# Towards Fully Automatic Placement of Security Sanitizers and Declassifiers

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Stephen Chong
Harvard University
chong@seas.harvard.edu

## Abstract

A great deal of research on sanitizer placement, sanitizer correctness, checking path validity, and policy inference, has been done in the last five to ten years, involving type systems, static analysis and run-time monitoring and enforcement. However, in pretty much all work thus far, the burden of sanitizer placement has fallen on the developer. However, sanitizer placement in large-scale applications is difficult, and developers are likely to make errors, and thus create security vulnerabilities.

This paper advocates a radically different approach: we aim to fully automate the placement of sanitizers by analyzing the flow of tainted data in the program. We argue that developers are better off leaving out sanitizers *entirely* instead of trying to place them.

This paper proposes a fully automatic technique for sanitizer placement. Placement is *static* whenever possible, switching to *run time* when necessary. Run-time taint tracking techniques can be used to track the source of a value, and thus apply appropriate sanitization. However, due to the run-time overhead of run-time taint tracking, our technique avoids it wherever possible.

***Categories and Subject Descriptors*** D.2.4 [*Software/Program Verification*]: Validation; D.3.4 [*Processors*]: Compilers; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

***General Terms*** Languages, Security, Verification

***Keywords*** Security analysis, vulnerability prevention

## 1. Introduction

Tracking of explicit information flow has received a great deal of attention in recent years. Two primary applications for explicit information flow tracking stand out prominently:

- preventing injection attacks within web applications such as cross-site scripting (XSS) and SQL injection; and
- preventing private data leaks, such as those recently observed in a variety of popular mobile applications [10].



*POPL'13,* January 23–25, 2013, Rome, Italy.


These attacks have motivated a great deal of research in the last five to ten years on sanitizer placement, sanitizer correctness [15, 45], checking path validity, and policy inference [23, 41], involving type systems [8, 32], static analysis [17, 18, 22, 42, 43, 47], and run-time monitoring and enforcement [6, 7, 11, 25].[1]

Much academic work in this space focuses on finding missing sanitizers and is applied to relatively small applications. Several projects have explored the use of run-time techniques, motivated in part by the scalability and precision challenges that static analysis typically encounters. Additional motivation for exploring run-time techniques comes from the complexity of large-scale web applications with multiple, potentially nested sanitizers, which recent assessments [38, 39] suggest is well beyond the ability of developers to address using static reasoning and code reviews.

We also feel that the run-time approach is most practical in the long run. However, the overhead of run-time approaches can be considerable. Prior work on sanitizer placement advocates dynamic sanitizer placement through a combination of *inline* instrumentation [25] and *library-based* instrumentation [6, 7]. The main advantage of library-based instrumentation is reduced overhead: only library code (as opposed to application code) needs to be instrumented. However, library-based approaches do not deal well with information propagated through non-library code and data structures such as `char[]`, `byte[]`, and custom character-level sanitizers. Custom character-level sanitizers are quite common, and sanitizers typically deal with string data at the level of characters [15]. The overhead of these approaches varies, but is generally between 1–20%, depending on the application. In the case of library-based instrumentation, the "depth" of the data propagation path largely determines the overhead. In large enterprise applications, we know that data can undergo a high number of transformations during its lifetime [28], resulting in higher overhead than experiments with smaller applications would lead us to believe. Prior research has proposed the use of pointer analysis as a way to reduce the number of instrumentation points [2, 24]. However, the number of program points that are deemed to be reachable from sources and may flow to sinks is still quite large in practice, leading to a high number of instrumentation points. We feel that it is crucial to develop novel ways to decrease the performance penalty for inline instrumentation to make it practical.

[1] For simplicity, in the rest of this paper, we shall talk primarily about sanitizer placement (for integrity preservation). Our techniques apply equally well to the placement of declassifiers (for confidentiality preservation).

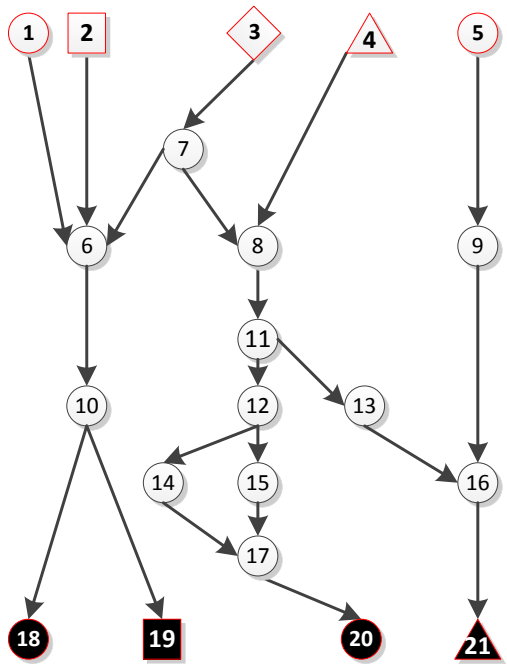


**Figure 1.** Motivating example of a small, but illustrative flow graph. Sources are at the top; sinks are at the bottom.

We aim to *fully automate* the placement of sanitizers by analyzing the flow of tainted data. A key observation is that, given a policy, sources and sinks within the application *induce* restrictions on the placement of sanitizers. It is difficult for developers to place sanitizers so as to satisfy all of these restrictions, especially in large-scale applications [39]. In fact, we argue that developers are better off leaving out sanitizers *entirely*, allowing them to be placed automatically.

In this paper we propose a fully automatic technique for sanitizer placement. The goal is to minimize both run-time overhead and code bloat due to instrumentation. Sanitizer placement is *static* whenever possible, switching to *run-time techniques* when necessary. We perform analyses on the inter-procedural dataflow graph of the program to identify where sanitizers can be placed, and where values must be tracked at run-time in order to determine which sanitizer to apply. In order to reduce run-time overhead, we resort to run-time tracking only when necessary.

### 1.1 Sanitization Policies

Large applications come with libraries of sanitizers. Developers are heavily discouraged from writing their own sanitizers. This is in part because most of the time, they get them wrong [4, 15]. Since sanitizers are implemented as library functions, they are typically pure functions, with type `String` → `String`.

Policies can be given in the form of a table that for every type of data source and data sink indicates the appropriate sanitizer for values that flow from that source to that sink. Policies are declarative specifications, and can both provide developer guidance and simplify the code review process. Section 2 gives examples of policies.

### 1.2 Dataflow Graphs and Policies

Figure 1 shows a simple dataflow graph that will be used as an example throughout this paper. The policy for this example graph is shown in Figure 2. Source types (◯, □, ◇, △) are shown in rows and sink types (●, ■, ▲) are shown in columns. We use ⊘ as a special kind of source type and sink type, for data production or consumption that is not relevant to security (such as constant strings or other trusted sources of data). Thus, we assume that every source and sink of data has a type that appears in the table.

Entries in the table indicate which sanitizer should be applied to data. We use metavariable $\mathcal{P}$ to range over policies, $I$ to range over source types, $O$ to range over sink types, and $S$ to range over sanitizers. We write $\mathcal{P}(I, O)$ for the entry in policy $\mathcal{P}$ for source $I$ and sink $O$. We assume that a node cannot be both a source and a sink, and write $\tau(n)$ for the source type or sink type of node $n$. For example, in Figure 1, where $n_i$ is the node labeled with integer $i$ we have $\tau(n_3) = ◇$ and $\tau(n_{19}) = ■$. Since $n_{11}$ is neither a source nor a sink, $\tau(n_{11})$ is undefined.

| | ● | ■ | ▲ | ⊘ |
|---|---|---|---|---|
| ◯ | $S_1$ | $S_1$ | $S_4$ | $\bot$ |
| □ | $S_1$ | $S_2$ | $\bot$ | $\bot$ |
| ◇ | $S_2$ | $S_1$ | $S_3$ | $\bot$ |
| △ | $\bot$ | $\bot$ | $S_3$ | $\bot$ |
| ⊘ | $\bot$ | $\bot$ | $\bot$ | $\bot$ |

**Figure 2.** Example policy. Sources shown vertically; sinks shown horizontally. $\bot$ means no sanitization required.

For example, let $\mathcal{P}$ be the policy in Figure 2. Data originating from a source of type □ and going to a sink of type ● should have sanitizer $S_1$ applied to it. If $\mathcal{P}(I, O) = \bot$ then no sanitization should be applied data flowing from source type $I$ to sink type $O$. This may indicate, for example, that constant string data should not be sanitized before being displayed to the user.

### 1.3 Contributions

This paper makes the following contributions:

- **Fully automatic sanitizer placement.** We argue that sanitizer placement should be automatic, given a policy and an application, instead of the current approach of the developer being responsible for getting it right.
- **Node-based placement.** We propose a simple node-based strategy for static sanitizer placement. While it is simple to implement and incurs no run-time overhead, it is incorrect for many dataflow graphs.
- **Edge-based placement.** We propose an edge-based strategy for sanitizer placement, which attempts to place sanitizers statically and "spills over" into run time whenever necessary. This strategy is appropriate to use when the simple node-based strategy fails.
- **Correctness.** We define the correctness of sanitization of values in a dataflow graph, and prove that our edge-based strategy for placement is correct.
- **Experiments.** We extensively evaluate how our placement strategies affect the number of instrumentation points on both large applications (up to 1.8 million lines of code) and synthetically generated dataflow graphs. While the node-based approach only instruments a fraction of all nodes, in most cases it fails to provide sanitization on all paths. The edge-based approach, while it requires more sophisticated analysis, provides full sanitization, while reducing the number of instrumentation points by 6.19× on average. Our edge-based technique works even better in the case of a precise underlying dataflow graph: for sparser synthetically generated

graphs, we see a reduction in the number of instrumentation points as high as 27×, compared to the naïve version.

### 1.4 Paper Organization

Section 2 presents examples that highlight the need for automated sanitizer placement. Section 3 gives an overview to our approach for automatic sanitizer placement. Section 4 presents dataflow analyses and algorithms to implement our approach. Section 5 describes our experimental evaluation. Related work is discussed in Section 6. Finally, Sections 7 and 8 describe future work and conclude.

## 2. Motivating Examples

Our examination of large-scale applications has shown that data processing is typically performed via a fixed set of sanitizers whose proper selection depends on the kind of source and sink and can be expressed as a table, as in Figure 2 [39]. Sanitization policies in this section illustrate the complexity of real-world data manipulation scenarios.

