element's dependency set. The instability level of an unstable-type classifier, Unstable$\langle l \rangle$, is dynamically calculated as the number of loops containing the method call plus $|l|$. We describe this more precisely in Section 3.2.4 below.

To calculate stability classifiers for a method, Conflux first converts the method into an intermediate representation in static single assignment (SSA) form. By converting the method into SSA form, Conflux can easily identify reaching definitions, since there will be exactly one definition reaching each use. Next, Conflux creates a control flow graph for the method and uses this graph to identify which natural loops within the method contain each instruction. We use loops(i) to denote the set of natural loops containing an instruction i. Finally, Conflux calculates the stability classifier of each conditional branching statement, assignment statement, non-void return statement, method call receiver, method call argument, and method call return value location. The stability classifiers of conditional branching statements, assignment statements, and non-void return statements (which Conflux treats like interprocedural assignment statements) are directly used to determine whether to propagate along a control flow in accordance with the loop-relative stability heuristic. To construct an execution context for a method call, Conflux uses the stability classifier of the method call's receiver, arguments, and return value location. We discuss this in detail in Section 3.2.3.

For the sake of computing stability classifiers, we define a function merge(c_1, c_2) in Algorithm 1 for combining two classifiers, c_1 and c_2 to produce a classifier c_3 such that for any possible execution context the instability level of c_3 will be greater than or equal to the the instability level for c_1 and c_2. Using this merge function, Conflux can then calculate stability classifiers for value expressions (program entities that are evaluated to produce a value) and storage locations (program entities that represent a place for storing a value, i.e., the left-hand side of an assignment statement) as described in Algorithms 2 and 3, respectively.

ALGORITHM 1: Function for combining two stability classifiers c_1 and c_2.

```
1:  function MERGE(c1, c2)
2:      if c1 = STABLE then
3:          return c2
4:      else if c2 = STABLE then
5:          return c1
6:      else if c1 = DEPENDENT⟨d1⟩ and c2 = DEPENDENT⟨d2⟩ then
7:          return DEPENDENT⟨d1 ∪ d2⟩
8:      else if c1 = UNSTABLE⟨l1⟩ and c2 = UNSTABLE⟨l2⟩ then
9:          return UNSTABLE⟨l1 ∪ l2⟩
10:     else if c1 = UNSTABLE⟨l1⟩ then
11:         return c1
12:     else
13:         return c2
14:     end if
15: end function
```

ALGORITHM 2: Function for calculating the stability classifier of a value expression e that appears in an instruction i.

```
1:  function valueClassifier(e, i)
2:      f ← loops(i)
3:      if e is a constant or literal then
4:          return Stable
5:      else if e is a storage allocation (e.g., an array definition) then
6:          return Unstable⟨f⟩
7:      else if e loads a value from an array or field then
8:          return Unstable⟨f⟩
9:      else if e is a method invocation then
10:         return Unstable⟨f⟩
11:     else if e is a local variable x then
12:         v ← the value expression assigned to x in the reaching definition of x
13:         c_v ← valueClassifier(v, j)
14:         if c_v = Unstable⟨g⟩ then
15:             return Unstable⟨g ∩ f⟩
16:         else
17:             return c_v
18:         end if
19:     else if e is of the form ⋄_u a (where ⋄_u is a unary operator) then
20:         return valueClassifier(a, i)
21:     else if e is of the form a ⋄_b b (where ⋄_b is a binary operator) then
22:         c_a ← valueClassifier(a, i)
23:         c_b ← valueClassifier(b, i)
24:         return merge(c_a, c_b)
25:     else if e is a parameter or method receiver x then
26:         return Dependent⟨{x}⟩
27:     else if e is a caught exception x then
28:         return Unstable⟨f⟩
29:     else if e is a Φ function then
30:         return Unstable⟨f⟩
31:     end if
32: end function
```

Conflux directly uses the `valueClassifier` function (Algorithm 2) to calculate stability classifiers for method call receivers and arguments, since these program elements are value expressions. The `valueClassifier` function (Algorithm 2) is also used to calculate the stability classifier of each conditional branching statement by applying the function to the predicate expression of the branching statement. To calculate the stability classifier of a non-void return statement, Conflux first calculates the stability classifier of the expression returned by the statement and then combines this classifier with a dependent-type classifier with a single dependency on the execution context's return value location, More formally, the stability classifier of a non-void return statement is merge(Dependent⟨{α}⟩, c_v), where c_v is the stability classifier of the expression returned by the statement. The stability classifier of a method call return value location is determined by applying the `locationClassifier` (Algorithm 3) function to the storage location assigned the value

ALGORITHM 3: Function for calculating the stability classifier of a storage location x that appears in an instruction i.

```
1:  function locationClassifier(x, i)
2:      f ← loops(i)
3:      if x is a local variable then
4:          if x is directly used in an invoke expression on the right-hand side of an assignment
    statement then
5:              return Unstable⟨f⟩
6:          end if
7:          c ← Stable
8:          A ← the set of assignment statements that directly use the value at x
9:          for a ∈ A do
10:             y ← the left-hand side of a (i.e., destination storage location)
11:             if y is not a local variable then
12:                 c_y ← locationClassifier(y)
13:                 c ← merge(c, c_y)
14:             end if
15:         end for
16:         if the value at x is directly used in a return statement then
17:             c ← merge(c, Dependent⟨α⟩)
18:         end if
19:         return c
20:     else if x is an instance field of the form a. field then
21:         return valueClassifier(a, i)
22:     else if x is a class field or global variable then
23:         return Stable
24:     else if x is an array element of the form a[b] then
25:         c_a ← valueClassifier(a, i)
26:         c_b ← valueClassifier(b, i)
27:         return merge(c_a, c_b)
28:     end if
29:     return Stable
30: end function
```

of the result of the method call. If the return value of a method call is unused, then the stability classifier of the return value location for that method call is Stable.

Conflux uses both the `valueClassifier` and `locationClassifier` functions to calculate the stability classifier of an assignment statement. However, Conflux makes a special consideration when calculating the stability classifiers of assignment statements to address situations in which the value to be stored by an assignment was produced from an expression containing a term matching the current value at the destination storage location of the assignment statement. We refer to this as an "update" assignment and exclude any update terms, terms that match the current value at the storage location, from the stability calculation for the value. Thus, before Conflux can determine the stability classifier of an assignment statement, it must identify any portions of the right-hand side of the assignment statement that correspond to excluded update terms. If the destination storage location of the assignment statement is a local variable, then any uses of that definition of that local variable that reach the assignment statement are considered to be update

terms. If the destination storage location is an array, then Conflux considers any values that are loaded from the same array at the same position to be candidate update terms. These values may represent update terms, but it is also possible that the value at that storage location was written between the instruction that loaded the candidate update term and the assignment statement. Conflux ignores the potential for concurrent writes to that storage location and accepts a candidate as being a true update term if there does not exist an execution path containing an array store or a method call between the instruction that loaded the candidate and the assignment statement. In practice, checking all possible paths between the two instructions may be impractical. Thus, for small control flow graphs (those containing less than 10 basic blocks), Conflux checks all possible paths between the two instructions. For larger control flow graphs, Conflux assumes that such a path exists if the two instructions are not in the same basic block. If the destination storage location of the assignment statement is a field, then Conflux considers any values loaded from the same field and, for instance fields, from the same instance to be candidate update terms. Once again, Conflux checks all paths between the instruction that loads the candidate update term and the assignment statement. However, in this case, the paths are checked for method calls and field stores.

Once Conflux has identified any portions of the right-hand side of the assignment statement that are considered to be excluded update terms, it calculates the stability classifier of the assignment statement based on the remaining portions of the right-hand side of the assignment statement and the left-hand side of the assignment statement (i.e., its destination storage location). Algorithm 4 shows how Conflux performs this calculation.

ALGORITHM 4: Function for calculating the stability classifier of an assignment statement that appears in an instruction i and assigns a value expression v to a storage location x.

```
1: function assignmentClassifier(x, v, i)
2:     W ← set containing the portions of v that are not excluded update terms
3:     c ← locationClassifier(x, i)
4:     for w ∈ W do
5:         c_w ← valueClassifier(w, i)
6:         c ← merge(c, c_w)
7:     end for
8:     return c
9: end function
```

3.2.3 Execution Contexts. The loop-relative stability of a statement can vary between executions, depending upon the dynamic execution context in which the call to the method that contains the statement was made. For example, if a call is made to a method from within a loop, then statements within the method may vary with respect to that loop. Additionally, whether a statement within the callee method is stable with respect to a loop may depend on the arguments passed to that method. As a result, the loop-relative stability of the execution of a statement cannot be fully determined through purely static analysis. Instead, it must be calculated at runtime within a specific execution context. To address this, at runtime Conflux records information about execution contexts and passes this information between methods.

The execution context for a particular method call consists of two components: a depth and a level map. The depth of a method call is the number of loops that contain the call to that method. We use $depth(e)$ to denote the depth of an execution context e. The level map of a method call specifies the instability levels of the arguments passed to the method call, the method call's receiver (if the method is an instance method), and the storage location for the return value of the method call (if

the method is non-void). We use level(e, a) to denote the instability level of an argument, method receiver, or return value location a specified by some execution context e.

Before a method call is made from some caller method, m, Conflux constructs an execution context for the call, e'. This execution context is created using the calling method's execution context and the stability classifiers that were determined for the method call's arguments, receiver, and return value location, as described in Section 3.2.2. Conflux defines $depth(e')$ to be the sum of $depth(e)$ and the number of loops within m that contain the method call about to be made. Conflux calculates the instability level of each of the arguments passed to the method call using e and the stability classifier for the argument; this value is then recorded in e''s level map. If the callee method is an instance method, then Conflux calculates and records the instability level of the method call's receiver using e and the stability classifier for the receiver. If the callee method is a non-void method, then Conflux calculates and records the instability level of the method call's return value location using e and the stability classifier of the return value location. Finally, Conflux passes e' to the callee method.

At the program entry point, Conflux creates an initial execution context for the initial method call, since this call does not have a caller from which it would receive an execution context. Thus, Conflux uses an initial execution context e such that $depth(e) = 0$ and level$(e, a) = 0$ for all method elements a.

ALGORITHM 5: Function for calculating an instability level based on a stability classifier c and an execution context e.

```
1: function instabilityLevel(c, e)
2:     if c = Stable then
3:         return 0
4:     else if c = Dependent⟨d⟩ then
5:         return max_{a∈d} level(e, a)
6:     elsec = Unstable⟨l⟩
7:         return depth(e) + |l|
8:     end if
9: end function
```

3.2.4 Applying the Loop-relative Stability Heuristic. At runtime, Conflux combines the static stability classifier (Section 3.2.2) and the dynamic execution context (Section 3.2.3) of an executing program element to compute its instability level. This computation is detailed in Algorithm 5. Computed instability levels are used to construct execution contexts as discussed in Section 3.2.3 and, ultimately, to apply the loop-relative stability heuristic. In particular, let b be an execution of a conditional branching statement and a be an execution of an assignment statement that is within the scope of b's control flow. The loop-relative stability heuristic propagates along the control flow from b to a only if b's instability level is less than or equal to a's instability level.

For example, consider the programs shown in Listings 3 and 4 in Figure 3. These programs are nearly identical; they differ only in the names of methods and the value passed to the call to the method `set` on line 13. However, this slight, semantic difference is enough to the impact instability levels in a way that produces different taint tag propagation.

