Idea: Optimized Automatic Sanitizer Placement

Gebrehiwet Biyane Welearegai ([orcid]) and Christian Hammer ([orcid])

University of Potsdam, Potsdam, Germany
welearegai@uni-potsdam.de,hammer@cs.uni-potsdam.de

Abstract. Sanitization is a primary defense mechanism against injection attacks, such as cross-site scripting (XSS) and SQL injection. Most existing research on sanitization focuses on vulnerability detection and sanitization correctness, leaving the burden of sanitizer placement with the developers. However, manual sanitizer placement is complex in realistic applications. Moreover, the automatic placement strategies presented in the literature do not optimize the number of sanitizer positions, which results in *inconsistent multiple-sanitization* errors and duplicated code in our experience.
As a remedy this paper presents an optimized automatic sanitizer placement to reduce the number of positions where sanitization is required. To that end, we analyze the dataflow of a program via static analysis. We optimize the number of sanitizer positions by preferring nodes common to multiple paths as sanitizer positions. Our evaluation displays equal sanitization coverage as previous approaches with a reduced number of sanitizers, and reduces the number of sanitization errors to 0.

Keywords: Vulnerability, Sanitization, Automatic sanitizer placement

1 Introduction

Web applications are often vulnerable to script injection attacks, such as *cross-site scripting (XSS)* and *SQL injection*. XSS vulnerabilities allow an attacker to inject client-side scripting code into the output of an application which is then sent to another user's web browser [1]. The scripting code can be crafted to send sensitive data, such as user login credentials, credit card numbers, to a third party when executed in the browser. Likewise, SQL injection vulnerabilities allow an attacker to execute malicious SQL statements that violate the application's desired data integrity or confidentiality policies [2]. The predominant first-line approach to prevent such attacks is *sanitization*, the practice of encoding or filtering untrusted inputs of an application. Existing research on sanitization mainly deals with vulnerability detection [3,4]. These approaches assume that an application is secure if sanitization is applied on all paths from sources to sinks. However, this does not always hold as the sanitization process itself might be buggy or incomplete. This triggered a line of research on sanitizer correctness [4,5]. Yet, having correct sanitizers is not enough to completely mitigate scripting attacks; the sanitizers must also be placed correctly.

```
1  main() {
2    exchange = getSource();
3    func1(exchange);
4    func2(exchange);
5  }
6  func1(exchange) {
7    exchange.getIN();
8  }
9  func2(exchange) {
10   exchange.getIN();
11 }
```

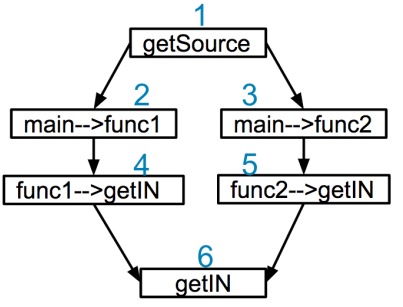


Fig. 1. Path decomposition due to flow-insensitivity (code and its data flow graph)

Sanitizer placement depends on how input data is being used [6], where it originates from [7] and the context in which it is rendered [6,8]. Hence, manual sanitizer placement is difficult and error prone [4,5]. On top of missing sanitization, common errors are *inconsistent sanitization* (mismatch of sanitizer and context) and *inconsistent multiple sanitization* (sanitizers may not be idempotent) [9]. This motivated research on automatic sanitizer placement [8,6,9].

Automatic sanitizer placement techniques assume that the individual sanitizers are correct, and that the mapping of sanitizers to their contexts [8,9] or source and sink policy [6] is given. The goal of the automatic placement strategy is to place sanitizers on every program path between an untrusted input and a possibly scriptable output of an application while preventing errors due to inconsistent sanitization and inconsistent multiple sanitization. Currently, the automatic placement techniques strive for this goal but do not try to optimize the number of sanitizer calls in the code. In the worst case each path in the code may have its own call to an appropriate sanitizer. However, due to the flow-insensitive points-to-analysis characteristics, a single runtime path can be decomposed into several paths that may lead to inconsistent multiple sanitization. Assuming the policy of Figure 1 does not allow to put a sanitizer at node n_6, nodes n_4 and n_5 are selected as sanitizer positions according to existing research. However, at runtime these two nodes are being executed subsequently, i.e. the value in exchange is being sanitized twice. Our experiments (cf. Section 4) display an error rate of 20% for the non-optimized algorithm, i.e., 20% of all sanitizers might suffer from erroneous inconsistent multiple sanitization. An alternative approach that does not suffer from the shortcomings of flow-insensitivity analysis would be to dynamically taint untrusted input data at runtime and to propagate these taints during computation, but this may incur significant runtime overheads.