### 2.1 Web Applications

The OWASP Enterprise Security API (ESAPI) is an open-source web-application security library. Usage guidelines of ESAPI reveal that the correct sanitization to apply to data depends on how the data will be used, that is, on the sink context. To sanitize a user-provided URL, function `ESAPI.encoder().encodeForURL(`*input*`)` should be used. But to sanitize user input that will be used to construct a CSS attribute, function `ESAPI.encoder().encodeForCSS(`*input*`)` should be used.

| | URL | CSS |
|---|---|---|
| **input** | `encodeForURL` | `encodeForCSS` |

### 2.2 Web Application Roles

In large-scale web applications, sanitization requirements often vary based on who is interacting with the application. This is referred to as *role-based* sanitization. For example, Wordpress allows authors to insert certain HTML tags in their blog posts that commenters may not [46]. Similar approaches are taken by phpBB and Drupal. This complexity is reflected in sanitization libraries such as AntiXSS [27], OWASP HTML Sanitizer Library [30], and HTML Purifier [48], where the developer can select different policies for sanitization of HTML.

This source sensitivity arises because not all users are created equal, and that authentication provides a degree of trust (and increase of capabilities) that is not warranted for non-authenticated users.

### 2.3 Encrypted Cloud

Consider a web application using a public cloud provider for storage. The web application wants to use the cloud for scalability and to reduce storage hardware costs, but does not fully trust the cloud to protect the confidentiality of its data. The application therefore will use encryption when serializing data to the database, and decryption when deserializing.

| | output | cloud |
|---|---|---|
| **input** | ⊥ | `encrypt` |
| **cloud** | `decrypt` | ⊥ |

In this scenario, the sources are of types *input* and *cloud* and sinks are of types *output* and *cloud*. The policy would encrypt data before it goes into the cloud and decrypt it on the way out of the cloud. The correct sanitization to apply (if any) depends on both the source and sink of data.

### 2.4 Mobile App Privacy and Security

Previous studies have shown that applications on Android and other mobile platforms leak user data to untrusted parties. One solution is that the developer needs to filter out private data before it is allowed to go outside [9, 10]. However, the app often has legitimate reasons to send user input and data outside. Consider a gmail app that needs to communicate with its "parent" site, or its *host*, in this case, `mail.google.com`. It is necessary to send information to that hosting URL, including keystrokes, files on the local system if those are to be attached to email, etc. There is perhaps no compelling need to send user data to `AdMob.com`, a third-party mobile advertisement provider whose library is embedded in the app [10], and so data sent to a third-party should be cleansed, i.e., should have sensitive information removed, a form of declassification. This highlights the need to treat the hosting site differently from third-party sites.

Source types for this scenario are *user input*, *data from host*, and *data from 3rd-party site*. Sink types are *screen output*, *isolated app storage*, *send to host*, and *send to 3rd-party site*. Data sent to a third party site that does not originate from the third party site should be cleansed. Also, third-party data being shown to the screen might need to be pretty-printed or checked for integrity in some way, which we we refer to in the table below as `ascii-sanitizer`. No other sanitization or declassification is required.

| | screen output | isolated app storage | to host | to 3rd-party site |
|---|---|---|---|---|
| **user input** | ⊥ | ⊥ | ⊥ | cleanse |
| **host** | ⊥ | ⊥ | ⊥ | cleanse |
| **3rd-party site** | ascii-sanitizer | ⊥ | ⊥ | ⊥ |

### 2.5 Properties of the Placement Problem

**Sink sensitivity:** Sanitization is *sink sensitive*: sanitization to apply to data depends on how the data will be used.

**Source sensitivity:** Sanitization is *source sensitive*: the correct sanitizer to use on data depends on where the data comes from. Source sensitivity also makes full automatic sanitization (as advocated by Samuel et al. [38]) difficult.

**Context-sensitivity:** As elaborated in ScriptGard [39] and by Weinberger et al. [46], sanitization is context-sensitive: to choose the proper sanitizer, the nested context needs to be determined. Consider the following snippet of HTML code, which displays a comment (the value *untrusted*) when the element is clicked.

```
<div class='comment-box'
  onclick='displayComment(untrusted, this)'>
    ... hidden comment ...
</div>
```

The untrusted comment is in two nested contexts: it is in the `onclick` attribute of an HTML tag, and it is in a single-quoted JavaScript string context. To properly sanitize the untrusted comment, we must ensure that the untrusted comment does not contain either JavaScript or HTML metacharacters. In general, more than one sanitizer may be needed on a path between a given source and a sink. We model this using a single function to represent the *composition* of sanitizers, as required.

**Not idempotent or reversible:** Note that sanitizers are *not* guaranteed to be either idempotent or reversible, meaning that we cannot apply them more than once. A recent

study [15] shows that out of 24 sanitizers considered, 19 are idempotent, and that only 2 are reversible. Moreover, order is important, as less than 30% of pairs of sanitizers commute. Finally, over-sanitization is also a significant issue, which often leads to malformed double-encoded data.

## 3. Overview

In this section we define the problem and present an overview of two solutions: a completely static node-based solution, and an edge-based solution that uses static analysis and run-time taint tracking. The static node-based solution incurs no run-time overhead, but doesn't always result in correct sanitization. The edge-based solution is always correct, but may incur run-time overhead due to taint tracking.

### 3.1 Valid Sanitizer Placement Problem

A *dataflow graph* $G = \langle N, E \rangle$ is a directed graph over a set of nodes $N$ with edges $E$ that describes how data flows through a system. Nodes represent computation and/or storage locations, and edges represent the flow of data in the system. As the program performs computation, values traverse the dataflow graph, following edges in the graph, with nodes representing both computation performed on values, and where values are stored during execution. A dataflow graph may have cycles in it. This work is not directly concerned with the precision or soundness of the analysis used to produce the dataflow graph: improvements to the precision and soundness of analyses for dataflow graph construction will seamlessly improve the quality and soundness of our results. Our focus is to ensure correct sanitization of data in a program, and as such we require an interprocedural dataflow graph.

Suppose policy $\mathcal{P}$ describes which sanitizer should be applied to data traversing a dataflow graph. We aim to provide sanitization for values traversing the dataflow graph, according to the following correctness definition.

**Definition 1.** *Given a dataflow graph $G = \langle N, E \rangle$, sanitization for the graph is* valid for policy $\mathcal{P}$ *if for all source nodes $s$, and all sink nodes $t$:*

- *if $\mathcal{P}(\tau(s), \tau(t)) = S$ then every value that flows from $s$ to $t$ has sanitizer $S$ applied exactly once, and no other sanitizer is applied.*
- *if $\mathcal{P}(\tau(s), \tau(t)) = \bot$ then every value that flows from $s$ to $t$ has no sanitizer applied.*

We require that a sanitizer be applied at most once on any given path because sanitizers are not necessarily idempotent [15]: applying it multiple times might result in incorrect sanitization. We require that sanitizers are not applied needlessly. We model multiple (potentially nested) sanitizers as a single (composite) sanitizer.

We consider two strategies for sanitizer placement: a node-based formulation (Section 3.2) that is efficient, but may fail to produce valid sanitization; and an edge-based formulation (Section 3.3) that always provides correct sanitization, but may require run-time taint tracking in order to determine the correct sanitizer to apply.

We assume that the dataflow graph $G = \langle N, E \rangle$ does not contain any node that has both multiple in-coming edges and multiple out-going edges. This assumption is without loss of generality, since if a graph does not satisfy this requirement, it can easily be transformed to one that does by the insertion of synthetic nodes. This assumption is required for the correctness of the edge-based formulation, and is analogous to assumptions in control-flow graph analysis of the absence of *critical edges*: edges that go from nodes with multiple successors to nodes with multiple predecessors.

| | **Possible** | **Exclusive** |
|---|---|---|
| $S_1$ | 1, 2, 3, 6, 7, 10, 18, 19 | 1 |
| $S_2$ | 2, 3, 6, 7, 8, 10, 11, 12, 14, 15, 17, 19, 20 | |
| $S_3$ | 3, 4, 7, 8, 11, 13, 16, 21 | 13 |
| $S_4$ | 5, 9, 16, 21 | 5, 9 |
| $\bot$ | 4, 8, 11, 12, 14, 15, 17, 20 | |

**Figure 3.** $S_i$-possible and $S_i$-exclusive nodes for Figure 1.

### 3.2 Node-based Formulation

We say that a node $n$ is *$S_i$-possible* if it is on a path from a source node $s$ to a sink node $t$ that requires sanitizer $S_i$, that is, $\mathcal{P}(\tau(s), \tau(t)) = S_i$. Thus, if $n$ is $S_i$-possible, then at least some of the data passing through node $n$ requires application of $S_i$. We say a node $n$ is *$S_i$-exclusive* if it is $S_i$-possible, and it is not $S_j$-posible for any $j \neq i$. In other words, node $n$ is $S_i$-exclusive if it is $S_i$-possible, and for any source $s$ and sink $t$, if $n$ is on a path from $s$ to $t$, then that path requires sanitizer $S_i$ (i.e., $\mathcal{P}(\tau(s), \tau(t)) = S_i$).

**Definition 2.** *Node $n \in N$ is* $S_i$-possible *if there is a source node $s$ and sink node $t$ such that $n$ is on a path from $s$ to $t$ and $\mathcal{P}(\tau(s), \tau(t)) = S_i$.*

**Definition 3.** *Node $n \in N$ is* $S_i$-exclusive *if it is $S_i$-possible and for all source nodes $s$ and sink nodes $t$, if $n$ is on a path from $s$ to $t$ then $\mathcal{P}(\tau(s), \tau(t)) = S_i$.*

Figure 3 shows possible and exclusive nodes for sanitizers $S_1$, $S_2$, $S_3$, and $\bot$ for the dataflow graph of Figure 1. Note that while possible nodes are plentiful, exclusive nodes are rarer. In fact, $S_2$ and $\bot$ have no exclusive nodes at all. Note that node $n_{13}$ is on a path both from $n_3$ to $n_{21}$, and from $n_4$ to $n_{21}$. However, it is $S_3$-exclusive because both $\tau(n_3) = \diamondsuit$ and $\tau(n_4) = \triangle$ require the same sanitizer when going to sink $\blacktriangle$: $\mathcal{P}(\diamondsuit, \blacktriangle) = \mathcal{P}(\triangle, \blacktriangle) = S_3$.