Consider two program executions: one starting from the `main` method in Listing 3 and the other starting from the `main` method in Listing 4. Both executions proceed as follows: The `main` method calls an intermediate method, `indexOf` in one case and `contains` in the other. This intermediate method contains one natural loop L (the for-loop on lines 9–15). In both executions, there are two

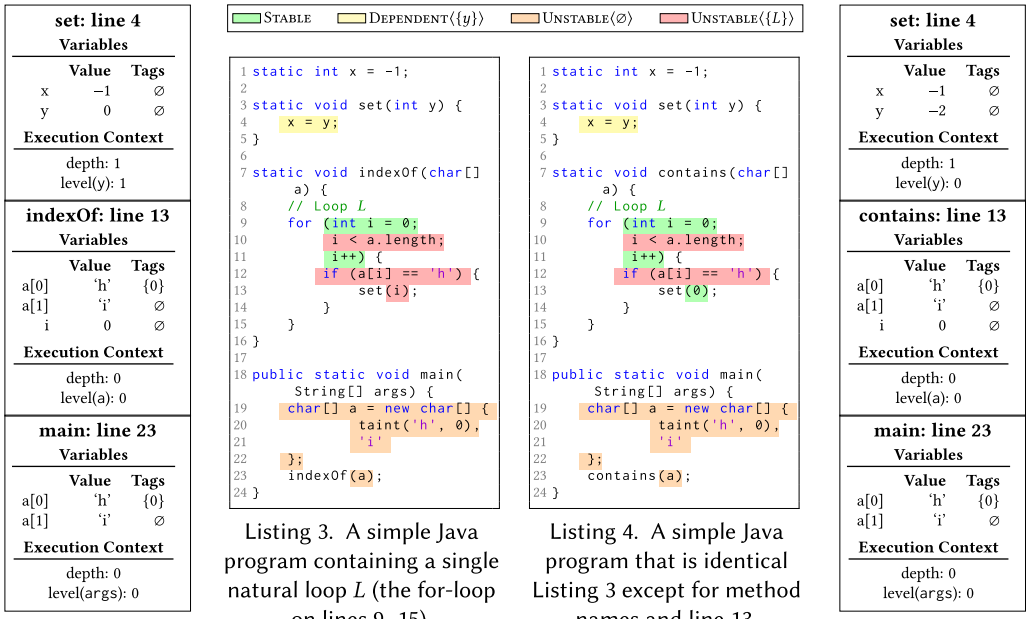


Fig. 3. Listings 3 and 4 depict Java methods with programs elements colored green, yellow, orange, or red to indicate a stability classifier of Stable, Dependent⟨{y}⟩, Unstable⟨∅⟩, or Unstable⟨{L}⟩, respectively. Figure 3(a) shows the stack trace for an execution of the program in Listing 3 just before the instruction on line 4 of the `set` method in Listing 3 is executed for a program execution starting from the `main` method in Listing 3. Figure 3(b) shows the stack trace just before the instruction on line 4 of the `set` method in Listing 4 is executed for a program execution starting from the `main` method in Listing 4. Both of these stack traces (Figures 3(a) and 3(b)) consist of three frames. For each frame, we show the value (**Value**) and associated taint tags (**Tags**) of each local variable within the scope of the frame. We also show the execution context (**Execution Context**) calculated by Conflux for each frame.

iterations of this loop. On the first iteration, when `i` is `0`, the predicate of the branch on line 12 evaluates to true, causing a call to be made to the `set` method. The `set` method assigns the value 0 to a static field `x` and then returns. On the second iteration of L, the predicate of the branch on line 12 evaluates to false, and no call is made to the `set` method. After the loop L ends, the intermediate method returns. Then, finally, the method `main` returns.

In both executions, when the branch on line 12 of the intermediate method was taken, the value of the operand `a[0]` of the predicate of that branch was tainted (as depicted in the middle frame of Figures 3(a) and 3(b)). Thus, when the assignment statement on line 4 of `set` executed, it wrote a value within the scope of a control flow introduced by a tainted predicate. Figure 3(a) shows the stack trace for the moment just before the assignment statement on line 4 of the `set` method executed for the execution of the program in Listing 3. Similarly, Figure 3(b) shows the stack trace for the same moment for the execution of the program in Listing 4. To determine whether to propagate along these control flows, Conflux must know the static stability classifiers for the methods called in these two executions and the dynamic execution contexts of the calls made at runtime to these methods.

We annotated Listings 3 and 4 to indicate the stability classifiers of program elements. These stability classifiers are statically determined by applying the rules described in Section 3.2.2.

The assignment statement `x = y` on line 4 of both programs has a stability classifier of Dependent$\langle\{y\}\rangle$. The storage location `x` always refers to the same location regardless of the context in which it executes. However, whether the value represented by the expression `y` changes over the iterations of some theoretical loop containing a call to `set` depends on whether the argument `y` changes over the iterations of that loop. Thus, this assignment statement has a stability classifier of Dependent$\langle\{y\}\rangle$. The assignment statement `i = 0` on line 9 of both programs has a stability classifier of Stable because, regardless of the context in which it executes, it always assigns the same value, `0`, to the same location, the local variable `i`. The statement `i++` on line 11 of both programs is an augmented assignment equivalent to `i = i + 1`. The term `i` in the assignment `i = i + 1` is an excluded update term, and the remaining term `1` is constant. Therefore, the statement `i++` on line 11 of both programs has a stability classifier of Stable. The stability classifiers of the branch on line 10 of both programs (`i < a.length`) and the branch on line 12 of both programs (`a[i] == 'h'`) are both Unstable$\langle\{L\}\rangle$, because both statements use the value of `i`, which changes over the iterations of L. The argument `i` passed to the call to `set` on line 13 of Listing 3 has a stability classifier of Unstable$\langle\{L\}\rangle$, because the value of `i` changes over the iterations of L. By contrast, the argument `0` passed to the call to `set` on line 13 of Listing 4 has a stability classifier of Stable because it is a literal. The assignment statement on lines 19–22 of both programs contains a storage allocation, thus its stability classifier is Unstable$\langle\varnothing\rangle$. The argument `a` passed to the method call on line 23 of both programs has a stability classifier Unstable$\langle\varnothing\rangle$, because the value of the reaching definition of `a` from lines 19–22 contains a storage allocation.

In addition to these stability classifiers, Conflux must also calculate the dynamic execution contexts of method calls according to the rules described in Section 3.2.3. Let e_{main}, $e_{indexOf}$, and e_{set} denote the execution contexts for the calls made to `main`, `indexOf`, and `set`, respectively, in the program execution for Listing 3. Let $e_{main'}$, $e_{contains}$, and $e_{set'}$ denote the execution contexts for the calls made to `main`, `contains`, and `set`, respectively, in the program execution for Listing 4.

Since `main` is the program entry point, e_{main} and $e_{main'}$ are initial execution contexts. Thus, depth(e_{main}) and depth($e_{main'}$) are 0, and level(e_{main}, `args`) and level($e_{main'}$, `args`) are also 0. The bottom frame of Figure 3(a) (labeled "main: line 23") depicts e_{main}, and the bottom frame of Figure 3(b) (labeled "main: line 23") depicts $e_{main'}$.

Conflux constructs $e_{indexOf}$ using e_{main} and $e_{contains}$ using $e_{main'}$. Since there are no natural loops inside `main` that contain the call to `indexOf`, depth($e_{indexOf}$) is depth(e_{main}) + 0 = 0. The argument passed to the call to `indexOf` has a stability classifier of Unstable$\langle\varnothing\rangle$, thus level($e_{indexOf}$, `a`) is depth($e_{main}$) + $|\varnothing|$ = 0. The middle frame of Figure 3(a) (labeled "indexOf: line 13") depicts $e_{indexOf}$. Since the `main` methods for the two programs are identical and $e_{main} = e_{main'}$, $e_{contains}$ is identical to $e_{indexOf}$. The middle frame of Figure 3(b) (labeled "contains: line 13") depicts $e_{contains}$.

Next, Conflux constructs e_{set} using $e_{indexOf}$ and $e_{set'}$ using $e_{contains}$. Since the loop L contains the call to `set` in `indexOf`, depth(e_{set}) is depth($e_{indexOf}$) + 1 = 1. For the same reason, depth($e_{set'}$) is depth($e_{contains}$) + 1 = 1 The argument passed to the call to `set` in `indexOf` has a stability classifier of Unstable$\langle\{L\}\rangle$. Therefore, level(e_{set}, `y`) is depth($e_{indexOf}$) + $|\{L\}|$ = 1. However, the argument passed to the call to `set` in `contains` has a stability classifier of Stable. Therefore, level($e_{set'}$, `y`) is 0. The top frame of Figure 3(a) (labeled "set: line 4") depicts e_{set}. The top frame of Figure 3(b) (labeled "set: line 4") depicts $e_{set'}$.

Finally, these execution contexts and stability classifiers can be used to calculate instability levels for the execution of the program in Listing 3. The stability classifier of the branch on line 12 of `indexOf` is Unstable$\langle\{L\}\rangle$, and the execution of this branch occurred within the runtime execution context $e_{indexOf}$. Therefore, the instability level of the execution of the conditional branching statement is depth($e_{indexOf}$) + $|\{L\}|$ = 1. The stability classifier of the assignment statement on line 4 of `set` is Dependent$\langle\{y\}\rangle$. The execution of the assignment statement occurred within the

runtime execution context e_{set}. Therefore, the instability level of the execution of the assignment statement is $\max_{a \in \{y\}} \text{level}(e_{set}, a) = 1$. Overall, the instability level of the execution of the conditional branching statement was less than or equal to that of the execution of the assignment statement. Thus, following the loop-relative stability heuristic, Conflux would propagate along the control flow between the two statement executions.

As was done for the execution of the program in Listing 3, the execution contexts and stability classifiers can be used to calculate instability levels for the execution of the program in Listing 4. The stability classifier of the branch on line 12 of `contains` is Unstable$\langle\{L\}\rangle$, and the execution of this branch occurred within the runtime execution context $e_{contains}$. Therefore, the instability level of the execution of the conditional branching statement is $\text{depth}(e_{contains}) + |\{L\}| = 1$. The stability classifier of the assignment statement on line 4 of `set` is Dependent$\langle\{y\}\rangle$. The execution of the assignment statement occurred within the runtime execution context $e_{set'}$. So, the instability level of the execution of the assignment statement is $\max_{a \in \{y\}} \text{level}(e_{set'}, a) = 0$. In this case, the instability level of the execution of the conditional branching statement was greater than that of the execution of the assignment statement. Therefore, unlike in the other execution, Conflux would not propagate along the control flow between the two statement executions.

4 IMPLEMENTATION

Although our overall approach is language-agnostic and suitable to many languages, we chose Java as a target language, implementing Conflux as an extension to Phosphor [9, 10], a Java taint tracking framework that propagates taint tags by rewriting Java bytecode using the ASM bytecode instrumentation and analysis framework [55]. Phosphor is a state-of-the-art taint tracking tool upon which several software engineering tools have already been built [13, 35, 38, 64, 68, 69]. Implementing Conflux as an extension to Phosphor allows Conflux to be easily integrated with these tools and any future tools built on top of Phosphor. Furthermore, this choice allows Conflux to support any existing features supported by Phosphor, such as Phosphor's "auto-tainting mode," which allows developers to specify "source" methods at which taint tags are automatically applied and "sink" methods at which taint tags are automatically checked [11]. Phosphor creates a shadow variable for each local variable, object field, and method argument to store taint information. It propagates control flows by adding a parameter of the type `ControlFlowStack` to methods' signatures. This allows Phosphor to use `ControlFlowStack` instances to track the scopes of control flows between method boundaries by passing a `ControlFlowStack` instance from the caller method to callee method as an argument. In a similar fashion, Conflux uses `ControlFlowStack` instances to pass loop-relative stability information and execution contexts between methods.

We modified Phosphor to support custom control flow propagation policies by allowing users to extend Phosphor's default behavior at three phases: annotation, instrumentation, and runtime; we then used these extension points to implement Conflux. During the annotation phase, the system is provided with a list containing a method's instructions and may insert annotations, special notes used to inform the instrumentation process, into this list. In the instrumentation phase, Phosphor traverses the instructions of a method, forwarding any annotations it encounters to the extending system, informing the extending system when certain structures within the method are encountered, and allowing the extending system to add instructions to the method in response. Phosphor allows a system to modify behavior during the runtime phase by specifying a custom subclass of `ControlFlowStack` for use at runtime.

4.1 Annotation

Before a method is instrumented, Conflux statically annotates its instructions with control flow information. These annotations mark static features of the method (e.g., the scopes of control flows)

as well as properties of those features (e.g., stability classifiers). When the method is instrumented, these features and properties are used to determine what code to generate.