For sparser dataflow graphs, exclusive nodes will be more plentiful. Exclusive nodes are good candidates at which to apply a sanitizer to all data passing through the node. However, exclusive nodes are not necessarily unique: there may be multiple $S_i$-exclusive nodes on a single path from a source to a sink. If there are multiple $S_i$-exclusive nodes on a path, we need to choose just one of them at which to apply sanitizer $S_i$. Since in many common applications of data sanitization, a sanitized value is larger than the unsanitized value (e.g., escaping special characters in a string will increase the length of the string), we prefer to perform sanitization as late as possible. We say that node $n$ is *$S_i$-latest-exclusive* if it is $S_i$-exclusive, and for every path going through $n$, it is the last $S_i$-exclusive node on that path.

**Definition 4.** *Node $n \in N$ is* $S_i$-latest-exclusive *if $n$ is $S_i$-exclusive and for every source node $s$ and sink node $t$, and for every path from $s$ to $t$, if $n$ is on that path, then $n$ is the last $S_i$-exclusive node on the path.*

By this definition, we see that in Figure 1 nodes $n_1$, $n_9$, and $n_{13}$ are latest exclusive nodes (for sanitizers $S_1$, $S_4$ and $S_3$ respectively). Node $n_5$ is not an $S_4$-latest exclusive node, since there is another $S_4$-exclusive node later on a path from $n_5$. It is easy to see from the definition that for any path from source node $s$ to sink node $t$ with $\mathcal{P}(\tau(s), \tau(t)) = S_i$, there is at most one $S_i$-latest-exclusive node on that path.

There may, however, be no $S_i$-latest-exclusive node on a path: if there is a path from source node $s$ to sink node $t$ with

$\mathcal{P}(\tau(s), \tau(t)) = S_i$, but there is no $S_i$-latest-exclusive node on that path, then node-based placement will not sanitize values traveling from $s$ to $t$ on that path. Thus, placing sanitizers only at latest-exclusive nodes may fail to produce a valid placement (Definition 1).

As will be seen in Section 5, the static node-based approach does not produce a valid placement for all but simple and sparse dataflow graphs.

### 3.3 Edge-based Formulation

We consider instead an edge-based formulation that is able to always find a correct placement of sanitizers in a dataflow graph, although it may be necessary to record and track at run time some information about the path that a value has taken in the graph in order to determine the correct sanitizer (if any) to apply to the value.

Figure 4 summarizes the key concepts used in our edge-based solution. We provide full definitions and intuition for each of these terms below.

We say that an edge $e$ is *source dependent* if the sanitization to apply to values traversing $e$ depends on which source produced the value.

**Definition 5.** *An edge $e$ is* source dependent *if there exist sources $s_0$ and $s_1$ and sinks $t$ such that $e$ is on a path from $s_0$ to $t$ and on a path from $s_1$ to $t$ and $\mathcal{P}(\tau(s_0), \tau(t)) \neq \mathcal{P}(\tau(s_1), \tau(t))$ (i.e, the sanitizer to use depends on the source).*

Similarly, we say an edge is *sink dependent* if the sanitization to apply to values traversing it depends on which sink the value will go to.

**Definition 6.** *An edge $e$ is* sink dependent *if there exist source $s$ and sinks $t_0$ and $t_1$ such that $e$ is on a path from $s$ to $t_0$ and on a path from $s$ to $t_1$ and $\mathcal{P}(\tau(s), \tau(t_0)) \neq \mathcal{P}(\tau(s), \tau(t_1))$ (i.e., the sanitizer to use depends on the sink).*

Intuitively, if an edge is sink dependent, then when a value traverses the edge, we do not yet know which sanitizer to apply. By contrast, if an edge is source dependent, we do not know which sanitizer to apply to values traversing the edge unless we know from which source the value originated. If an edge is neither source dependent nor sink dependent, then all values traversing the edge are meant to have the same sanitizer applied.

We say that edge $e$ is *source (sink) independent* if it is not source (sink) dependent.

In Figure 1, the edge from node $n_6$ to node $n_{10}$ is both source dependent and sink dependent. It is sink dependent because it is on paths from $n_2$ to both $n_{18}$ and $n_{19}$, but $\mathcal{P}(\tau(n_2), \tau(n_{18})) = S_1 \neq S_2 = \mathcal{P}(\tau(n_2), \tau(n_{19}))$. It is source dependent since it is on paths from both $n_1$ and $n_2$ to $n_{19}$ and $\mathcal{P}(\tau(n_1), \tau(n_{19})) = S_1 \neq S_2 = \mathcal{P}(\tau(n_2), \tau(n_{19}))$.

The edge from node $n_7$ to node $n_8$ is source independent (since only one source node can reach it), but is sink dependent.

#### 3.3.1 Trigger Edges

To apply a sanitizer at a source-dependent edge, we must know from which source a value originated. We can use run-time tracking to "taint" a value so that we can determine its source. However, run-time taint tracking can be expensive, and we do not need to track all values manipulated by the system, just those for which we need to know the source in order to determine which sanitizer to apply.

We identify edges where it is necessary to start run-time tracking of values, and edges where, if we were tracking, it suffices to stop tracking. Edge $e$ is an *in-trigger edge* if it is a source-independent edge but has an edge after it that is source dependent. In-trigger edges are the edges where we have sufficient information to know where a value came from, and need to start run-time tracking because the origin of a value affects which sanitizer to apply. If $e$ is an in-trigger edge from node $n_1$ to $n_2$, then there must be at least one other edge going to node $n_2$, since $n_2$ is a node where paths from different sources merge.

**Definition 7.** *Edge $e$ is an* in-trigger edge *if it is a source-independent edge from node $n_1$ to node $n_2$ such that there exists a source-dependent edge $n_2 \rightarrow n_3$.*

Edge $e$ is an *out-trigger edge* if it is a source-independent edge that is preceded by a source-dependent edge $e'$. If we were tracking run-time values as they traverse edge $e'$, then we no longer need to track them when they traverse edge $e$. If $e$ is an out-trigger edge from node $n_1$ to $n_2$, then there must be at least one other edge leaving $n_1$, since $n_1$ is a node where paths from different sources to different sinks split.

**Definition 8.** *Edge $e$ is an* out-trigger edge *if it is a source-independent edge from node $n_1$ to node $n_2$ such that there exists a source-dependent edge $n_0 \rightarrow n_1$.*

Once we have sanitized a value, we will not need to perform run-time tracking for the value. (This is an invariant that our run-time discipline will enforce: only values that require sanitization and have not yet been sanitized will be tagged at run time.) Because run-time tracking of values can be expensive, we typically want to perform sanitization as early as possible. We can only perform sanitization at sink-independent edges (because at sink-dependent edges, the sanitization to apply depends on the future use of the value). *Sanitization edges* are the earliest possible edges at which we can perform sanitization: they are sink-independent edges that are the earliest sink-independent edge for some path from a source to a sink. That is, if $e$ is a sanitization edge, then for at least one path from a source to a sink, it is the earliest sink-independent edge.

**Definition 9.** *Edge $e$ is a* sanitization edge *if it is a sink-independent edge and there is a source node $s$ and sink node $t$ such that $e$ is the earliest sink-independent edge on a path from $s$ to $t$.*

Figure 5 shows the source-dependent edge, sink-dependent edges, in-trigger edges, out-trigger edges, and sanitization edges for our running example from Figure 1. For example, edge $n_4 \rightarrow n_8$ is an in-trigger edge, since it is source independent, but has a successor edge $n_8 \rightarrow n_{11}$ that is source dependent. Edge $n_{10} \rightarrow n_{19}$ is a sanitization edge as it is the earliest sink-independent edge on the path from node $n_2$ to $n_{19}$. Note that edge $n_{13} \rightarrow n_{16}$ is not a sanitization edge, even though it is sink independent. This is because any path that goes through $n_{13} \rightarrow n_{16}$ must first go through the sink independent edge $n_{11} \rightarrow n_{13}$.

In addition, Figure 5 shows for each edge $e$ the *policy table at $e$*. This is simply the policy table $\mathcal{P}$ restricted to the source types $I$ and sink types $O$ such that $e$ is on a path from a source node of type $I$ to a sink node of type $O$. Policy tables at edges are a useful concept for computing an appropriate placement, and will be used in Section 4.

**Definition 10.** *The* edge policy *at edge $e$ is the restriction of the (global) policy $\mathcal{P}$ to include only source types $I$ and*

| Term | Brief description |
|---|---|
| Source-dependent edge | Sanitizer to apply to values traversing the edge depends on which source type the value came from. |
| Sink-dependent edge | Sanitizer to apply to values traversing the edge depends on which sink type the value will go to. |
| In-trigger edge | Source-independent edge with a source-dependent successor. |
| Out-trigger edge | Source-independent edge with a source-dependent predecessor. |
| Sanitization edge | Earliest sink-independent edge on *some* path from a source to a sink. |
| Tag edge | In-trigger edge that isn't dominated by sanitization edges. Start run-time tagging of values. |
| Untag edge | Either a sanitization edge, or an out-trigger edge that is not dominated by sanitization edges. Stop run-time tagging of values. |
| Carry edge | Edge that (a) is reachable from a tag edge without an intervening untag edge, and (b) can reach an untag edge, and (c) is neither a tag nor an untag edge. Instrument to propagate run-time taint values. |

**Figure 4.** Summary of terms for edge-based placement.

*sink types $O$ such that $e$ is on a path from a source node of type $I$ to a sink node of type $O$. We write $\mathcal{P}_e$ for the edge policy at edge $e$.*

### 3.3.2 Tag, Untag, and Carry edges

In-trigger edges and out-trigger edges help us identify where we may need to start, and can stop, run-time tracking of values. However, we can refine these notions to reduce the amount of run-time tracking we must perform.

Intuitively, run-time tracking is necessary only when a sanitization edge needs to distinguish values coming from different sources. These are exactly the sanitization edges that are source dependent. We need to propagate taint information only along source-dependent edges, and only until we sanitize the value.