Conflux starts by adding annotations to delineate the binding scopes of control flows. The control flow graph of the method being annotated is created to identify branch edges that introduce control flows. Each branch edge that performs an equality check is marked as introducing a control flow that should be propagated at runtime. A branch edge is considered to perform an equality check if one of the following is true:

- it is the edge between a branching instruction conditioned on a non-null equality check and its branch target;
- it is the edge between a branching instruction conditioned on a non-null inequality check and the instruction following it;
- it is the edge between a `switch` instruction and the branch target associated with one of its non-default cases and no other case for that `switch` instruction has the same branch target;
- it is either of the edges out of a branching instruction conditioned on a Boolean comparison.

The last condition, which deals with Boolean comparisons, is a special case. Since there are only two possible values for a Boolean, both edges out of the branch are traversed for only a single value. Thus, both edges correspond to equality checks. The Java Virtual Machine does not provide dedicated Boolean instructions and instead uses instructions that operate on integer values [44]. As a result, Conflux will mark both edges of an integer conditional branch instruction for propagation if it statically determines that the instruction likely performs a Boolean comparison.

Once branch edges have been marked for propagation, Conflux annotates the start of the scope of the control flow associated with each of the edges. Then, it constructs the modified control flow graph described in Section 3.1 and uses Cooper et al. [23]'s algorithm to calculate the dominance frontier of each replacement node (b_e) associated with each marked branch edge (e). An annotation is added at the beginning of each basic block in the dominance frontier of a replacement node to indicate the end of the scope of the control flow associated with the node.

Next, Conflux annotates program elements with their stability classifiers. The method being annotated is converted from Java bytecode into a register-based intermediate representation and then placed into SSA form using Cytron et al. [25]'s algorithm. Conflux maintains a direct correspondence between this intermediate representation and the original Java bytecode so properties calculated on the intermediate representation can be used to annotate instructions in the original bytecode without having to convert out of the intermediate form. This intermediate representation is used to calculate the stability classifiers of program elements in the method being analyzed (as described in Section 3.2.2). Conflux labels each conditional branching statement, assignment statement, non-void return statement, method call receiver, method call argument, and method call return value location with its calculated stability classifier.

Finally, Conflux uses the control flow graph for the method to record loop information. Each method call is annotated with the number of natural loops that contains it within the method being annotated. Additionally, Conflux records the exit points (i.e., the first instruction of a basic block not contained within a particular loop whose predecessor is contained within that loop) of any loops within the method. These features of the method are used at runtime to calculate execution contexts and instability levels.

4.2 Instrumentation and Runtime

During the instrumentation of the method, the annotations added by Conflux are used to generate method calls to the `ControlFlowStack`, the structure Phosphor uses to store the taint tags of branches and propagate control flows [11]. Phosphor's default behavior adds method calls to

push the taint tag associated with each control flow's predicate at the start of its scope and pop any taint tags pushed for each control flow at the end of its scope, as described in Section 2. CONFLUX modifies this behavior to support the loop-relative stability heuristic.

Before making a method call, the `ControlFlowStack` prepares an execution context for the call. The instability levels of the receiver, arguments, and return value location of the method call are calculated based on their stability classifiers and the current method call's execution context. These instability levels are recorded by the `ControlFlowStack`. `ControlFlowStack` also calculates the number of loops containing the method call based on the number of loops within the current method containing the method call and the current method call's execution context.

At the start of the callee method, the `ControlFlowStack` initializes an execution context based on the values prepared by the caller method. Then, the `ControlFlowStack` pushes this new execution context onto a stack of execution contexts. Before a method exits, the current method call's execution context is popped from this stack. When the start of a control flow's scope is hit, the `ControlFlowStack` calculates the instability level of the control flow based on the conditional branching statement's stability classifier and the current method call's execution context. Then, the `ControlFlowStack` records the taint tag of the predicate of the branch with the flow's instability level. At the end of the scope of a control flow, any taint tags recorded for the flow are cleared. As discussed in Section 3.2.1, to use instability levels, all control flows from a branch's predicate must be terminated as soon as the innermost loop relative to which that branch is not stable is exited. Thus, at the exit point of a loop any taint tags recorded at the instability level associated with that loop are removed from the `ControlFlowStack`.

When a non-void return statement or assignment statement executes, the `ControlFlowStack` calculates its instability level based on its stability classifier and the current method call's execution context. The taint tags of any control flow recorded at an instability level less than or equal the instability level of the statement propagate to the program data being assigned a value by the statement.

At instrumentation boundaries, such as native code calls, the callee method will not receive a prepared execution context from the caller. In these cases, CONFLUX assumes that no loops contain the method call. If there are no loops that contain the method call, then there are no loops with respect to which the method call's receiver, arguments, and return value location could be non-stable. Thus, CONFLUX uses an execution context e such that $depth(e) = 0$ and $\text{level}(e, a) = 0$ for all method elements a.

5 LIMITATIONS

Like prior work from Bao et al. [8] and Kang et al. [41], CONFLUX uses a heuristic approach to mitigate control-flow-related over-tainting. Therefore, malicious programs, those intentionally designed to circumvent analyses, are outside the scope of this work. Some applications of dynamic taint tracking, such as confidential enforcement, may be interested in analyzing malicious programs. However, many applications of dynamic taint tracking primarily consider non-malicious programs [7, 19, 33, 35, 36].

Additionally, since CONFLUX relies on a heuristic rather than a sound or complete flow analysis, it is difficult to determine how appropriate it is for a particular application. Furthermore, the heuristic we use relies on conservative assumptions about values loaded from arrays and fields, and it only takes into account direct uses of a local variable when determining its storage location stability. The loop-relative stability heuristic does not consider cycles introduced via recursion, which could potentially impact its applicability to languages that favor recursion over loops. Nonetheless, in our evaluation of real-world Java programs, we found our approach to be quite effective.

Conflux does not address all sources of control-flow-related over-tainting. For example, if an element conditionally added to a resizable array triggers an array resize, then all of the elements copied from the original array to the new, larger array will be tainted with the label associated with the branch that triggered the element to be added to the resizable array. However, mature Java code typically uses `System.arraycopy` to copy elements from the original array to the new, larger array. `System.arraycopy` is a native method, and therefore not instrumented by Phosphor. Phosphor uses predefined propagation logic for many native methods (regardless of the propagation policy). If this were not the case, then there would likely be over-tainting in `System.arraycopy`. However, code performing this sort of array resizing is often contained within a single method. Therefore, it could be identified statically and handled specially by a taint tracking system.

Finally, Conflux does not aim to address over-tainting from sources other than control-flow propagation, e.g., bit-packing and caches. This is an interesting topic for future work.

6 EVALUATION

We performed an evaluation of Conflux to answer the following research questions:

RQ1: How accurately does Conflux propagate taint tags?
RQ2: How does Conflux's performance vary with input sizes?
RQ3: How does Conflux impact a concrete application of taint tracking?

RQ1 and RQ2 examine the utility of Conflux. For these questions, we report metrics like F1 score, which quantify the accuracy of Conflux by comparing taint tags propagated by Conflux against a ground truth. F1 score is calculated as $\frac{TP}{TP+0.5*(FP+FN)}$ where TP is the number of true positives, FP is the number of false positives, and FN is the number of false negatives. However, it is difficult to know for an arbitrary, real-world program which taint tags should propagate to which values and, therefore, challenging to specify a ground truth.

Past works have explored automated approaches for determining ground truth taint tag sets. For example, Bao et al. [8] and Jee [37] used automated approaches to determine ground truth taint tag sets in their evaluations of taint tracking systems. These automated approaches run the same program multiple times with different inputs and compare the outputs. Jee [37] interpreted differences in the outputs for different input values as indicating that taint tags from the input should have propagated to the output. Bao et al. [8] assume that if different input values lead to the same output, the input's tag should not propagate to the output. However, it is infeasible to exhaustively explore any non-trivial input space. Thus, these approaches can only consider a sample of possible input values, and their efficacy is tied to the quality of that sampling. This is problematic, because automatically generating diverse samples from an input space is a challenging problem in its own right.

Furthermore, in ordered output, like text, different inputs may cause outputs to shift positions without impacting their actual values. This makes it challenging to determine which outputs were impacted by a particular change to the input. For example, consider a simple program that receives an HTTP request containing a "message" query parameter and returns an HTML document that contains that parameter. The inputs used in the execution in Figure 4(a) and 4(b) differ only in a single character. However, the outputs differ in every character after "hello." An automated approach may misinterpret this change and expect the taint tag of the changed input character to propagate to every output character following "hello."

Due to these issues, we chose to not use an automated approach for determining expected taint tag sets and, instead, manually determined expected taint tag sets. Manually determining expected taint tag sets for arbitrary, real-world programs is error-prone and subjective. However, it is possible to leverage known properties of a program to determine a ground truth. This approach was

Fig. 4. Two sample executions of a program that receives an HTTP request and replies with an HTML document. In each execution, the value of the "message" query parameter of the input HTTP request is inserted into the output HTML document. Yellow is used to highlight differences between the executions.

used by McCamant and Ernst [45] to evaluate Flowcheck, their system for measuring the amount of information that flows from secret program inputs to public outputs. One of the experiments conducted by McCamant and Ernst [45] used *bzip2*, a lossless compression tool, as an evaluation subject. For their purposes, *bzip2* was an appealing subject, because the expected flow size for a valid input could easily be determined due to the nature of the tool. Specifically, when presented with a compressible, secret input, all of *bzip2*'s output (except a small amount of input-independent output like headers) will leak secret information, because *bzip2*'s compression algorithm is lossless.

Like McCamant and Ernst [45], we chose to leverage program properties to construct a ground truth for programs included in our benchmark for evaluating RQ1 and RQ2. Unfortunately, this limited the types of programs that could be included in the benchmark to those that met certain criteria. These criteria are discussed in detail in Section 6.2. However, the subjects used to evaluate RQ3 were not impacted by this limitation, because we do not report quantitative metrics for RQ3 and, therefore, do not require a known ground truth.

RQ3 explores the practical impacts of Conflux on an application of taint tracking. For this purpose, we chose to examine Clause and Orso [19]'s approach for identifying failure-relevant inputs. The primary assessment originally performed by Clause and Orso [19] was qualitative. Therefore, we decided to also provide a qualitative assessment in our case study instead of a quantitative one. This choice allowed us to be less constrained in our selection of subject applications for the case study, since we no longer needed to determine a ground truth.

6.1 Experimental Setup

We evaluated Conflux in comparison to three other propagation policies: data-only, basic-control, and SCD, an implementation of the control flow semantics presented by Bao et al. [8]. Each of these policies was implemented using Phosphor. All four of the policies propagate taint tags along data flows in a similar fashion to what was described by Bell and Kaiser [10]. Additionally, all of the policies propagate taint tags from an index used to access an element of an array to the element accessed; this applies to both read and write accesses. Only the data-only and basic-control policies propagate taint tags through `instanceof` operations, as that instruction does not match the semantics targeted by Conflux and SCD. Conflux, basic-control, and SCD all propagate taint tags through pointer dereferencing. Each of the policies uses different semantics for propagating control flows. data-only does not propagate along control flows. basic-control propagates along all control flows using the standard scoping semantics described in Section 2. SCD uses the standard scoping semantics, but only propagates along edges corresponding to equality checks. These edges are determined using the same rules described in Section 4.1.

All of our experiments were conducted on OpenJDK version 1.8.0_222. Tests for each propagation policy were run in a JVM that was instrumented according to the policy. Before a test started, any taint tags on values in the JVM were cleared to ensure that taint tags from one test could not impact the results of another test.

6.2 Benchmark

The benchmark we created to answer RQ1 and RQ2 consists of methods for encoding and decoding text drawn from the OpenJDK Java Class Library (version 1.8.0_222) [54] and seven different real-world Java projects:

- Apache Commons Text (version 1.8) [4]
- Apache Commons Codec (version 1.14) [3]
- Bouncy Castle Provider (version 1.46) [43]
- Guava (version 28.2-jre) [29]
- jsoup (version 1.11.3) [39]
- Spring Web (version 5.2.5) [57]
- Tomcat Embed Core (version 9.0.19) [5],

To the best of our knowledge, DroidBench [6] is the only existing Java taint tracking benchmark that contains tests that consider control flows. However, DroidBench only contains four tests involving control flows and was designed for evaluating static analyzers. Thus, we choose not to use it in our evaluation.

The programs featured in our benchmark for encoding and decoding text are all information-preserving. Each of these programs transforms one representation of a sequence of abstract entities into another representation of the same sequence of abstract entities. An abstract entity is an atomic unit of information. For example, when escaping text for inclusion in HTML, the character <, the character entity reference <, and the numeric character reference < all represent the same abstract entity. The information-preserving nature of the programs included in our benchmark provides a clear ground truth: The expected set of taint tags for a program output contains the taint tags of the program inputs that represent the same abstract entity that the output represents. For example, a program might perform percent encoding on the string :@ resulting in the output string %3A%40. The first percent-encoded octet, %3A, represents the same abstract entity represented by the input character :. The second percent-encoded octet, %40, represents the same abstract entity represented by the input character @. Thus, the expected label set for each of the characters in the first octet would contain the unique label assigned to :, and the expected label set for each of the characters in the second octet would contain the unique label assigned to @.

Each method selected for the benchmark we created had to meet several criteria. First, the method had to be deterministic. For valid input, the output of the method had to represent the same sequence of abstract entities as the input. A taint tracking policy that does not propagate control flows should not produce false positives when tracking taint tags through the method. Likewise, a taint tracking policy that propagates along every control flow using the standard semantics described in Section 2 should not produce false negatives when tracking taint tags through the method. This limits the scope of the benchmark to situations in which observed under-tainting or over-tainting is likely related to control flow propagation, as opposed to other sources of imprecision such as caches and bit-packing.

Every test in the benchmark uses one of the selected methods and follows the same general format. The test starts by creating an input representing a sequence of abstract entities appropriate for the method. Each character (or byte in the case of hex encoding) in the input is assigned a unique taint label. The test then transforms the tainted input using the target test method. The

taint label set that is propagated to each character (or byte in the case of hex decoding) of the method's output is compared to the set of labels expected for that element. Labels in both the propagated and expected sets are counted as true positives. Labels in the expected set, but not the propagated set, are counted as false negatives. Labels in the propagated set, but not the expected set, are counted as false positives.

In some cases, a single method is used in multiple tests (e.g., the method `PercentCodec.encode()` from Apache Commons Codec is used in tests in the groups *unicode-percent-decode* and *reserved-percent-decode*). In these instances, we choose to separate different transformations performed by a single method into multiple tests so their results could be more directly compared to a different implementation of the same transformation.

6.3 RQ1: Accuracy

To compare the accuracy of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX, we applied each of the propagation policies to the tests in our benchmarks. For each test, we created an input sequence of eight abstract entities (RQ2 considers different length inputs). We recorded the number of false negatives, false positives, and true positives that were reported by each propagation policy on each test.

Table 1 presents our findings for the different propagation policies on the benchmarks. Overall, CONFLUX had the highest F1 score on a majority of the tests, 43 out of the 48 total tests. DATA-ONLY had the highest F1 score on 23 tests. SCD had the highest F1 score on 21 tests. And finally, BASIC-CONTROL had the highest F1 score on only 3 tests.

Due to the selection criteria for the methods used in the benchmark, DATA-ONLY reports no false positives and BASIC-CONTROL reports no false negatives. In some of the tests, DATA-ONLY fails to propagate any tags to the output, because there are no data flows between the input and the output. For example, in the *html-escape* test using jsoup's `Entities` class, input values flow into a `switch` statement that selects a constant string to append to the output; all of the information that flows between the input and the output is transferred through the `switch` statement. In other tests, DATA-ONLY does report some or all of the possible true positives, because data flows are able to fully or partially capture the relationship between the input and the output. For instance, for tests in the *reserved-percent-encode* group, DATA-ONLY reports two true positives for every false negative. This occurs because tests in this group take a sequence of URI-reserved characters and encode them using percent-encoded octets (e.g., the character @ would end up encoded as %40). There is a data flow between the value of the input character and the two hex digits in the octet, but there is only a control flow between the input character and the percent sign. Thus, DATA-ONLY correctly tracks the relationship between an input character and the two hex digits of the output octet. However, it misses the relationship between an input character and the percent sign of the output octet. By contrast, BASIC-CONTROL never missed a relationship between an input and an output, but reported a relatively large number of false positives on all but three tests. In many cases, BASIC-CONTROL marked all of the inputs as being related to all of the outputs.

SCD reported relatively few false negatives, and 102 out of the 132 of these false negatives occurred in tests from the *unicode-percent-encode* group. Tests in the *unicode-percent-encode* group take a sequence of non-ASCII characters and transform them into UTF-8, percent-encoded octets. Typical implementations of this transformation determine whether an input character needs to be encoded either by checking if it falls into some value range, is outside some value range, or is not present in some set of values that do not need to be encoded. These checks are generally not equality checks, so SCD does not propagate along the branches associated with them, resulting in under-tainting. CONFLUX also under-taints in these tests for the same reason.

Table 1. Comparison of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX on Our Benchmark

Test Group	Project	Implementation	DATA-ONLY				BASIC-CONTROL				SCD				CONFLUX			
			F1	FN	FP	TP	F1	FN	FP	TP	F1	FN	FP	TP	F1	FN	FP	TP
hex-decode	Apache Commons Codec	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Bouncy Castle Provider	Hex	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
	Java Class Library	DatatypeConverter	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Tomcat Embed Core	HexUtils	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
hex-encode	Apache Commons Codec	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Bouncy Castle Provider	Hex	1.00	0	0	16	0.22	0	112	16	1.00	0	0	16	1.00	0	0	16
	Java Class Library	DatatypeConverter	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
	Tomcat Embed Core	HexUtils	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16	1.00	0	0	16
html-escape	Apache Commons Text	StringEscapeUtils	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Guava	HtmlEscapers	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils-ISO-8859-1	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils-UTF-8	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	0.00	36	0	0
	jsoup	Entities	0.00	36	0	0	0.22	0	252	36	0.54	0	62	36	1.00	0	0	36
html-unescape	Apache Commons Text	StringEscapeUtils	0.00	36	0	0	0.22	0	252	36	1.00	0	0	36	1.00	0	0	36
	Spring Web	HtmlUtils	0.71	16	0	20	0.22	0	252	36	0.56	0	56	36	0.71	16	0	20
javascript-escape	Apache Commons Text	StringEscapeUtils	0.50	16	0	8	0.22	0	168	24	0.54	4	30	20	0.91	4	0	20
	Spring Web	JavaScriptUtils	0.00	24	0	0	0.22	0	168	24	0.62	0	30	24	1.00	0	0	24
javascript-unescape	Apache Commons Text	StringEscapeUtils	0.59	14	0	10	0.30	0	112	24	0.65	0	26	24	0.96	2	0	22
quoted-printable-decode	Apache Commons Codec	QuotedPrintableCodec	0.80	14	0	28	0.22	0	294	42	0.48	0	90	42	1.00	0	0	42
quoted-printable-encode	Apache Commons Codec	QuotedPrintableCodec	0.80	14	0	28	0.22	0	294	42	0.28	10	156	32	0.80	14	0	28
reserved-percent-decode	Apache Commons Codec	PercentCodec	0.80	8	0	16	0.22	0	168	24	0.63	0	28	24	1.00	0	0	24
	Apache Commons Codec	URLCodec	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	1.00	0	0	24
	Java Class Library	URLDecoder	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	0.63	0	28	24
	Spring Web	UriUtils	0.80	8	0	16	0.22	0	168	24	0.46	0	56	24	1.00	0	0	24
	Tomcat Embed Core	UDecoder	0.80	8	0	16	0.22	0	168	24	0.63	0	28	24	1.00	0	0	24
reserved-percent-encode	Apache Commons Codec	PercentCodec	0.80	8	0	16	0.22	0	168	24	0.80	8	0	16	0.80	8	0	16
	Apache Commons Codec	URLCodec	0.80	8	0	16	0.22	0	168	24	0.22	0	168	24	0.80	8	0	16
	Guava	UrlEscapers	0.80	8	0	16	0.22	0	168	24	0.70	0	21	24	0.80	8	0	16
	Java Class Library	URLEncoder	0.80	8	0	16	0.22	0	168	24	0.22	0	168	24	0.80	8	0	16
	Spring Web	UriUtils	0.80	8	0	16	0.22	0	168	24	0.80	8	0	16	0.80	8	0	16
	Tomcat Embed Core	UEncoder	0.80	8	0	16	0.22	0	168	24	0.36	0	84	24	0.80	8	0	16
spaces-url-decode	Apache Commons Codec	PercentCodec	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
	Apache Commons Codec	URLCodec	0.00	8	0	0	0.22	0	56	8	0.22	0	56	8	1.00	0	0	8
	Java Class Library	URLDecoder	0.00	8	0	0	0.22	0	56	8	0.22	0	56	8	1.00	0	0	8
	Tomcat Embed Core	UDecoder	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
spaces-url-encode	Apache Commons Codec	PercentCodec	0.00	8	0	0	0.22	0	56	8	0.36	0	28	8	1.00	0	0	8
	Apache Commons Codec	URLCodec	0.00	8	0	0	0.22	0	56	8	1.00	0	0	8	0.00	8	0	0
	Guava	UrlEscapers	0.00	8	0	0	0.22	0	56	8	0.70	0	7	8	1.00	0	0	8
	Java Class Library	URLEncoder	0.00	8	0	0	0.22	0	56	8	0.70	0	7	8	0.00	8	0	0
unicode-percent-decode	Apache Commons Codec	PercentCodec	0.80	20	0	40	0.22	0	420	60	0.37	0	204	60	1.00	0	0	60
	Apache Commons Codec	URLCodec	0.80	20	0	40	0.22	0	420	60	0.37	0	204	60	1.00	0	0	60
	Java Class Library	URLDecoder	0.80	20	0	40	0.22	0	420	60	0.30	0	276	60	0.64	0	68	60
	Spring Web	UriUtils	0.80	20	0	40	0.22	0	420	60	0.30	0	276	60	1.00	0	0	60
unicode-percent-encode	Apache Commons Codec	PercentCodec	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Apache Commons Codec	URLCodec	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Guava	UrlEscapers	0.75	24	0	36	0.22	0	420	60	0.50	22	54	38	0.75	24	0	36
	Java Class Library	URLEncoder	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40
	Spring Web	UriUtils	0.80	20	0	40	0.22	0	420	60	0.80	20	0	40	0.80	20	0	40

Each row reports results for a single test. Tests are grouped by the type of transformation they perform. For each of the propagation policies, we report the number false negatives (FN), the number of false positives (FP), the number of true positives (TP), and the F1 score (F1) recorded for each test. The highest F1 score or scores for each test are colored purple.

CONFLUX only reported false positives in two tests, as opposed to the 28 tests in which SCD reported false positives and the 45 tests in which BASIC-CONTROL reported false positives. Both of these two tests have the same problematic flow. A simplified version of this flow is shown in Listing 5. CONFLUX marks both the branch on line 6 and the statement on line 12 as unstable with respect to all loops that contain them. Thus, CONFLUX propagates taint tags from the predicate of the branch on line 6 to the assigned value on line 12, `c`, resulting in over-tainting. In this case, the loop header (`c == '%'`) is the source of the flow, rather than the flow occurring within the body of the loop. Had the loop header instead been written as `input[inputPosition] == '%'`, then the load would have been considered as outside of the binding scope of the loop header, and CONFLUX would not have over-tainted.