This also means that we only need to *start* taint tracking (which we refer to as "tagging" data) when data values move from a source-independent edge to a source-dependent edge and the data is not yet sanitized (and will need sanitization in the future). Similarly, we can *stop* taint tracking (which we refer to as "untagging" data) when tagged data is sanitized, or when it moves from a source-dependent edge to a source-independent edge.

Specifically, a *tag edge* (where we tag values at run time, and start the run-time taint tracking) are in-trigger edges such that a value traversing the edge might be unsanitized and require sanitization in the future. A value is unsanitized if it has not gone through a sanitization edge, and thus the tag edges are in-trigger edges that are not dominated[2] by a sanitization edge. Note that an edge dominates itself, and thus a tag edge cannot also be a sanitization edge.

**Definition 11.** *Edge $e$ is a* tag edge *if $e$ is an in-trigger edge that is not dominated by sanitization edges.*

An *untag edge* is an edge such that a tagged value can reach it (i.e., it is not dominated by sanitization edges), and we no longer need to track the tagged values. It is either a sanitization edge (since after sanitization we no longer need to track taint), or an out-trigger edge.

**Definition 12.** *An* untag edge *is either (a) an out-trigger edge that is not dominated by a sanitization edge; or (b) a sanitization edge.*

At tag edges we tag values and start taint tracking, and continue taint tracking the tagged value until it reaches an untag edge: if the untag edge is a sanitization edge, we apply the appropriate sanitizer; otherwise, the untag edge is an out-trigger edge and we can stop taint tracking. (Note that if we stop taint-tracking a value at an untag edge, we may potentially resume taint tracking if the value later encounters another in-trigger edge not dominated by sanitization edges.) Edges between tag edges and untag edges will need to propagate tag values. We refer to these edges as *carry edges*.

[2]We define domination in dataflow graphs as follows. Edge $e$ is *dominated* by edge $e'$ if any path from any source that ends with edge $e$ must contain $e'$.

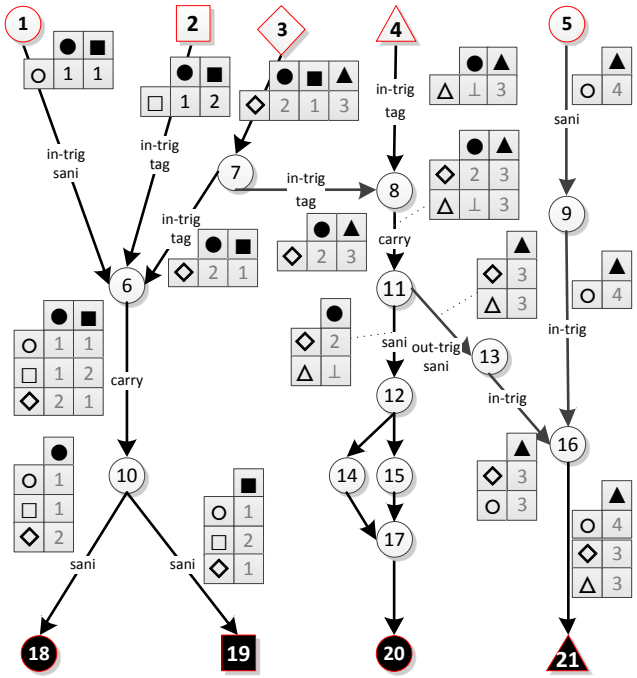


**Figure 5.** Policy tables are shown at every node and trigger edges are marked.

**Definition 13.** *Edge $e$ is a* carry edge *if $e$ is on a path from a tag edge to an untag edge such that the path does not contain an untag edge. That is, if edges $e_0, \ldots, e_n$ are a path where $e_0$ is a tag edge, $e_n$ is an untag edge, and $e_1, \ldots, e_{n-1}$ are not tag edges, then edges $e_1, \ldots, e_{n-1}$ are carry edges.*

In Figure 5, edge $n_7 \rightarrow n_6$ is a tag edge: it is an in-trigger edge (since it is source independent and successor edge $n_6 \rightarrow n_{10}$ is source dependent) that is not dominated by sanitizer edges. By contrast, edge $n_{13} \rightarrow n_{16}$ is an in-trigger edge, but it is not a tag edge, since it is dominated by sanitizer edge $n_{11} \rightarrow n_{13}$. This means that any values traversing $n_{13} \rightarrow n_{16}$ will already be sanitized, and so there is no need to track their source type in order to determine

which sanitizer to apply. In Figure 5, all untag edges are sanitization edges.

Edge $n_6 \to n_{10}$ is a carry edge, as it is on a path from tag edge $n_1 \to n_6$ to untag edge $n_{10} \to n_{18}$ without an intervening untag edge. Edge $n_6 \to n_{10}$ will propagate the tags that tag edges $n_1 \to n_6$, $n_2 \to n_6$, and $n_7 \to n_6$ create, and enable sanitization edges $n_{10} \to n_{18}$ and $n_{10} \to n_{18}$ to apply the appropriate sanitization.

#### 3.3.3 Run-Time Taint Tracking for Sanitization

We have defined several different kinds of edges that are relevant to the run-time discipline for applying correct sanitization to values: sanitization edges, tag edges, untag edges, and carry edges. We summarize what the instrumentation for these edges is required to do at run time:

- **tag edge:** when a value traverses a tag edge, if the value is not tagged then tag the value with *one of the source types* reaching it. A reaching source node, and its type, can be statically determined by examining the edge policy. Multiple source types may reach the tag edge, and any can be used, since tag edges are source independent.
- **untag edge:** when a tagged value traverses an untag edge, untag it.
- **sanitization edge:** If the sanitization edge is not preceded by a carry edge, then no tagged values can reach this edge, and all values traversing this edge should have the same sanitizer applied. Otherwise, apply sanitization only if the value is tagged by looking up the tag (which is a source type) in the edge's policy table to find the appropriate sanitizer to apply (which might be $\perp$, in which case no sanitization is applied).
- **carry edge:** when a value traverses a carry edge, any taint on the value must be propagated.

For example, in Figure 5, consider a value flowing from source node $n_3$ to sink node $n_{20}$. At tag edge $n_7 \to n_8$, the value will be tagged with its originating source type $\tau(n_3) = \diamondsuit$. The value with its tag will be propagated over carry edge $n_8 \to n_{11}$. Upon reaching sanitization edge $n_{11} \to n_{12}$, its tag will be examined, and the appropriate sanitizer ($S_3$) applied. Note that the tag was needed for $n_{11} \to n_{12}$ to determine which sanitizer to apply, since a value from source node $n_4$ could also traverse that edge, requiring no sanitization ($\perp$).

Note that if an edge $e$ is not a tag edge, untag edge, sanitization edge, or carry edge, then $e$ requires no instrumentation, as no tagged value will traverse $e$, and $e$ does not need to perform any tagging, untagging, or sanitization. For example, in Figure 5, edges between nodes $n_{12}$ and $n_{20}$ require no instrumentation.

#### 3.3.4 Correctness of Edge-based Placement

The edge-based placement produces a valid placement (Definition 1). We present here the key lemma that proves this.

**Lemma 1.** *For every path from a source node $s$ to a sink node $t$, the following conditions hold for values flowing along that path.*

1. *There is at least one sanitization edge on the path.*
2. *If $\mathcal{P}(\tau(s), \tau(t)) = \perp$ then no sanitization will be applied.*
3. *If $\mathcal{P}(\tau(s), \tau(t)) \neq \perp$ then sanitizer $\mathcal{P}(\tau(s), \tau(t))$ will be applied at the first sanitization edge.*
4. *No sanitization will be applied at the second or subsequent sanitization edges.*

*Proof:* Condition (1) holds from the definition of sanitization edges, and because an edge whose target is a sink node must be sink independent.

Let $e_0, \ldots, e_n$ be a path from a source node to a sink node, and let $e_i$ be the first sanitization edge. Conditions (2) and (3) hold by the following argument. If $e_i$ is source independent then all values traversing $e_i$ will have the same sanitization applied (either sanitizer $S$ if $\mathcal{P}(\tau(s), \tau(t)) = S$, or no sanitization if $\mathcal{P}(\tau(s), \tau(t)) = \perp$). Suppose that $e_i$ is source dependent. Then there must be some other source $s'$ such that there is a path from $s'$ to $e_i$ and some output $t'$ reachable from $e_i$ such that $\mathcal{P}(\tau(s), \tau(t')) \neq \mathcal{P}(\tau(s'), \tau(t'))$. Since $e_0$ is source independent and $e_i$ is source dependent and the first sanitizer edge, there must be some edge $e_j$ on the path $e_0, \ldots e_{i-1}$ such that $e_j$ is a tag edge, and all edges $e_{j+1}, \ldots, e_{i-1}$ are carry edges. Thus, at $e_j$, the value will be tagged with source type $\tau(s)$ (or some other source type $I$ such that $\mathcal{P}(\tau(s), \tau(t)) = \mathcal{P}(I, \tau(t))$), the carry edges will propagate this tag, and so at sanitization edge $e_i$, the correct sanitization will be applied.

Suppose that condition (4) does not hold. Then there is some edge $e_k$ in path $e_{i+1}, \ldots, e_n$ such that $e_k$ is a sanitization edge, and $e_k$ applies sanitization to values traversing path $e_o, \ldots, e_n$. Since $e_k$ is a sanitization edge, it is the earliest sink-independent edge on some path from source to a sink, and so there must be some other source node $s'$ that can reach $e_k$. Since $e_k$ is the first sink-independent edge on a path from $s'$, there must be another edge leaving $\mathit{source}(e_k)$ (where $\mathit{source}(e)$ denotes the source node of edge $e$) such that on that edge, some sink node $t'$ is reachable that is not reachable from $e_k$, and $\mathcal{P}(\tau(s'), \tau(t)) \neq \mathcal{P}(\tau(s'), \tau(t'))$. More over, since $\mathit{source}(e_k)$ has multiple edges coming from it, by assumption that the dataflow graph has no nodes with both multiple successors and multiple predecessors, node $\mathit{source}(e_k)$ has a single predecessor, edge $e_{k-1}$, and so $e_{k-1}$ is also on the path from $s'$ to $e_k$. Therefore, edge $e_{k-1}$ must be source dependent: since $e_i$ can reach $t'$, and $e_i$ is sink independent, it means that either $\mathcal{P}(\tau(s), \tau(t')) \neq \mathcal{P}(\tau(s'), \tau(t'))$ or $\mathcal{P}(\tau(s), \tau(t)) \neq \mathcal{P}(\tau(s'), \tau(t))$.