6.4 RQ2: Accuracy versus Input Size

Taint tags often accumulate on program data over loop iterations leading to progressively more over-tainting on each iteration. Many common applications of taint tracking (e.g., fuzzing guidance) tend to use relatively large inputs that often trigger a large number of loop iterations. In

```
1 public static String decode(char[] input) {
2   char[] output = new char[input.length / 3];
3   int outputPosition = 0;
4   int inputPosition = 0;
5   char c = input[0];
6   while(c == '%') { // source of problematic flow
7     output[outputPosition++] = hexToChar(input[inputPosition + 1],
      input[inputPosition + 2]);
8     inputPosition += 3;
9     if(inputPosition + 2 >= input.length) {
10      break;
11    }
12    c = input[inputPosition]; // target of problematic statement
13  }
14  return new String(output);
15 }
```

Listing 5. Simplified code for the tests where Conflux reports false positives.

these cases, it may be impractical to use the standard semantics for control flow propagation due to the "explosion" of taint tags resulting from their accumulation in loops. To evaluate whether Conflux could be used to address this issue, we applied the propagation policies (data-only, basic-control, SCD, and Conflux) to our benchmark with inputs of various lengths (8, 16, 32, 64, 128, 256, 512, and 1,024 abstract entities). We then recorded the F1 score for each policy on each test for each of the lengths and produced a series of plots. We examined these plots to determine how the F1 score for each policy changed as the size of the input scaled. Figure 5 shows four of these plots that represent common cases seen across many of the plots.

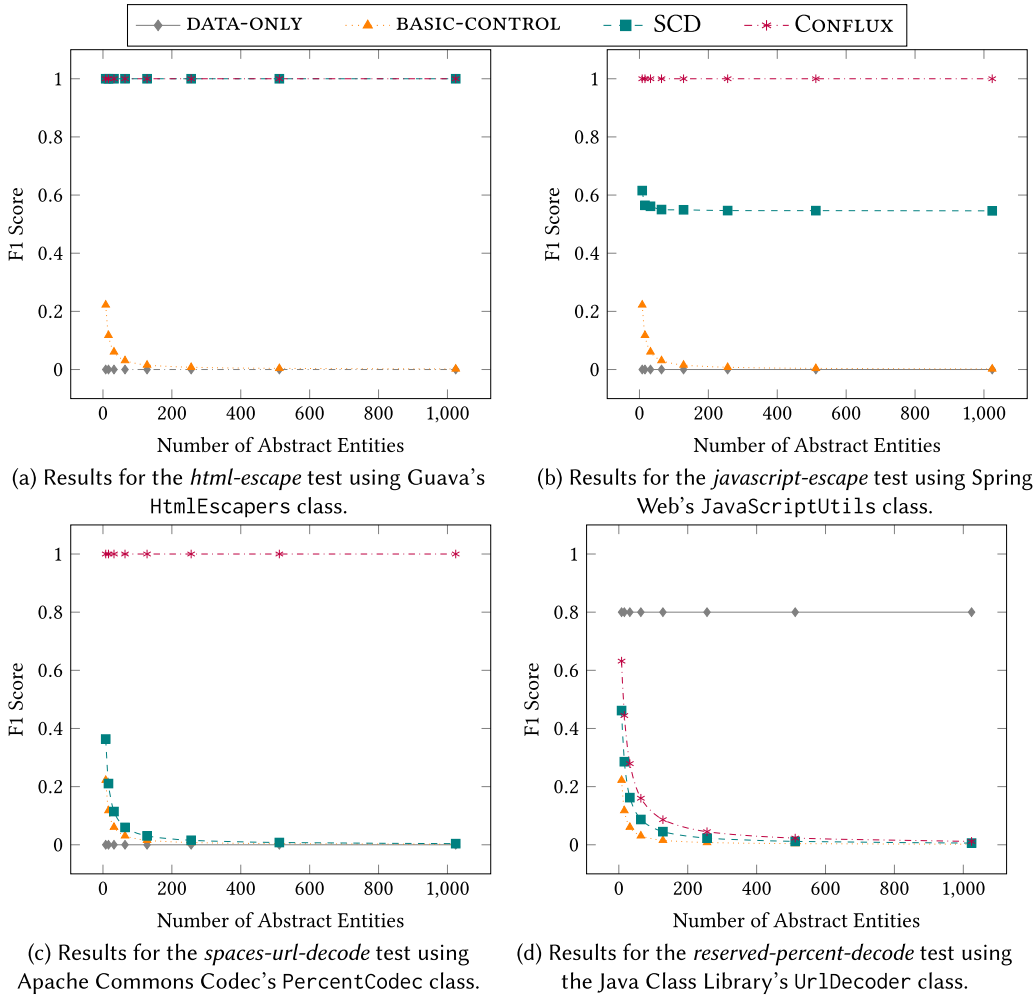
In all of the tests the F1 score for data-only was constant as the size of the input increased. The behavior of the F1 scores for the other three policies either stayed constant, decreased to some non-zero value, or decreased to zero as the size of the input increased. We divided tests into categories based on how the F1 score reported for the different policies changed as the size of the input increased.

There were 3 tests where the F1 scores for all of the policies remained constant as the input size increased. In 17 tests, the F1 scores for data-only, SCD, and Conflux remained constant, but the F1 scores for basic-control decreased to zero. A plot for one of these tests is depicted in Figure 5(a). In 6 tests, the F1 scores for data-only and Conflux remained constant, the F1 scores for basic-control decreased to zero, and the F1 scores for SCD decreased to some non-zero value. A plot for one of these tests is depicted in Figure 5(b). In 20 of the tests, the F1 scores for data-only and Conflux remained constant, but the F1 scores for basic-control and SCD decreased to zero. A plot for one of these tests is depicted in Figure 5(c). There were only 2 tests in which the F1 scores for data-only remained constant, and the F1 scores for basic-control, SCD, and Conflux decreased to zero. A plot for one of these tests is depicted in Figure 5(d).

Overall, Conflux's F1 score stayed constant in all but 2 tests, similar to data-only. By contrast, the F1 score of basic-control and SCD typically degraded as input sizes scaled. In this respect, Conflux's control flow tracking behaved more similarly to data flow tracking than the control flow tracking performed by basic-control and SCD.

6.5 RQ3: Impact on a Concrete Application of Taint Tracking

To examine the effect of Conflux on a practical application of taint tracking, we implemented a prototype of Clause and Orso [19]'s approach for identifying which failure-inducing inputs



(a) Results for the *html-escape* test using Guava's `HtmlEscapers` class.

(b) Results for the *javascript-escape* test using Spring Web's `JavaScriptUtils` class.

(c) Results for the *spaces-url-decode* test using Apache Commons Codec's `PercentCodec` class.

(d) Results for the *reserved-percent-decode* test using the Java Class Library's `UrlDecoder` class.

Fig. 5. Comparison of DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX on selected tests from our benchmark for varying input lengths.

(i.e., inputs that produce a failure) are failure-relevant (i.e., useful for analyzing the failure). We used this prototype to perform a case study exploring the impact of CONFLUX on Clause and Orso [19]'s approach. We conducted an experiment similar to the one originally performed by Clause and Orso [19]. In this experiment, we provide a qualitative assessment of the failure-inducing inputs that were marked as relevant by the different propagation policies, DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX, on five failures from popular, open-source Java projects. Our assessment is limited to the impact of the propagation policies on the values marked as failure-relevant; it does not examine the usefulness of Clause and Orso [19]'s approach in general, since that was explored in the original work.

Table 2 details the five failures that we included in our case study. Each of the these failures was chosen from an issue reported in a project's issue tracker that described a system failure. Only issues in which the system accepted a human-interpretable input or inputs were considered, since it would otherwise be difficult to apply Clause and Orso [19]'s approach. For the sake of

Table 2. Evaluation Subjects Used in the RQ3 Case Study

Project		Issue		Fix	
Checkstyle (version 8.37)	[16]	#8934	[15]	70c7ae0	[14]
Google Closure Compiler (version v20140814)	[22]	#652	[21]	aac5d11	[20]
Mozilla Rhino (version 1.7.11)	[48]	#539	[47]	0c0bb39	[46]
OpenRefine (version 3.4-SNAPSHOT)	[53]	#2584	[52]	825e687	[51]
H2 Database Engine (version 1.4.200)	[32]	#2550	[31]	6c564e6	[30]

For each subject, we list the name and version of the project (Project), the issue in which the failure was reported (Issue), and the commit in which the bug that produced the failure was corrected (Fix).

simplicity, we only selected issues in which the described failure could be consistently reproduced. Furthermore, we only selected issues that had already been fixed in a single, associated commit to facilitate analysis of the failure. A similar criterion was used by Just et al. [40] in the selection of bugs for Defects4J, a widely used database of real Java defects. We used these fixing commits along with the issues filed in the issue trackers to construct appropriate failure-inducing inputs for each failure.

Additionally, for each of the failures, we identified developer comments made in either the fixing commit or the issue that discussed the conditions that produced the failure. These comments reflect the developers' understanding of the failure and the conditions under which the failure manifests. However, mapping these conditions to specific portions of the input is subjective, and the developers' understanding of the failure may be flawed. Thus, these developer-identified failure conditions cannot be used as a ground truth for the failure-relevant portions of an input. For each of the failures, we also created a simplified, failure-inducing input using a combination of random trials of reduced inputs and Zeller and Hildebrandt [73]'s *ddmin* algorithm. The *ddmin* algorithm produces a failure-inducing input that is guaranteed to be "1-minimal," i.e., removing any single element would cause the input to no longer induce the failure; however, the simplified input is not guaranteed to be minimal [73]. Even though a simplified, failure-inducing input can be useful in understanding a failure, it does not necessarily correspond to the failure-relevant portions of the original, failure-inducing input. For example, even a minimized, failure-inducing input can contain portions of the input that are necessary to pass a validation step, but not necessarily useful for investigating the failure. While neither developer-identified failure conditions nor simplified failure-inducing inputs can be used as a ground truth for the failure-relevant portions of an input, they both provide valuable insights into the nature of a failure. Therefore, we used these insights to guide our qualitative assessment of the failure-inducing inputs that were marked as relevant by the different propagation policies.

Our prototype implementation applies a unique taint tag to each character input presented to the applications. These taint tags are propagated in accordance with a particular propagation policy, as described in Section 6.1. In addition to this policy-based propagation, our prototype employs special propagation logic for exceptions. If a Java exception is thrown by the execution of an instruction (as detailed in the Java Virtual Machine Specification [44]), then our prototype propagates the taint tags of the operands of that instruction to the exception. Any input values associated with taint tags that propagate to the exception that produces the studied system failure are marked as failure-relevant.

For each of the bugs that we analyzed, we display the entire program input with annotations that specify which portions of the input were marked as failure-relevant by each policy. These visualizations demonstrate the impact of propagation policies on a concrete software engineering application of taint tracking. Across all five examples, DATA-ONLY marks only a single character of

input as failure-relevant, demonstrating the need to propagate taint tags along control flows in this application. Meanwhile, BASIC-CONTROL marks almost every input character as failure-relevant, underscoring the consequences of control-flow-related over-tainting. We discuss in detail the input values marked as failure-relevant by each policy for each of the studied failures below.

```
1 throw
```

Listing 6. The simplified, failure-inducing JavaScript input for the Closure failure reported in issue #652 [21] created as described in Section 6.5.

```
1
2 2
```

Listing 7. The simplified, failure-inducing CSV input for the OpenRefine failure reported in issue #2584 [52] created as described in Section 6.5. The empty first line is required to induce the failure.

```
1 {
2     "guessCellValueTypes":
        true,
3     "trimStrings": true
4 }
```

Listing 8. The simplified, failureinducing JSON input for the OpenRefine failure reported in issue #2584 [52] created as described in Section 6.5. Whitespace characters have been added to improve the readability of the input.

```
1 function() {
2     try {
3     } finally {
4         v
5     }
6   yield
7 }
```

Listing 9. The simplified, failure-inducing JavaScript input for the Rhino failure reported in issue #539 [47] created as described in Section 6.5. Whitespace characters have been added to improve the readability of the input.

```
1 CREATE TABLE t;
2 MERGE INTO t
3 USING (SELECT 1)
4 ON ()
5 WHEN NOT MATCHED AND b
        THEN INSERT VALUES()
```

Listing 10. The simplified, failure-inducing SQL input for the H2 failure reported in issue #2550 [31] created as described in Section 6.5. Whitespace characters have been added to improve the readability of the input.

```
1 class E {
2     d t = (switch(a) {
3         case 0 -> 1;
4         case 2 -> n;
5     })
6 }
```

Listing 11. The simplified, failure-inducing Java input for the Checkstyle failure reported in issue #8634 [15] created as described in Section 6.5. Whitespace characters have been added to improve the readability of the input.

Checkstyle. Checkstyle is a static analysis tool that finds and reports violations of coding standards in Java code [16]. We studied the failure reported in Checkstyle issue #8934 [15]. A Checkstyle developer described this failure by saying, "FinalLocalVariable throws a NPE on Switch expression in assignment" [15]. However, in the fixing commit for the failure, a different Checkstyle developer noted that, "assigning to [the] `switch` is not the problem . . . wrapping [the] `switch` inside a function foo() makes the problem disappear" [14]. These developer comments indicate that the failure reported in Checkstyle issue #8934 occurs when the "FinalLocalVariable" rule is applied to a Java source code class containing a `switch` expression that is not contained within a method-level or block-level scope [14, 15].

The failure-inducing input that we used to reproduce the failure reported in Checkstyle issue #8934 was a Java source code class based on the Java source code class included in issue #8934 [15]. Figure 6 shows the input Java source code class in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 11 depicts the simplified,

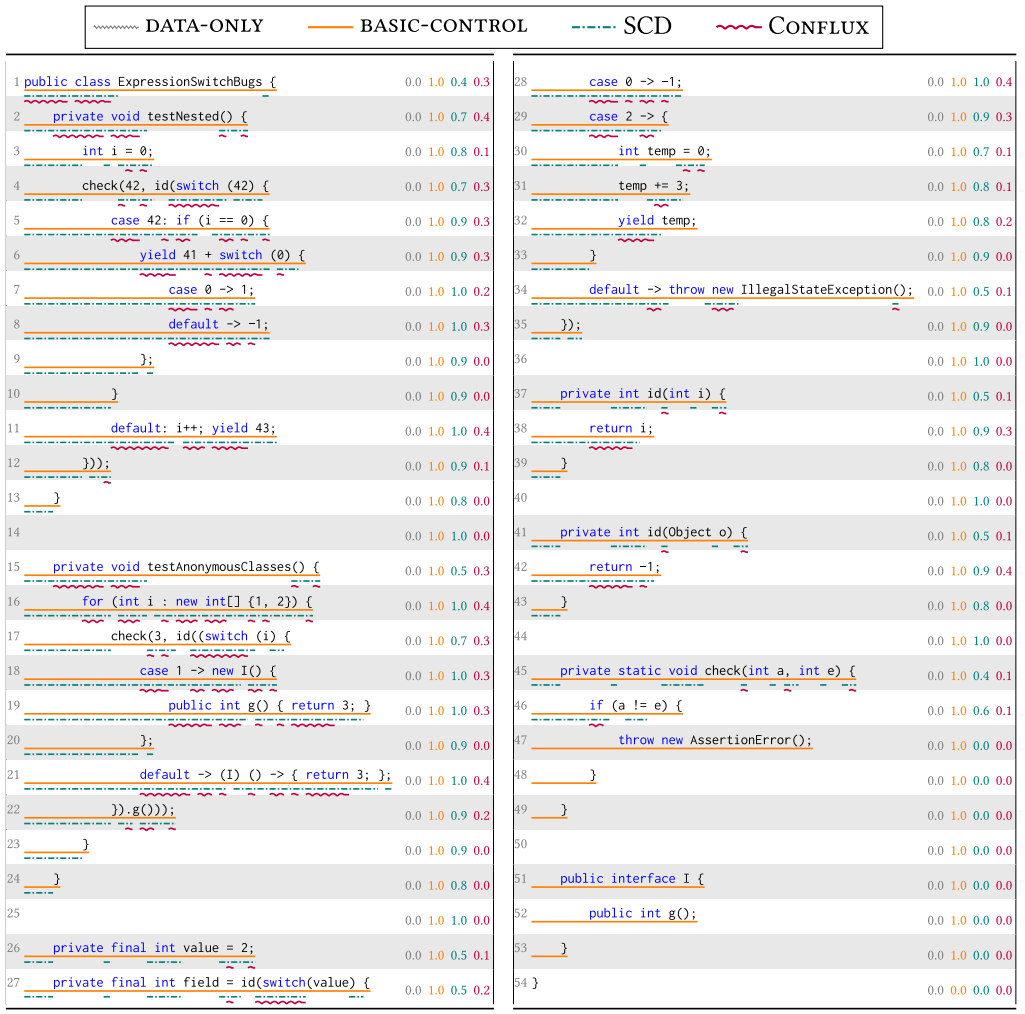


Fig. 6. Java input used to reproduce the Checkstyle failure reported in issue #8634 [15]. Failure-relevant input regions identified by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX are underlined in gray, orange, teal, and magenta, respectively. To the right of each line of input, the ratio between the total number of characters on that line and the number of characters on that line marked as failure-relevant by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX is displayed in gray, orange, teal, and magenta, respectively. This ratio includes newline characters.

failure-inducing input produced for the failure-inducing input in Figure 6. This simplified, failure-inducing input consists of a Java class declaration containing a field declaration that initializes the field to be the value of a `switch` expression. Both the developer-identified failure conditions and the simplified, failure-inducing input suggest that the `switch` expression that appears on lines 27 through 35 of Figure 6 is the cause of the failure.

As shown in Figure 6, DATA-ONLY did not mark any portions of the input as failure-relevant. By contrast, BASIC-CONTROL reported the entire input text except the final closing bracket as

failure-relevant. Both SCD and Conflux report large portions of the `switch` expression that triggered the failure including the `switch` keyword. However, both of these policies also report other regions of the input that are not likely to be helpful to developers trying to debug the failure. Conflux reports fewer of these regions than SCD.

Google Closure Compiler. The Google Closure Compiler is a tool for compiling and optimizing JavaScript code [22]. The failure we selected from the Google Closure Compiler was reported in issue #652 [21]. In the fix for this failure, a Closure developer noted that Closure should "report a parse error if there is a throw followed by a semicolon or newline" [20]. This developer continued by noting that according to grammar for JavaScript, "'throw;' or 'thrown expr;' are illegal" [20]. These developer comments indicate that the failure manifests when Closure compiles code containing a malformed throw statement where the `throw` keyword is followed by a semicolon or newline and not an expression.

We reproduced the failure reported in issue #652 using JavaScript source code based on the JavaScript source code provided in the original issue. Figure 7(a) shows the input JavaScript source code in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 6 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 7(a). The simplified, failure-inducing input consists of just the keyword `throw`. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the failure described in issue #652 is triggered by the malformed throw statement on line 9 of Figure 7(a).

As in the Checkstyle failure, data-only did not mark any portions of the input as failure-relevant. basic-control, SCD, and Conflux all marked the malformed throw statement as failure-relevant. However, basic-control also marked almost every other input character as failure-relevant. SCD and Conflux report some additional regions of the input that may obfuscate the cause of the failure. Once again, Conflux reports fewer of these regions than SCD.

Mozilla Rhino. Mozilla Rhino is an implementation of JavaScript written in Java [48]. Rhino includes a compiler for translating JavaScript source code into Java class files. Issue #539 [47] describes the failure that we examined. In the fix for this failure, a Rhino developer noted that the failure "cropped up when generators were used in a function that had a try..catch..finally block and a yield after the finally" [46]. This comment indicates that the failure manifests when Rhino tries to compile JavaScript source code that contains a generator function or legacy generator function in which a `yield` expression is present after a `finally` block.

We reproduced the failure reported in issue #539 using JavaScript source code based on a test case that was added in the commit that fixed the failure. Figure 7(c) shows the input JavaScript source code in its entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 9 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 7(c). The simplified, failure-inducing input contains a legacy generator function with a `yield` expression following a `try-finally` statement. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the failure described in issue #539 is related to the `finally` on line 6 and the `yield` on line 11 of Figure 7(c).

As shown in Figure 6, data-only did not mark any portions of the input as failure-relevant, and basic-control marked almost every input character as failure-relevant. Unexpectedly, SCD and Conflux often marked only part of a keyword as failure-relevant, for example, both policies marked only the "i" in `finally` as failure-relevant. This was due to the structure of the code that Rhino uses for lexing, converting characters sequences into tokens, JavaScript inputs [48]. Part of this code is displayed in Listing 12. In the case of the keyword `finally`, there is not control flow from all of the characters of the input string "finally" to the token produced from it; there is only a flow from the character "i."

(a) JavaScript input used to reproduce the Closure failure reported in issue #652 [21].

(b) CSV input used to reproduce the OpenRefine failure reported in issue #2584 [52].

(c) JavaScript input used to reproduce the Rhino failure reported in issue #539 [47].

(d) JSON input used to reproduce the OpenRefine failure reported in issue #2584 [52].

(e) SQL input used to reproduce the H2 failure reported in issue #2550 [31].

Fig. 7. Failure-relevant input regions identified by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX are underlined in gray, orange, teal, and magenta, respectively. To the right of each line of input, the ratio between the total number of characters on that line and the number of characters on that line marked as failure-relevant by DATA-ONLY, BASIC-CONTROL, SCD, and CONFLUX is displayed in gray, orange, teal, and magenta, respectively. This ratio includes newline characters.

```
1 L: switch (s.length()) {
2     case 2: c=s.charAt(1);
3     if (c=='f') { if (s.charAt(0)=='i') {id=Id_if; break L0;} }
4     else if (c=='n') { if (s.charAt(0)=='i') {id=Id_in; break L0;} }
5     else if (c=='o') { if (s.charAt(0)=='d') {id=Id_do; break L0;} }
6     break L;
7     case 7: switch (s.charAt(1)) {
8         case 'a': X="package";id=Id_package; break L;
9         case 'e': X="default";id=Id_default; break L;
10        case 'i': X="finally";id=Id_finally; break L;
11        case 'o': X="boolean";id=Id_boolean; break L;
12        case 'r': X="private";id=Id_private; break L;
13        case 'x': X="extends";id=Id_extends; break L;
14    } break L;
```

Listing 12. Part of the Java code used by Mozilla Rhino for lexing JavaScript source code [48].

OpenRefine. OpenRefine is a tool for manipulating, managing, and visualizing data [53]. We studied the OpenRefine failure reported in issue #2584 [52]. A developer for OpenRefine described this failure as occurring "when both trim and autodetect are enabled in tabular parser" [51]. In this and other comments, the developer notes that if both the "trimStrings" and "guessCellValueTypes" configuration options are enabled, OpenRefine will crash when importing a numeric value using an importer that subclasses `TabularImportingParserBase` [51, 52].

Two separate inputs were needed to reproduce the failure reported in issue #2584 [52] The first input was a comma separated values (CSV) file containing input data to be imported that was provided in issue #2584 [52]. The other input was a JavaScript object notation (JSON) configuration file that was based on a test case that was added in the commit that fixed the failure. Figure 7(b) shows the CSV input in its entirety, and Figure 7(d) shows the JSON input in its entirety. Both figures show which portions of the input were identified as failure-relevant by each propagation policy. Listing 7 depicts the simplified, failure-inducing input produced for the failure-inducing, CSV input in Figure 7(b). The simplified, failure-inducing input contains two rows: an empty header row and a row containing a single numeric value. Listing 8 depicts the simplified, failure-inducing input produced for the failure-inducing, JSON input in Figure 7(d). This input is a JSON object with two properties: "guessCellValueTypes" and "trimStrings." The value of both of these properties is the Boolean `true`. Both the developer-identified failure conditions and the simplified, failure-inducing input suggest that the JSON properties "guessCellValueTypes" and "trimStrings" and their values are relevant to the failure. Additionally, both the developer-identified failure conditions and the simplified, failure-inducing input indicate that a numeric value in the CSV input is relevant to the failure. The fixing commit suggests that first numeric value in the CSV input that is not in a header row (the number "1" on line 2 of Figure 7(b)) triggers the failure [51].