Now consider whether $e_{k-1}$ is a carry edge. **Suppose $e_{k-1}$ is a carry edge.** We will show that none of the edges on $e_i, ..., e_{k-1}$ can be a tag edge, and thus, a value coming from $e_i$ cannot be tagged, and so at sanitizer $e_k$, no sanitization will be applied. This contradicts the assumption that condition (4) doesn't hold.

Note that $e_i$ is not a tag edge, as it is a sanitization edge. Then there must be some tag edge $e_m$ between $e_i$ and $e_{k-1}$. Since it is not dominated by sanitizer edges, there must be a path from a source node $s_0$ (such that $\tau(s_0) \neq \tau(s)$) to $\mathit{source}(e_m)$ without a sanitization edge. Since $e_m$ is an intrigger edge, it is source independent. That means that for all sink types $O$ reachable from $e_m$, and all source types $I$ that can reach $e_m$, we have $\mathcal{P}_{e_m}(\tau(s), O) = \mathcal{P}_{e_m}(I, O)$. But any sink type $O$ reachable from $e_m$ is also reachable from $e_i$, and $e_i$ is sink independent. That means that for any sink types $O_1$ and $O_2$ we have $\mathcal{P}_{e_m}(\tau(s), O_1) = \mathcal{P}_{e_m}(\tau(s), O_2)$. Together these imply that $e_m$ is sink independent, since for any source $I$ that can reach $e_m$ and sinks $O_1$ and $O_2$ that can be reached from $e_m$ we have:

$$
\begin{aligned}
\mathcal{P}_{e_m}(I, O_1) &= \mathcal{P}_{e_m}(\tau(s), O_1) && e_m \text{ is source independent} \\
&= \mathcal{P}_{e_i}(\tau(s), O_1) && O_1 \text{ is reachable from } e_i \\
&= \mathcal{P}_{e_i}(\tau(s), O_2) && e_i \text{ is sink independent,} \\
& && \text{and } O_2 \text{ is reachable from } e_i
\end{aligned}
$$

$$= \mathcal{P}_{e_m}(\tau(s), O_2)$$
$$= \mathcal{P}_{e_m}(I, O_2) \qquad e_m \text{ is source independent}$$

But then $e_m$ is the first sink-independent edge on the path from $I_0$ to $e_m$, and so it is a sanitization edge. This is a contradiction, as $e_m$ is a tag edge.

**Suppose $e_{k-1}$ is not a carry edge.** Edge $e_k$ is an untag edge (since it is a sanitization edge). Note that $e_{k-1}$ is not a tag edge, since it is source dependent. But since $e_{k-1}$ is source dependent, and $e_k$ is the first sanitization edge on the path from $s'$ to $e_k$, then there must be a tag edge on the path from $s'$ to $e_k$ without any intervening untag edges between it and $e_k$. Therefore $e_{k-1}$ is a carry edge, which is a contradiction. ∎

The correctness of the edge-based placement follows trivially from Lemma 1 and the fact that no edge other than a sanitizer edge applies sanitization.

#### 3.3.5 Optimizations

There are several opportunities for optimization in the edge-based placement approach.

**Remove un-needed sanitization edges:** For simplicity of the presentation and the proof, we have defined the behavior of tag edges and sanitization edges treating "no sanitization" $\perp$ as if it were a sanitizer. If a sanitizer is not preceded by a carry edge, and the policy dictates that no sanitization should be applied, then the sanitization edge does not perform any computation, and should not be instrumented. Similarly, if a tag edge is tagging a value with a source type that will never require sanitization, then the tag edge can be removed, and the value never tagged. This optimization is valid because a sanitization edge that may receive tagged values will never sanitize an untagged value.

**Sanitization edges preceded by carry edges:** For simplicity we required that any sanitization edge preceded by a carry edge needed to check the run-time tag before applying sanitization. There are some situations (statically determinable) where a sanitization edge will be preceded by a carry edge, yet all values going through it should have the same sanitization applied. In Figure 5 edge $n_{11} \to n_{13}$ is an example of this: the preceding edge $n_8 \to n_{11}$ is a carry edge, but $n_{11} \to n_{13}$ is source independent and the first sanitization edge on any path that goes through it. Thus, all values traversing $n_{11} \to n_{13}$ will have sanitizer $S_3$ applied, so there is no need to examine the tag.

**Attaching tags to run-time values:** We envision the run-time taint tracking being implemented simply by attaching tags to run-time values. This is a strategy that works well for dynamic languages such as Java, PHP, or JavaScript. The tags can be quite compact: we have described it above as tagging a value with the source type that it originated from (or a source type with equivalent sanitization requirements), but it would suffice to use bit strings that uniquely identify a source type. The number of sources depends on the policy, but will typically be small, meaning that a tag of 3–4 bits would suffice. There are opportunities for efficient implementation of taint-tracking when the tags are this small, such as placing the tag within the value header at run time. With this tagging approach, instrumentation for carry edges becomes trivial, since tags will be copied if they exist. Thus, the only instrumentation required will be to tag values as they traverse tag edges, untag them as they traverse untag edges, and apply sanitization at sanitization edges.

| | | | |
|---|---|---|---|
| Semi-lattice | $L$ | set of source types | |
| Top | $\top$ | $\emptyset$ | |
| Initial value | $init(n)$ | $\emptyset$ | |
| Transfer function | $TF(n)$ | add $\tau(n)$ to set<br>*identity* | if $n$ is a source<br>otherwise |
| Meet operator | $\sqcap(x, y)$ | union $x \cup y$ | |
| Direction | | *forward* | |

(a) Available source types.

| | | | |
|---|---|---|---|
| Semi-lattice | $L$ | set of sink types | |
| Top | $\top$ | $\emptyset$ | |
| Initial value | $init(n)$ | $\emptyset$ | |
| Transfer function | $TF(n)$ | add $\tau(n)$ to set<br>*identity* | if $n$ is a sink<br>otherwise |
| Meet operator | $\sqcap(x, y)$ | union $x \cup y$ | |
| Direction | | *backward* | |

(b) Anticipated sink types.

**Figure 6.** Available source types and anticipated sink types.

**Efficient lookup for sanitization:** Since the number of possible tags that can reach a given sanitization edge is small and known statically, we can pre-compute a lookup table for each sanitization edge that maps the tag number to the required sanitizer, thus minimizing run-time calculations.

**Early vs. late sanitizer placement:** The static node-based placement strategy performs sanitization as late as possible, at latest-exclusive nodes. The edge-based placement performs sanitization as early as possible, at the earliest sink-independent edges. The reason for this difference is that for the purely static node-based placement, it is slightly better to perform sanitization late, as many common sanitizers increase the size of data, and thus place additional pressure on memory. By contrast, for edge-based placement, early sanitization will reduce the amount of run-time taint tracking required, and we believe the cost of any run-time taint tracking outweighs the cost of increased size of data from sanitization.

## 4. Placement Algorithms

In this section, we propose concrete algorithms for computing the sets and relations described in Section 3. At the core of these computations, we have dataflow analysis, as described in Aho et al. [1]. As we will see, we can often stage our computation and break it down into a series of two or three analyses, one after another. As Knoop et al. [20] observe, this is often advantageous compared to a more complex equation-based approach, because each analysis stage completes quickly.

### 4.1 Node-based Placement

To implement the node-based placement strategy we compute the set of nodes that are $S_i$-possible and $S_i$-exclusive for each sanitizer $S_i$ for $i$ ranging from 1 to $k$. To combine the computation of these properties for different sanitizers $S_i$, we use *bit vectors* as our representation. Generally, a 1 at position $i$ for a value at node $n \in N$ means that the property (either possibility or exclusiveness) holds for $S_i$.

First, we compute *available* source types and *anticipated* sink types at every node using a dataflow analysis, as shown in Figure 6. We specify dataflow analyses by giving the semi-

| | | |
|---|---|---|
| Semi-lattice | $L$ | bit vector of length $k$ |
| Top | $\top$ | $\bar{0}$ |
| Initial value | $init(n)$ | $\bar{0}$ |
| Transfer function | $TF(n)$ | $\begin{cases} \text{bit } i = 1 & \text{if } n \text{ is } S_i\text{-exclusive} \\ identity & \text{otherwise} \end{cases}$ |
| Meet operator | $\sqcap(x, y)$ | bitwise or $x \vert y$ |
| Direction | | *backward* |

**Figure 7.** Computes exclusive_anticipated($i$).

lattice of dataflow facts, the initial values of start nodes (source nodes for forward analyses, sink nodes for backwards analyses), the transfer function for nodes, and the direction of the dataflow analysis. This is a complete specification of the dataflow analyses.

We then combine the *available* sources and *anticipated* sinks information as described in Algorithm 1, to determine for each node which sanitizers are possible at every node. This is done by *projecting* the policy table to only the available source types and anticipated sink types. We write $Project(\mathcal{P}, S, T)$ for the policy table that contains only the rows of policy table $\mathcal{P}$ for sink types $S$, and only the columns of $\mathcal{P}$ for sink types $T$. If at node $n$, sanitizer $S_i$ appears in policy table $Project(\mathcal{P}, available(n), anticipated(n))$, then $n$ is $S_i$-possible.

**Algorithm 1. Possible nodes.**

```
for all n ∈ N do
  for all S_i ∈ Project(P, available(n), anticipated(n)) do
    possible(S_i) = possible(S_i) ∪ {n}
```

The nodes that are $S_i$-exclusive are a subset of nodes that are $S_i$-possible. Computing $S_i$-exclusive nodes is a simple matter of removing from the set of $S_i$-possible nodes any node that is $S_j$-possible, for any $i \neq j$, as shown in Algorithm 2.

**Algorithm 2. Exclusive nodes.**

```
for all i ∈ [1..k] do
  exclusive(S_i) = possible(S_i)
  for all j ∈ [1..k] do
    if i ≠ j then
      for all n ∈ possible(S_i) do
        if n ∈ possible(S_j) then
          exclusive(S_i) = exclusive(S_i) \ {n}
```

The last step is to compute latest-exclusive nodes: $S_i$-exclusive nodes that for some path from a source to a sink are the last $S_i$-exclusive node on that path. Figure 7 describes a backward dataflow analysis that identifies, for each $S_i$, which nodes can reach an $S_i$-exclusive node. We write exclusive_anticipated($S_i$) for the set of nodes that can reach a $S_i$-exclusive node. Latest-exclusive nodes are simply the set of exclusive nodes, minus the set of anticipated-exclusive nodes (Algorithm 3).