As depicted in Figure 7(b), all of the propagation policies, even DATA-ONLY, marked the numeric value that triggered the failure (the number "1" on line 2 of Figure 7(b)) as failure-relevant. This is the only input value ever marked as failure-relevant by DATA-ONLY in any of the failures we examined. BASIC-CONTROL marked additional portions of the CSV input as failure-relevant, likely obfuscating the true cause of the failure. For the JSON input depicted in Figure 7(d), DATA-ONLY did not mark any portions of the input as failure-relevant, and BASIC-CONTROL marked almost every input character as failure-relevant. SCD and CONFLUX reported the properties "guessCellValueTypes" and "trimStrings," and their values as failure-relevant. However, they both reported additional regions of the JSON input that are unlikely to be helpful to a developer trying to debug the failure. CONFLUX reports fewer of these regions than SCD.

H2 Database Engine. H2 is a Relational Database Management System (RDBMS) implemented in Java that supports a subset of Structured Query Language (SQL) [32]. H2 issue #2550 [31] details the failure that we selected. In the fixing commit for this failure, an H2 developer described the failure as a "NullPointerException with MERGE containing unknown column in AND condition of WHEN" [30]. This comment indicates that the failure occurs when H2 tries to execute a `MERGE` statement containing a `WHEN NOT MATCHED` clause with an `AND` expression that refers to an unknown column [30, 31].

We reproduced the failure reported in issue #2550 using the SQL statements provided in issue #2550 [31]. Figure 7(e) shows the input SQL statements in their entirety along with the portions of the input identified as failure-relevant by each propagation policy. Listing 10 depicts the simplified, failure-inducing input produced for the failure-inducing input in Figure 7(e). The simplified, failure-inducing input contains a `CREATE TABLE` statement and a `MERGE` statement. Both the developer-identified failure conditions and the simplified, failure-inducing input indicate that the command `MERGE`, the clause `WHEN NOT MATCHED`, the keyword `AND`, and the unknown reference, "s.b," which appears on line 8 of Figure 7(e), are relevant to the failure.

As depicted in Figure 7(e), DATA-ONLY did not mark any portions of the input as failure-relevant. Conversely, BASIC-CONTROL reported the entire input as failure-relevant. SCD and CONFLUX only marked portions of the `MERGE` statement as failure-relevant. SCD marked almost the entire `MERGE` statement as failure-relevant. CONFLUX only marked small portions of the `MERGE` statement as relevant; it is unclear whether these smaller portions better illuminate the cause of the failure.

Summary. When considering the input values marked as failure-relevant by each propagation policy, it is clear that DATA-ONLY and BASIC-CONTROL are unlikely to be useful to a developer attempting to debug a failure. This underscores the need for alternative taint tag propagation semantics. The portions of each input marked as failure-relevant by SCD and CONFLUX appear to be more useful for analyzing the failures. SCD tended to mark more characters as failure-relevant than CONFLUX and, in some cases, marked most of the input as failure-relevant.

6.6 Threats to Validity

One threat to the validity of our experiments stems from our selection of evaluation subjects for the benchmark. Our benchmark tests a limited number of methods from only a handful of projects. As discussed in Section 6, it is challenging to determine which taint tags should propagate to which values for an arbitrary, real-world program. Thus, only methods that met certain criteria (detailed in Section 6.2) could be included in the benchmark. As a result, the benchmark is not necessarily representative of all Java programs. However, we selected these methods based on a search for popular Java libraries.

Additionally, the ground truth expected label sets we used for the benchmark may not be appropriate for every application of taint tracking. For example, in some applications it could be desirable for propagated labels to reflect looser or stronger relationships than those reflected in our ground truth. Nonetheless, we feel that our ground truth selection follows best practices of state-of-the-art taint tracking evaluations [45, 56].

Unlike the benchmark, the case study evaluation that we performed did not require a manually specified ground truth. However, the case study evaluation was limited to a single application of taint tracking and examined a limited number of subjects and failures. Therefore, it may not generalize to other applications of taint tracking or other subjects.

7 RELATED WORK

Control flow tracking approaches. Several existing taint tracking systems do not offer support for control flow tracking [17, 50, 58]. However, some systems support the standard semantics

discussed in Section 2 [11, 12, 18]. Several of these systems attempt to address control-flow-related over-tainting. Bao et al. [8] propose propagating control flows only along branches that correspond to equivalence checks. Kang et al. [41] put forward a similar approach, but instead target branches related to information-preserving transformations. These branches are determined by analyzing execution traces to find control flow paths that can only be reached by a single input value. Attariyan and Flinn [7] mitigate over-tainting from control flows by reducing the weight of a taint tag when it is propagated via a control flow. Cox et al. [24] address control-flow-related over-tainting in their approach for preventing Android applications from leaking users' passwords by quantifying the amount of information revealed by control flows. This quantity is then used to decide whether to propagate taint tags along a control flow. Unlike these approaches, Conflux uses an alternative definition for control flow scopes and propagates to a subset of statements within control flows' scopes.

Applications of control flow tracking. Both static and dynamic information flow analyses have been applied to a variety of applications in which control flows could significantly impact results. Sabelfeld and Myers [60] explore approaches to security-type systems and semantics-based security models for enforcing information-flow confidentiality policies. They focus on a noninterference policy for confidentiality that is highly conservative and must therefore take implicit flows, such as control flows, into account. McCamant and Ernst [45] propose measuring the maximum flow of secret information with a network flow capacity model instead of using a taint tracking approach to confidentiality enforcement. This quantitative approach may support different techniques for addressing control flows than traditional taint tracking. Halfond et al. [33] provide an automated technique for detecting and preventing SQL injection attacks using "positive-tainting." Positive-tainting tracks the flow of trusted values, as opposed to the more common approach of "negative" tainting, which tracks untrusted values. However, control flows can result in malicious values being built from trusted ones; this issue is not addressed by Halfond et al. [33]. Clause and Orso [19] present Penumbra, a tool for identifying the subset of a failure-inducing inputs that are failure-relevant to assist with program debugging. They found that, for some programs, propagating taint tags along control flows resulted in a prohibitively large number of program inputs being marks as failure-relevant. Huo and Clause [36] use dynamic taint analysis to identify test cases that check too much of the program state, making them difficult to maintain and test cases that check too little of the program state, reducing their ability to detect bugs. Their approach requires both data and control flows to be tracked. Various existing fuzzing tools leverage taint tracking to generate "interesting" inputs capable of finding bugs deep in a program's execution [28, 59, 70]. To the best of our knowledge, none of these tools propagate taint tags along control flows. However, we believe that these tools could likely benefit from applying Conflux's control flow propagation semantics.

Evaluating taint tracking systems. An assortment of techniques have been used to evaluate the accuracy of taint tracking systems. Jee [37]'s tool, TaintMark, looks at system outputs when given different input values to determine if taint tags should propagate from the inputs to the outputs. Differences in the outputs for different input values are interpreted as meaning that taint tags from the input should have propagated to the output. By contrast, ReproDroid, Pauck et al. [56]'s framework for comparing Android taint analysis tools, requires the ground truth for test cases to be manually classified. Pauck et al. [56] note that this manual determination of the ground truth is necessary, because "tools that could potentially be used to derive the ground truth are at the same time the tools we want to evaluate." Inspired by Pauck et al. [56], our evaluation does *not* use an automatically determined ground truth. Other evaluations have also used application specific techniques [7, 35, 36, 59].

Various studies have considered the impact of propagating implicit and control flows on taint tracking results. Staicu et al. [66] investigate the prevalence of implicit flows and the criticality of detecting implicit flows when using dynamic taint tracking to enforce security and privacy policies in JavaScript applications. They conclude that it is sufficient to consider only data flows to detect security-related source-to-sink flows, but to discover privacy-related source-to-sink flows, implicit flows also needed to be considered. King et al. [42] examine explicit and implicit flows that are reported by a security-typed language enforcing noninterference. They found that implicit flows caused the majority of true positives reported, but also caused a large number of the false positives. Additionally, they found that the vast majority of exception-induced flows due to unchecked exceptions were false alarms that could not occur at runtime. Exception-induced information flows are beyond the scope of this work. Clause et al. [18] explore the relationship between different taint propagation approaches and the amount of memory tainted, finding that propagating control flows resulted in significantly more tainted memory. However, they did not evaluate the accuracy of different propagation policies.

Dynamic slicing. One related technique to taint tracking is dynamic slicing [67, 71]. Dynamic slicing computes the subset of program statements that affected values at a particular program point for a particular program execution or executions, referred to as a "slice." Like taint tracking, slicing aims to reason about relationships in programs. However, slicing relates values to statements, whereas taint tracking relates values to other values.

Early work on dynamic slicing examined imprecision related to inaccurate dynamic dependence calculations. These errors were caused by analyses that did not distinguish between different executions of the same instruction. Dynamic taint tracking systems typically track control flows using the stack-based approach described in Section 2. This stack-based approach accurately calculates dynamic dependences and is not subject to the inaccuracy of early dynamic slicers [72]. Agrawal and Horgan [1] introduce the notion of the "Dynamic Dependence Graph" (DDG), which contains a node for each instruction execution and edges between instruction executions that are dynamically dependent. They propose a technique for calculating precise dynamic slices by using DDGs and a more efficient technique that uses a compacted version of the DDG. Zhang et al. [76] improve upon the work of Agrawal and Horgan [1] by using novel data structures that reduce the computational cost of constructing the DDG and leveraging an on-demand construction of DDGs to reduce memory usage. Zhang and Gupta [75] explore the space and time performance benefits of leveraging a novel, highly compact representation of the DDG. Unlike works on precise dynamic slicing, inaccurate dynamic dependence computations are not the source of the control-flow-related imprecision addressed in this work. Instead, Conflux aims to avoid propagating along control flows arising from dynamic dependences that are not likely to be information-preserving despite being genuine dynamic dependences.

Although not identical to the control-flow-related over-tainting problem that Conflux aims to address, prior work on slicing has proposed techniques for reducing the size of slices to better support automated debugging and program understanding tasks. Zhang et al. [74] use a heuristic approach based on the correctness of outputs computed using a statement to identify and remove statements that are unlikely to be related to a fault from computed slices. In contrast to Zhang et al. [74]'s approach, Conflux's heuristic is suitable for applications other than fault analysis and does not require an oracle for determining the correctness of outputs. A related technique, thin slicing, was proposed by Sridharan et al. [65]. Thin slicing only includes statements that are part of some sequence of assignments that compute and copy a value to a target location in the slice for the target location [65]. Whereas Conflux uses binding scopes and the loop-relative stability heuristic to identify control flow relationships that are unlikely to be information-preserving, thin slicing simply excludes all control dependences in its slice construction. An interesting topic for

future work might be to apply Conflux's notion of binding scopes and loop-relative stability to slicing.

8 CONCLUSION

Techniques that require high-precision, dynamic taint tracking are highly prevalent in software engineering research [7, 27, 28, 35, 36, 49, 59, 61, 63, 70]. Many of these techniques would greatly benefit from accurately propagated control flow relationships. However, the standard control flow propagation semantics are mismatched with the standard data flow semantics. This mismatch often results in severe over-tainting making the standard control flow propagation semantics impractical for most applications. Prior approaches to mitigate this over-tainting fail to address many of its fundamental sources. Conflux, our alternative control flow propagation semantics, decreases the scope of control flows and leverages a novel heuristic, loop-relative stability, to determine whether a control flow's taint tags should propagate to a particular statement. We compared Conflux to three other control flow propagation policies on a benchmark containing 48 tests consisting of programs for encoding and decoding text. Conflux had the highest F1 score on 43 out of the 48 total tests when using test inputs of a fixed size. Additionally, when the size of test inputs scaled, Conflux's F1 score remained constant on all but 2 of the 48 tests, indicating that Conflux helped to mitigate taint explosions associated with large inputs. We also examined the impact of Conflux of a concrete application of taint tracking, automated debugging. Conflux and the experiments described in this article are publicly available under the BSD 3-Clause License [34].

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. *SIGPLAN Not.* 25, 6 (June 1990), 246–256. DOI: https://doi.org/10.1145/93548.93576.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc.

[3] Apache Software Foundation. 2019. Apache Commons Codec (version 1.14). Retrieved from http://commons.apache.org/proper/commons-codec/.

[4] Apache Software Foundation. 2019. Apache Commons Text (version 1.8). Retrieved from https://commons.apache.org/proper/commons-text/.

[5] Apache Software Foundation. 2019. Apache Tomcat (version 9.0.19). Retrieved from https://tomcat.apache.org.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI'14)*. Association for Computing Machinery, New York, NY, 259–269. DOI: https://doi.org/10.1145/2594291.2594299

[7] Mona Attariyan and Jason Flinn. 2010. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association 237–250.

[8] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. ACM, New York, NY, 13–24. DOI: https://doi.org/10.1145/1831708.1831711

[9] Jonathan Bell and Gail Kaiser. 2014. Phosphor. Retrieved from https://github.com/gmu-swe/phosphor.

[10] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating dynamic data flow in commodity JVMs. In *Proceedings of the ACM International Conference on Object-oriented Programming Systems Languages & Applications (OOPSLA'14)*. ACM, New York, NY, 83–101. DOI: https://doi.org/10.1145/2660193.2660212

[11] Jonathan Bell and Gail Kaiser. 2015. Dynamic taint tracking for Java with Phosphor (Demo). In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)*. Association for Computing Machinery, New York, NY, 409–413. DOI: https://doi.org/10.1145/2771783.2784768

[12] D. Chandra and M. Franz. 2007. Fine-grained information flow analysis and enforcement in a Java virtual machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE Computer Society, 463–475. DOI: https://doi.org/10.1109/ACSAC.2007.37

[13] Chia Che Tsai, Jeongseok Son, Bhushan Jain, John McAvey, Raluca Ada Popa, and Donald E. Porter. 2020. Civet: An efficient Java partitioning framework for hardware enclaves. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 505–522. Retrieved from https://www.usenix.org/conference/usenixsecurity20/presentation/tsai.

[14] Checkstyle Contributors. 2020. Checkstyle Commit 70c7ae0. Retrieved from https://github.com/checkstyle/checkstyle/commit/70c7ae0e1866074530a49c983d015936a0c2c10f.

[15] Checkstyle Contributors. 2020. Checkstyle Issue #8934. Retrieved from https://github.com/checkstyle/checkstyle/issues/8934.

[16] Checkstyle Contributors. 2020. Checkstyle (version 8.37). Retrieved from https://github.com/checkstyle/checkstyle.

[17] Erika Chin and David Wagner. 2009. Efficient character-level taint tracking for Java. In *Proceedings of the ACM Workshop on Secure Web Services (SWS'09)*. Association for Computing Machinery, New York, NY, 3–12. DOI: https://doi.org/10.1145/1655121.1655125

[18] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, New York, NY, 196–206. DOI: https://doi.org/10.1145/1273463.1273490

[19] James Clause and Alessandro Orso. 2009. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM, New York, NY, 249–260. DOI: https://doi.org/10.1145/1572272.1572301

[20] Closure Compiler Authors. 2014. Google Closure Compiler Commit aac5d11. Retrieved from https://github.com/google/closure-compiler/commit/aac5d11480a0ed3f37919c23a5d3cc210e534bd5.

[21] Closure Compiler Authors. 2014. Google Closure Compiler Issue #652. Retrieved from https://github.com/google/closure-compiler/issues/652.

[22] Closure Compiler Authors. 2014. Google Closure Compiler (version v20140814). Retrieved from https://github.com/google/closure-compiler.

[23] Keith Cooper, Timothy Harvey, and Ken Kennedy. 2006. *A Simple, Fast Dominance Algorithm*. Rice University, CS Technical Report 06-33870. Rice University.

[24] Landon P. Cox, Peter Gilbert, Geoffrey Lawler, Valentin Pistol, Ali Razeen, Bi Wu, and Sai Cheemalapati. 2014. SpanDex: Secure password tracking for Android. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security'14)*. USENIX Association, 481–494. Retrieved from https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/cox.

[25] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. DOI: https://doi.org/10.1145/115372.115320

[26] Dorothy E. Denning and Peter J. Denning. 1977. Certification of programs for secure information flow. *Commun. ACM* 20, 7 (July 1977), 504–513. DOI: https://doi.org/10.1145/359636.359712

[27] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Vancouver, BC. Retrieved from https://www.usenix.org/conference/osdi10/taintdroid-information-flow-tracking-system-realtime-privacy-monitoring.

[28] V. Ganesh, T. Leek, and M. Rinard. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the IEEE 31st International Conference on Software Engineering*. 474–484.

[29] Google LLC. 2020. Guava (version 28.2-jre). Retrieved from https://github.com/google/guava.

[30] H2 Contributors. 2020. H2 Commit 6c564e6. Retrieved from https://github.com/h2database/h2database/commit/6c564e63eb6a3c819eaab19f4aece3298db2ab5f.

[31] H2 Contributors. 2020. H2 Issue #2550. Retrieved from https://github.com/h2database/h2database/issues/2550.

[32] H2 Contributors. 2020. H2 (version 1.4.200). Retrieved from https://github.com/h2database/h2database/.

[33] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. 2006. Using positive tainting and syntax-aware evaluation to counter SQL injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'06/FSE-14)*. ACM, New York, NY, 175–185. DOI: https://doi.org/10.1145/1181775.1181797

[34] Katherine Hough and Jonathan Bell. 2021. A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships (Artifact). DOI: https://doi.org/10.6084/m9.figshare.16611424.v1

[35] Katherine Hough, Gebrehiwet Welearegai, Christian Hammer, and Jonathan Bell. 2020. Revealing injection vulnerabilities by leveraging existing tests. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. Association for Computing Machinery, New York, NY, 284–296. DOI: https://doi.org/10.1145/3377811.3380326

[36] Chen Huo and James Clause. 2014. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 621–631. DOI: https://doi.org/10.1145/2635868.2635917

[37] Kangkook Jee. 2015. *On Efficiency and Accuracy of Data Flow Tracking Systems*. Ph.D. Dissertation. Columbia University. DOI: https://doi.org/10.7916/D8MG7P9D

[38] Jianyu Jiang, Shixiong Zhao, Danish Alsayed, Yuexuan Wang, Heming Cui, Feng Liang, and Zhaoquan Gu. 2017. Kakute: A precise, unified information flow analysis system for big-data security. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*. Association for Computing Machinery, New York, NY, 79–90. DOI: https://doi.org/10.1145/3134600.3134607

[39] Jonathan Hedley. 2018. jsoup: Java HTML Parser (version 1.11.3). Retrieved from https://jsoup.org/.

[40] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. Association for Computing Machinery, New York, NY, 437–440. DOI: https://doi.org/10.1145/2610384.2628055

[41] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Xiaodong Song. 2011. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium*.

[42] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit flows: Can't live with 'Em, can't live without 'Em. In *Proceedings of the 4th International Conference on Information Systems Security(ICISS'08)*. Springer-Verlag, Berlin, 56–70. DOI: https://doi.org/10.1007/978-3-540-89862-7_4

[43] Legion of the Bouncy Castle Inc. 2011. Bouncy Castle Provider (version 1.46). Retrieved from http://bouncycastle.org/.

[44] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

[45] Stephen McCamant and Michael D. Ernst. 2008. Quantitative information flow as network flow capacity. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 193–205. DOI: https://doi.org/10.1145/1375581.1375606

[46] MDN Contributors. 2019. Mozilla Rhino Commit 0c0bb39. Retrieved from https://github.com/mozilla/rhino/commit/0c0bb391647600ec706b1ec66f71831893a6f564.

[47] MDN Contributors. 2019. Mozilla Rhino Issue #539. Retrieved from https://github.com/mozilla/rhino/issues/539.

[48] MDN Contributors. 2019. Mozilla Rhino (version 1.7.11). Retrieved from https://github.com/mozilla/rhino.

[49] Michaël Mera. 2019. Mining constraints for grammar fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. Association for Computing Machinery, New York, NY, 415–418. DOI: https://doi.org/10.1145/3293882.3338983

[50] James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'05)*.

[51] OpenRefine Contributors. 2020. OpenRefine Commit 825e687. Retrieved from https://github.com/OpenRefine/OpenRefine/commit/825e687b0b676fd1be1fa0a9d00be22de0e57060.

[52] OpenRefine Contributors. 2020. OpenRefine Issue #2584. Retrieved from https://github.com/OpenRefine/OpenRefine/issues/2584.

[53] OpenRefine contributors. 2020. OpenRefine (version 3.4-SNAPSHOT). Retrieved from https://github.com/OpenRefine/OpenRefine.

[54] Oracle Corporation. 2019. OpenJDK Java Class Library (version 1.8.0_222). Retrieved from https://openjdk.java.net/.

[55] OW2 Consortium. 2019. ASM (version 7.1). Retrieved from https://asm.ow2.io/.

[56] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android taint analysis tools keep their promises? In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, New York, NY, 331–341. DOI: https://doi.org/10.1145/3236024.3236029

[57] Pivotal Software. 2020. Spring Framework (version 5.2.5). Retrieved from https://spring.io/projects/spring-framework.

[58] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. 2006. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE Computer Society, 135–148. DOI: https://doi.org/10.1109/MICRO.2006.29

[59] Sanjay Rawat, Vivek Jain, Ashish Jith Sreejith Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*.

[60] A. Sabelfeld and A. C. Myers. 2006. Language-based information-flow security. *IEEE J. Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. DOI: https://doi.org/10.1109/JSAC.2002.806121

[61] Tejas Saoji, Thomas H. Austin, and Cormac Flanagan. 2017. Using precise taint tracking for auto-sanitization. In *Proceedings of the Workshop on Programming Languages and Analysis for Security* (Dallas, Texas) *(PLAS'17)*. ACM, New York, NY, 15–24. DOI: https://doi.org/10.1145/3139337.3139341

[62] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy (SP'10)*. IEEE Computer Society, 317–331. DOI: https://doi.org/10.1109/SP.2010.26

[63] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. 2015. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'15)*. Association for Computing Machinery, New York, NY, 227–238. DOI: https://doi.org/10.1145/2750858.2804260

[64] John Singleton. 2018. *Advancing Practical Specification Techniques for Modern Software Systems*. Ph.D. Dissertation. University of Central Florida. Retrieved from http://purl.fcla.edu/fcla/etd/CFE0007099.

[65] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. Association for Computing Machinery, New York, NY, 112–122. DOI: https://doi.org/10.1145/1250734.1250748

[66] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An empirical study of information flows in real-world JavaScript. In *Proceedings of the 14th Workshop on Programming Languages and Analysis for Security (PLAS'19)*. ACM, New York, NY, 15. DOI: https://doi.org/10.1145/3338504.3357339

[67] Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3 (1995), 121–189.

[68] John Toman and Dan Grossman. 2016. Staccato: A bug finder for dynamic configuration updates. In *Proceedings of the 30th European Conference on Object-oriented Programming (ECOOP'16) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 24:1–24:25. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2016.24

[69] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. 2021. White-box analysis over machine learning: Modeling performance of configurable systems. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 1072–1084. DOI: https://doi.org/10.1109/ICSE43902.2021.00100

[70] T. Wang, T. Wei, G. Gu, and W. Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 497–512.

[71] M. Weiser. 1984. Program slicing. *IEEE Trans. Softw. Eng.* SE-10, 4 (1984), 352–357. DOI: https://doi.org/10.1109/TSE.1984.5010248

[72] Bin Xin and Xiangyu Zhang. 2007. Efficient online detection of dynamic control dependence. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*. Association for Computing Machinery, New York, NY, 185–195. DOI: https://doi.org/10.1145/1273463.1273489

[73] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. 28, 2 (Feb. 2002), 183–200. DOI: https://doi.org/10.1109/32.988498

[74] Xiangyu Zhang, Neelam Gupta, and Rajeev Gupta. 2006. Pruning dynamic slices with confidence. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. Association for Computing Machinery, New York, NY, 169–180. DOI: https://doi.org/10.1145/1133981.1134002

[75] Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. *SIGPLAN Not.* 39, 6 (June 2004), 94–106. DOI: https://doi.org/10.1145/996893.996855

[76] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 319–329.

Received July 2021; accepted September 2021