**Algorithm 3. Latest-exclusive nodes**

```
for all i ∈ [1..k] do
  latest_exclusive(S_i) = exclusive(S_i)
  for all n ∈ exclusive_anticipated(S_i) do
    latest_exclusive(S_i) = latest_exclusive(S_i) \ {n}
```

We place sanitizer $S_i$ at all nodes that are $S_i$-latest-exclusive. Latest-exclusive nodes may be rare, especially in dense graphs. For the graph in Figure 1, this algorithm will place sanitizers only at nodes $n_1$, $n_{13}$, and $n_9$. However, this is clearly insufficient, because not all values traversing the graph will be sanitized, such as values flowing from source node $n_3$ to sink node $n_{20}$.

| | | |
|---|---|---|
| Semi-lattice | $L$ | *Bool* |
| Top | $\top$ | *true* |
| Initial value | $init(n)$ | *true* |
| Transfer function | $TF(n)$ | $\begin{cases} true & \text{if } \exists S_i.\ n \in \text{latest\_exclusive}(S_i) \\ \text{identity} & \text{otherwise} \end{cases}$ |
| Meet operator | $\sqcap(x, y)$ | conjunction $x \wedge y$ |
| Direction | | *forward* |

**Figure 8.** Detect whether static placement is valid.

Figure 8 describes a dataflow analysis to detect whether all paths from sources to sinks go through a latest-exclusive node. Dataflow facts are booleans, indicating whether all paths to the node have gone through a latest-exclusive node. (By construction, a path can have at most one latest-exclusive node, so there is no need to count the number of latest-exclusive nodes on a path.) The static placement is valid if and only if the dataflow analysis produces a value of *true* at all sink nodes. If the static placement is valid, then it can be used to correctly sanitize all values, with no run-time overhead. If the placement is not valid, then the edge-based placement can be used to ensure correct sanitization, albeit with some run-time overhead.

### 4.2 Edge-based Placement

To implement the edge-based solution, we must identify several different sets of edges, summarized in Figure 4. We present algorithms to compute each of these sets of edges.

**Source-dependent edges and sink-dependent edges:** First, we compute the available source types and anticipated sink types for every edge, similar to the dataflow analyses in Figure 6. However, whereas Figure 6 computes dataflow facts for nodes, we need to compute dataflow facts for edges.

Next, for each edge $e$ we compute edge policy $\mathcal{P}_e$: policy table $\mathcal{P}$ restricted to the available source types and anticipated sink types of edge $e$. We use edge policies to identify source-dependent edges and sink-dependent edges. Edge $e$ is source dependent if and only if $\mathcal{P}_e$ has more than one unique sanitizer in any column. Edge $e$ is sink dependent if and only if $\mathcal{P}_e$ has more than one unique sanitizer in any row.

**In-trigger and out-trigger edges:** Recall that in-trigger edges are source-independent edges with a source-dependent successor edge, and out-trigger edges are source-independent edges with a source-dependent predecessor edge. We can compute these edges efficiently simply by inspection of the dataflow graph. Let in_trigger denote the set of in-trigger edges, and out_trigger denote the set of out-trigger edges.

**Sanitization edges:** Sanitization edges are sink-independent edges that are the earliest sink-independent edge on *some* path from a source to a sink. They are the edges at which sanitization will be performed: at sink-independent edges the sanitization to apply to a value does not depend on which sink the value will go to.

Figure 9 presents a dataflow algorithm for computing sanitization edges. Note that the analysis computes dataflow facts for edges. Dataflow facts are pairs of boolean values. The first value is true for an edge if and only if all paths to the edge go through a sink-independent edge. The second boolean value is true for sanitization edges: edges that are the first sink-independent edge on some path, which is exactly the edges that are sink independent and have at least

| | | |
|---|---|---|
| Semi-lattice | $L$ | $Bool \times Bool$ |
| Top | $\top$ | *(true, true)* |
| Initial value | $init(e)$ | *(false, false)* |
| Transfer function | $TF(e)$ | $(f_1(e), f_2(e))$ |
| | $f_1(e)(a,b)$ | $= \begin{cases} true & \text{if } e \text{ is sink-independent} \\ a & \text{otherwise} \end{cases}$ |
| | $f_2(e)(a,b)$ | $= \begin{cases} true & \text{if } f_1(e) = true \text{ and } a = false \\ false & \text{otherwise} \end{cases}$ |
| Meet operator | $\sqcap(x,y)$ | pointwise $\wedge$ |
| Direction | | *forward* |

**Figure 9.** Computes sanitization($e$).

| | | | |
|---|---|---|---|
| Semi-lattice | $L$ | *Bool* | |
| Top | $\top$ | *true* | |
| Initial value | $init(e)$ | *false* | |
| Transfer function | $TF(e)$ | $\begin{cases} true & \text{if } e \text{ is a sanitization edge} \\ identity & \text{otherwise} \end{cases}$ | |
| Meet operator | $\sqcap(x,y)$ | $x \wedge y$ | |
| Direction | | *forward* | |

**Figure 10.** Computes dom_sani($e$): whether edge $e$ is dominated by sanitization edges.

one path to it that does not go through a sink-independent edge. Note that the transfer function for edge $e$ is given as a pair of functions, $f_1(e)$ and $f_2(e)$, each of which is a function from the input dataflow fact (a pair of boolean values, $(a, b)$) to a boolean value.

**Tag and untag edges:** Tag and untag edges are where we, respectively, start and stop run-time tracking of values. The definition of both tag and untag edges relies on identifying edges that are dominated by a sanitization edge, for which we use the dataflow analysis in Figure 10. We write dom_sani($e$) if edge $e$ is dominated by sanitization edges. If $e$ is a sanitization edge, then dom_sani($e$) is true.

The following algorithm computes the set of tag and untag edges. Tag edges are in-trigger edges that are not dominated by sanitization edges. Untag edges are either sanitization edges, or out-trigger edges that are not dominated by sanitization edges.

**Algorithm 4. Tag and untag edges.**

```
for all e ∈ E do
  if e ∈ in_trigger ∧ ¬dom_sani(e) then
    tag = tag ∪ {e}
  if (e ∈ out_trigger∧¬dom_sani(e))∨e ∈ sanitization then
    untag = untag ∪ {e}
```

**Carry edges:** Finally, carry edges are those on a path from a tag to an untag edge that does not pass through an untag edge. The set of carry edges can be computed by first performing a forward dataflow analysis to compute tag_available—the set of edges that are reachable from a tag edge without an intervening untag edge—and then performing a backward dataflow analysis to compute untag_anticipated, the set of edges that can reach an untag edge. These dataflow analyses are shown in Figure 11. Carry edges are the non-tag, non-untag edges that are in both the

| | | | |
|---|---|---|---|
| Semi-lattice | $L$ | *Bool* | |
| Top | $\top$ | *false* | |
| Initial value | $init(e)$ | *false* | |
| Transfer function | $TF(e)$ | $\begin{cases} true & \text{if } e \in \text{tag} \\ false & \text{if } e \in \text{untag} \\ identity & \text{otherwise} \end{cases}$ | |
| Meet operator | $\sqcap(x,y)$ | $x \vee y$ | |
| Direction | | *forward* | |

(a) Computes tag_available: reachable from tag edge without intervening untag edge.

| | | | |
|---|---|---|---|
| Semi-lattice | $L$ | *Bool* | |
| Top | $\top$ | *false* | |
| Initial value | $init(e)$ | *false* | |
| Transfer function | $TF(e)$ | $\begin{cases} true & \text{if } e \in \text{untag} \\ identity & \text{otherwise} \end{cases}$ | |
| Meet operator | $\sqcap(x,y)$ | $x \vee y$ | |
| Direction | | *backward* | |

(b) Computes untag_anticipated: can reach untag edge.

**Figure 11.** Dataflow analyses for carry edge computation.

| Benchmark | DLLs | DLL (KB) | LOC |
|---|---|---|---|
| Alias Management | 3 | 65 | 10,812 |
| Chat Application | 3 | 543 | 6,783 |
| Bicycle Club App | 3 | 62 | 14,529 |
| Software | 15 | 118 | 11,941 |
| Sporting Field Management | 3 | 290 | 15,803 |
| Commitment Management | 7 | 369 | 25,602 |
| New Hire | 11 | 565 | 5,595 |
| Expense Report Approval | 4 | 421 | 78,914 |
| Customer Support Portal | 14 | 2,447 | 66,385 |
| Relationship Management | 5 | 3,345 | 1,810,585 |

**Figure 12.** Benchmark applications, sorted by code size.

tag_available and untag_anticipated sets, as defined in the following algorithm.

**Algorithm 5. Carry edges.**

```
for all e ∈ E do
  if e ∈ tag_available ∧ e ∈ untag_anticipated then
    if e ∉ tag ∧ e ∉ untag then
      carry = carry ∪ {e}
```

## 5. Experimental Evaluation

Our evaluation focuses on comparing the number of instrumentation points using our sanitizer placement algorithms compared to a baseline implementation that performs dynamic taint tracking between all sources and sinks. Our target applications are long-running server applications. Runtime overhead of taint tracking is typically workload specific (e.g., [10]), so we choose not to evaluate run-time overhead directly. Reducing the number of instrumentation points is a valuable goal, since long-running applications with diverse workloads have high code coverage over time, hitting increasingly many instrumentation points. Section 5.1 presents the results of applying our techniques to large C# web applications written in ASP.NET. Section 5.2 evaluates our approach against large, synthetically constructed graphs.

### 5.1 Large Applications

Figure 12 contains a summary of information about our macro-benchmarks. These are relatively large business web

| Application | Graph nodes | Taint sources | Taint sinks | Tainted nodes forward | Tainted nodes backward | Tainted nodes both | Tainted nodes ratio | Exclusive nodes all | Exclusive nodes l.e. | Exclusive nodes ratio | Sanitization coverage |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Terralever | 156 | 64 | 69 | 140 | 140 | 76 | 48% | 122 | 50 | 32% | 82% |
| Alias Management | 59 | 11 | 12 | 22 | 21 | 11 | 18% | 19 | 9 | 15% | 86% |
| Contoso Bicycle Club | 161 | 50 | 54 | 145 | 133 | 87 | 54% | 94 | 40 | 24% | 50% |
| Windows Experience Catalog | 204 | 47 | 83 | 186 | 160 | 101 | 49% | 120 | 49 | 24% | 93% |
| Commitment Management | 356 | 135 | 132 | 299 | 296 | 183 | 51% | 221 | 86 | 24% | 79% |
| New Hire | 502 | 142 | 183 | 401 | 409 | 229 | 45% | 275 | 110 | 21% | 70% |
| Expense Report Approval | 805 | 214 | 322 | 722 | 637 | 408 | 50% | 389 | 170 | 21% | 82% |
| Customer Support Portal | 3,881 | 967 | 1,219 | 3,488 | 3,263 | 2,266 | 58% | 1,721 | 770 | 19% | — |
| Relationship Management | 3,639 | 1,054 | 982 | 3,321 | 3,104 | 2,241 | 61% | 1,565 | 637 | 17% | — |

**Figure 13.** Node-based analysis and its effectiveness.

| Application | Total edges | Taint sources | Taint sinks | Tainted forward | Tainted backward | Tainted both | Tainted ratio | Dependent source | Dependent sink | Triggers in | Triggers out | Triggers sanitizer | Edge count tag | Edge count untag | Edge count carry | Instr. total | Instr. ratio |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Terralever | 142 | 96 | 89 | 140 | 137 | 136 | 95% | 3 | 1 | 8 | 0 | 97 | 1 | 5 | 0 | 6 | **22.66** |
| Alias Management | 962 | 11 | 10 | 13 | 14 | 13 | 1% | 0 | 0 | 0 | 0 | 11 | 0 | 2 | 0 | 2 | **6.5** |
| Contoso Bicycle Club | 182 | 80 | 70 | 170 | 174 | 164 | 90% | 32 | 27 | 29 | 9 | 84 | 2 | 6 | 7 | 15 | **10.93** |
| Windows Experience | 430 | 68 | 162 | 386 | 364 | 358 | 83% | 133 | 125 | 14 | 117 | 193 | 2 | 31 | 123 | 156 | 2.29 |
| Commitment Management | 461 | 177 | 188 | 420 | 409 | 386 | 83% | 110 | 114 | 45 | 41 | 215 | 6 | 30 | 108 | 144 | 2.68 |
| New Hire | 873 | 258 | 347 | 680 | 771 | 652 | 74% | 181 | 192 | 108 | 75 | 342 | 27 | 67 | 126 | 220 | 2.96 |
| Expense Report Approval | 1,389 | 367 | 503 | 1,286 | 1,296 | 1,208 | 86% | 189 | 197 | 131 | 143 | 577 | 23 | 99 | 144 | 266 | **4.54** |
| Customer Support Portal | 8,985 | 1,505 | 2,167 | 8,534 | 8,469 | 8,069 | 89% | 4,315 | 4,364 | 901 | 1,331 | 2,875 | 232 | 541 | 3,917 | 4,690 | 1.72 |
| Relationship Management | 17,732 | 2,376 | 2,594 | 17,227 | 17,358 | 16,888 | 95% | 11,227 | 11,585 | 2,057 | 2,020 | 4,407 | 428 | 533 | 10,725 | 11,686 | 1.44 |

**Figure 14.** Edge-based analysis and its effectiveness. Reduction in number of instrumented edges is shown in last column.

applications written on top of the ASP.NET framework, consisting of several separate DLLs, as shown in column 2. Not all code contained within the application source tree is actually deployed to the Web server. Most of the time, the number and size of deployed DLLs primarily consisting of .NET bytecode is a good measure of the application size, as shown in column 3. Note that in several cases, libraries supplied in the form of DLLs without the source code constitute the biggest part of an application. Finally, to provide another measure of the application size, column 4 shows the traditional line-of-code metric for *all* the code within the application. Note that correct manual sanitization for these applications is a challenge, as explored in the Merlin project [23]; we therefore believe that fully automatic placement is a better alternative.

**Policy:** There are applications ranging from tens of thousands of lines of code to over a million in the case of the Relationship Management application.

We classified sources and sinks into the three categories: normal, file, resource, based on their functionality (i.e., `TextWriter.Write` is a file-related sink). We used the policy shown in the table in this paragraph for these applications. Finally, we completely disregarded existing sanitizers, fully automating sanitizer placement.

| | normal | file | ⊘ |
|---|---|---|---|
| normal | $S_1$ | $S_2$ | ⊥ |
| resource | $S_3$ | $S_4$ | ⊥ |
| ⊘ | ⊥ | ⊥ | ⊥ |

### 5.1.1 Node-based Placement

Figure 13 contains the results of applying the node-based placement strategy. Applications are represented as graphs, some nodes of which are marked as sources or sinks. Dataflow graphs are computed by the Cat.Net tool [26], and are fairly sparse. Nodes of the graph are parameters and return results of individual methods in the application or its libraries. Edges represent data flow as inferred by Cat.Net. A different static analysis tool could also be used to construct these graphs; the precision and soundness of Cat.Net results is orthogonal to our approach. The number of nodes (column 2) as well as sources and sinks (3–4) ranges from dozens to lower thousands.

Columns 5–8 summarize information about tainted nodes in the graph. Column 5 is forward-tainted nodes (i.e., nodes that can be reached from a source node). Column 6 is backward-tainted nodes (i.e., nodes that can reach a target node). Column 7 is both forward- and backward-tainted nodes (i.e., nodes that are on a path from a source node to a sink node) and column 8 is the fraction of these nodes compared to all nodes in the graph. We can see that for a well-connected graph, the percentage of such nodes can be quite high, going higher than 60%. The implication is that a *very high* fraction of nodes needs to be instrumented to propagate the taint forward at run time.

Columns 9–11 capture our exclusive node computation. Column 9 is the number of exclusive nodes and column 10 is the number of latest exclusive nodes. Column 11 is the fraction of latest exclusive nodes within the nodes of the graph. Finally, column 12 shows the *coverage*, which is the fraction of all source-sink paths that are properly sanitized with latest exclusive nodes.[3] Two key take-aways from this table are as follows:

- the naïve approach of taint-tracking on all nodes on a path from a source to a sink is *very expensive*, with as many as 60+% of nodes needing to be instrumented; and
- while instrumenting just the latest exclusive nodes requires less instrumentation, the obtained coverage is sig-

[3]Coverage numbers are not available for the largest two application; the large number of paths in the dataflow graph cannot be enumerated in reasonable time.

nificantly less than 100%, so the static node-based approach is generally *unacceptable* for sanitizer placement.

#### 5.1.2 Edge-based Placement

Figure 14 shows the results of applying the edge-based placement strategy. Column 2 shows the number of edges in the graph. Columns 3–4 show the number of sources and sinks, respectively; note that we use a slightly different policy for which nodes as marked as sources and sinks compared to Figure 13. Columns 5–7 show the number of forward and backward-tainted edges and edges tainted in both directions. Column 8 shows the percentage of edges tainted in both directions as a fraction of the number of edges in the graph.

Columns 9–10 show the number of source-dependent edges and sink-dependent edges. Columns 11–13 show the number of in-trigger edges, out-trigger edges, and sanitizer edges. Columns 14–16 show the counts for the other kinds of edges computed by the edge-based formulation. Finally, columns 17 and 18 show the number of edges needing instrumentation and the savings compared to the naïve approach of instrumenting edges that are both forward- and backward-tainted. We highlight particularly noticeable savings in bold. (As described in Section 3.3.5, we do not count unneeded sanitization edges when counting edges that require instrumentation.) Three key take-aways from this table are:

- We see that for most applications, the percentage of edges that are forward- and backward-tainted is quite high, indicating that the underlying dataflow analysis of Cat.Net is quite imprecise, leading to a great deal of connectivity within the dataflow graph.
- Savings in terms of the number of instrumentation points in the last column of Figure 14 are 6.19× on average.
- In general, our analysis is not as effective at reducing the number of instrumentation points for densely-connected graphs (the last several rows) as it is for the sparser graphs (the first several rows).

### 5.2 Synthetic Graphs

Finally, we evaluate our (edge-based) algorithm on some synthetically constructed graphs. To build such graphs, we start with 100 sources, 100 sinks, and 1,000 regular nodes. We randomize the type of the sources with equal probability between ⊘, □, ◯, ◇, and △, and the type of sinks between ⊘, ●, ■, and ▲, using the policy in Figure 2 for proper sanitizer placement. We connect sources to sinks by performing a random walk of length 10 starting at a random source and ending at a random sink through the graph, creating edges as we pass from node to node. We use a density parameter $d$ to vary how many such walks we perform, affecting the number of edges.

Figure 15 shows the improvements with edge-based instrumentation compared to naïve, taint-based instrumentation as the number of edges grows. We can see that for sparse graphs, the improvements are most noticeable, peaking at over 27×, gradually becoming less pronounced (only 48% improvement for 550 edges).

The improvements obtained with our strategy depends on the quality of the underlying dataflow graph. Results in Figure 15 suggest that our strategy performs better with *sparser* dataflow graphs, and performs worse when the dataflow graph is more highly connected. Since more precise analyses produce sparser dataflow graphs (since there are fewer conservative over-approximations of dataflow), investment

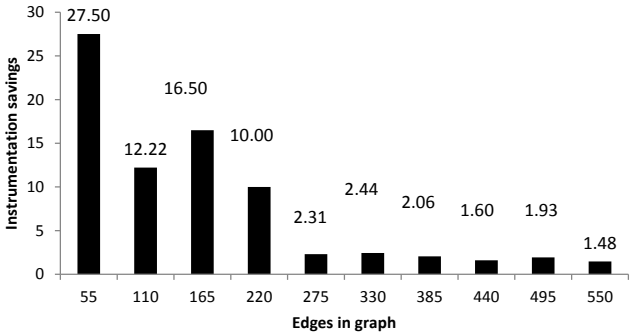


**Figure 15.** Synthetic graph results.

in precise static dataflow graph construction may improve the results of sanitization placement. This conclusion resonates with the experience of a number of projects that use static analysis to reduce the number of run-time instrumentation points: if static analysis is imprecise, the effect of this reduction is not very significant.

## 6. Related Work

The most closely-related work is that of King et al. [19], which considers the problem of resolving type errors in security-typed programs [36] by automatically placing mediation statements in a program (which explicitly declassify information or authorize the possibly dangerous information flow). They construct a graph representation of information flow in a program, such that source nodes are high-security inputs, and sink nodes are low-security outputs. A min-cut in this graph corresponds to a minimal set of program points such that insertion of mediation statements at these program points would allow the program to type check.

Their problem is similar to ours, in that every path from a source to a sink must have a mediation statement. However, their problem is simpler than ours, because all paths require the same kind of mediation statement. By contrast, our policies allow different source-sink pairs to require different sanitization. As such, we are unable to use a min-cut approach to identify sanitization program points. Also, in our setting, it is important to prevent over-sanitization: values should have the appropriate sanitization applied exactly once. By contrast, it is permissible (though perhaps undesirable) for a path from a source to a sink to have multiple mediation statements.

Samuel et al. [38] share our goal of automatic sanitization, but the details are very different. Our approach is designed to work on large legacy applications written in languages such as Java or C#, whereas they focus on much smaller programs in Google Closure. Their technique is based on constraint satisfaction using a custom solver, and has potential for scalability issues, whereas our approach uses dataflow analysis with well-understood scalability properties.

**Graph-based analysis for information security:** Several researchers have used program dependence graphs for analyzing information security of programs [12, 14, 40]. Program dependence graphs include both data dependencies and control dependencies, unlike the dataflow graphs we use in this work, which typically contain just data dependencies. Hammer et al. [13] consider enforcement of declassification [37] using program dependence graphs. However, Hammer et al. require certain nodes in a program dependence to be annotated as declassifiers, whereas we seek to *infer* where to insert declassifiers and sanitizers.

**Software security analysis of web applications:** Program analysis has a long history of being used for finding security bugs in web applications. Static analysis has been advocated for PHP, Java, and other languages [17, 18, 22, 47]. Multiple run-time analysis systems for information flow tracking have also been proposed [11, 25, 29, 31].

**Automating placement:** Most recently, we have seen increased interest in automating security-critical decisions for the developer [38, 46]. The use of a security type system for enforcing correctness is another case of cooperating with the developer to achieve better code quality and correctness guarantees [33].

**Sanitizer correctness:** Balzarotti et al. [3] show that custom sanitizer routines are often incorrectly implemented. Our concerns in this paper are complimentary to sanitizer correctness. The Cross-Site Scripting Cheat Sheet shows over two hundred examples of strings that exercise common corner cases of web sanitizers [34]. The Bek project proposes a systematic domain-specific languages for writing and checking sanitizers [15, 45].

**Specification inference:** Livshits et al. [23] propose an approach to inferring information flow specifications (sources, sanitizers, and sinks) using factor graphs. Kremenek et al. [21] propose belief inference as a way to infer specifications for static analysis checkers. Vaughan and Chong [44] propose policy inference to discover correct declassification policies.

**Graph algorithms: LCM and PRE:** A range of graph-theoretical algorithms from compiler literature is relevant for our work. In particular, Knoop et al. [20] describe lazy code motion. Rüthing et al. describe a variant of it called sparse code motion [35]. Partial redundancy elimination of PRE is described by Hosking et al. [16] and Briggs and Cooper [5].

## 7. Future Work

The design of our placement strategies is motivated by real-world concerns (for example, in the edge-based strategy, we perform sanitization as early as possible to reduce the amount of run-time taint-tracking). However, we do not offer a formal notion of optimality of our algorithms. This is in part because it is unclear what we should aim to optimize. Possible candidates include reducing the number of instrumentation points or run-time overhead for the worst-case or average-case workloads. Exploring these different notions of optimality in order to decide which is best requires building a more complete prototype.

Our matrix-based policy specification allows different sanitizers to be specified for all source-sink pairs. However, this may not be sufficiently expressive in all cases. For example, how do we properly sanitize the result of concatenating string data from two different kinds of sources? We see at least two possible solutions to this lack of expressiveness. One is to treat as sinks computations that combine two or more source types, effectively forcing proper sanitization to take place on values before computation occurs. An alternative that works for some compositional operations (e.g., string concatenation) is to perform finer-grained byte-level tagging, so that the appropriate sanitizer can be applied to the appropriate bytes of the value.

Another shortcoming of our matrix-based policy specification is that we collapse a sequence of sanitizers into a single one. This might prove to be a disadvantage in some settings, where keeping them separate would create interesting optimization opportunities.

We have described our approach as working on dataflow graphs that describe data dependencies. We believe our approach could be extended to work on graphs that also record control dependencies. However, it may be difficult to ensure that sanitizer and declassifiers correctly account for potentially dangerous control dependencies.

We also believe that the approach outlined here can apply to settings other than security. Consider the challenge of manually placing `catch` blocks in a program written in Java or C#. This problem has a similar structure to the sanitizer placement problem: instead of the dataflow graph we have the call graph of the program; for source nodes, we have statements that can throw exceptions; instead of sanitizers we have `catch` blocks. Finally, there is only one sink, the top-most `main` function, by the exit of which we generally need to catch all run-time-catchable exceptions.

## 8. Conclusions

Traditionally, developers have been responsible for properly dealing with the possibility of injection attacks and information leaks in their code, leading to numerous bugs, especially in large, complex code bases. The algorithms presented in this paper pave the way for completely automatic placement of sanitizers and declassifiers.

We proposed two strategies for automatic sanitizer placement. The first is a node-based entirely static approach that has no run-time overhead, but in many settings will not correctly sanitize all values. The second strategy is an edge-based approach that attempts to place sanitizers statically, but uses run-time taint-tracking when necessary to determine the appropriate sanitization to apply to values. The edge-based placement strategy will always sanitize values correctly, and reduces the number of nodes that require instrumentation, sometimes by as much as 27×, compared to naïve taint-tracking of values between sources and sinks.

## Acknowledgments

We thank the anonymous reviewers for their helpful comments. We appreciate the helpful comments of Trent Jaeger and Somesh Jha. This research is supported in part by the National Science Foundation under Grant No. 1054172.

## References

[1] A. V. Aho, M. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2007.

[2] D. Avots, M. Dalton, B. Livshits, and M. S. Lam. Improving software security with a C pointer analysis. In *Proceedings of the International Conference on Software Engineering*, May 2005.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[4] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the International World Wide Web Conference*, 2010.

[5] P. Briggs and K. D. Cooper. Effective partial redundancy elimination. In *Proceedings of the Conference on Programming Language Design and Implementation*, 1994.

[6] B. Chess and J. West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Information Security Technical Reports*, 13, January 2008.

[7] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proceedings of the Workshop on Secure Web Services*, 2009.

[8] S. Chong, K. Vikram, and A. C. Myers. Sif: enforcing confidentiality and integrity in Web applications. In *Proceedings of Usenix Security Symposium*, 2007.

[9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting privacy leaks in iOS applications. In *Proceedings of the Annual Network and Distributed System Security Symposium*, Feb. 2011.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the Usenix Conference on Operating Systems Design and Implementation*, 2010.

[11] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the Annual Computer Security Applications Conference*, Dec. 2005.

[12] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.

[13] C. Hammer, J. Krinke, and F. Nodes. Intransitive noninterference in dependence graphs. In *2nd International Symposium on Leveraging Application of Formal Methods, Verification and Validation*, Nov. 2006.

[14] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, Mar. 2006.

[15] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the Usenix Security Symposium*, Aug. 2011.

[16] A. L. Hosking, N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan. Partial redundancy elimination for access path expressions. *Software Practice and Experience*, 31, May 2001.

[17] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web*, 2004.

[18] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[19] D. King, S. Jha, D. Muthukumaran, T. Jaeger, S. Jha, and S. A. Seshia. Automating security mediation placement. In *Proceedings of the European Symposium on Programming*, 2010.

[20] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. *SIGPLAN Notes*, 39:460–472, April 2004.

[21] T. Kremenek, P. Twohey, G. Back, A. Y. Ng, and D. R. Engler. From uncertainty to belief: Inferring the specification within. In *Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[22] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, 2005.

[23] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[24] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, pages 365–383, 2005.

[25] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: runtime vulnerability protection for Web applications. Technical report, Stanford University, 2006.

[26] Microsoft Corporation. Microsoft Code Analysis Tool .NET (CAT.NET). `http://www.microsoft.com/en-us/download/details.aspx?id=19968`, 3 2009.

[27] Microsoft Corporation. Microsoft web protection library. `http://wpl.codeplex.com/`, 2012.

[28] N. Mitchell, G. Sevitsky, and H. Srinivasan. The diary of a datum: an approach to modeling runtime complexity in framework-based applications. In *Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, 2005.

[30] OWASP. OWASP-Java-HTML-sanitizer. `http://code.google.com/p/owasp-java-html-sanitizer/`, 2011.

[31] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, Sept. 2005.

[32] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the Usenix Security Symposium*, 2009.

[33] W. Robertson and G. Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[34] RSnake. XSS cheat sheet for filter evasion. `http://ha.ckers.org/xss.html`.

[35] O. Rüthing, J. Knoop, and B. Steffen. Sparse code motion. In *Proceedings of the Symposium on Principles of Programming Languages*, 2000.

[36] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[37] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*, pages 255–269. IEEE Computer Society, June 2005.

[38] M. Samuel, P. Saxena, and D. Song. Context-sensitive autosanitization in web templating languages using type qualifiers. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2011.

[39] P. Saxena, D. Molnar, and B. Livshits. ScriptGard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2011.

[40] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. Technical Report 2008-171, Sun Microsystems Labs, 2008.

[41] V. Srivastava, M. D. Bond, K. S. McKinley, and V. Shmatikov. A security policy oracle: detecting security holes using multiple API implementations. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2011.

[42] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *Proceedings of the Symposium on Principles of Programming Languages*, 2006.

[43] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2009.

[44] J. Vaughan and S. Chong. Inference of expressive declassification policies. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2011.

[45] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the Sympolisium on Principles of Programming Languages*, Jan. 2012.

[46] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *Proceedings of the European Symposium on Research in Computer Security*, Sept. 2011.

[47] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[48] E. Z. Yang. HTML purifier. `http://code.google.com/p/owasp-java-html-sanitizer/`, 2011.