In this paper, we propose an optimized automatic sanitizer placement technique by statically analyzing the flow of tainted data. This analysis takes a dataflow graph and the sanitization policy, that specifies the mapping of sanitizers with respect to source and sink combinations, as input. The goal is, then, to automatically find sanitizer positions satisfying sanitization correctness, i.e., every value that flows from a source to a sink must have the given type of sanitizer

applied exactly once. Our placement technique uses *static node-based* analysis, and nodes that are common to several paths requiring the same sanitizer type are proposed as the best candidate for optimized sanitizer positions. For the data flow graph in Figure 1, our optimized approach selects node n_1 as sanitization position, thus eliminating multiple sanitization and code duplication. Our evaluation (in Section 4) shows that the reduction of sanitizers due to this optimization reduces or eliminates the sanitization errors of previous approaches.

2 Sanitizer Placement Overview

This section presents background on dataflow graphs and sanitization policies which are the input to the sanitization placement problem, followed by a clarification of the placement problem.

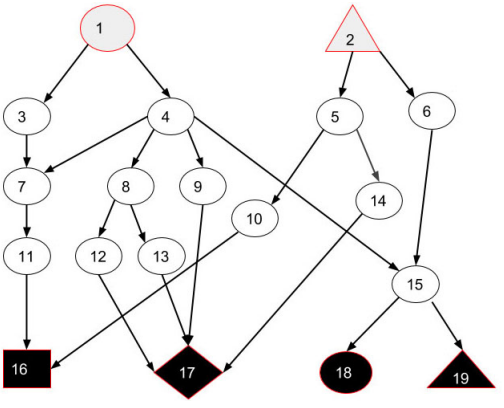
2.1 Dataflow Graph and Sanitization Policy

A dataflow graph is a static representation of a program where nodes represent statements/predicates and edges indicate data dependencies. When the program performs a computation, values may only flow according to the data dependence edges. To guarantee a secure flow of data, we require a sanitization policy that defines an appropriate sanitizer for values that flow from a given source to a given sink. The policy is usually given in the form of a table where sources and sinks are displayed in rows and columns and sanitizers are given as the entries [6]. Throughout this paper, we use the dataflow graph and its sanitization policy in Figure 2 as an example. Source types ($\bigcirc, \triangle$) are shown in the rows and sink types ($\blacksquare, \blacklozenge, \bullet, \blacktriangle$) are shown in the columns. $\oslash$ represents data sources and sinks that are irrelevant for security. We use metavariables P, I and O to range over policies, sources and sinks, respectively. Particularly, we write $P(I, O)$ for the entry in policy P for the source I and sink O. For instance, for the policy P in Figure 2, data originating from source type $\bigcirc$ going to sink type $\blacksquare$ should have sanitizer S_1 applied to it. If data that flow from source type I to sink type O does not require sanitization, we represent its policy as $P(I, O) = \bot$.

2.2 Sanitizer Placement Problem

The challenge of sanitizer placement is to identify an appropriate sanitizer and its position in the source code in order to prevent untrusted data flowing from a source to a sink with dangerous scripts to be injected in the sink context. This problem arises as the result of the source, sink and context-sensitivity property of sanitization placement [6,8,7]. In large applications, with a multitude of nodes and paths, manual sanitizer placement is hard to get right. Consequently, an automatic decision procedure is required whose goal is to ensure correct sanitization of data in an application given its dataflow graph and sanitization policy, i.e., according to the correctness definition [6] (Definition 1).

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



	■	♦	●	▲	⊘
◯	S_1	S_1	S_4	S_3	$\perp$
△	S_2	S_2	$\perp$	S_3	$\perp$
⊘	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

Fig. 2. Dataflow graph example (sources on top, sinks at bottom) and sanitization policy

- if $P(s,t) = S$, then every value that flows from source node s to sink node t has sanitizer S applied exactly once, and no other sanitizer is applied.
- if $P(s,t) = \perp$, then every value that flows from source node s to sink node t has no sanitizer applied.

The reason why sanitizers must be applied only once on a single path is that they are not necessarily idempotent [3]. Specifically, applying a sanitizer several times on a path may lead to inconsistent multiple sanitization errors [9]. However, untrusted input may be coming from a source in multiple contexts, which implies different sanitizers might be required within a single path leading to a sink. For this type of situation, multiple (potentially nested) sanitizers are modeled as a single (composite) sanitizer [6].

3 Our Approach

To solve the sanitizer placement problem, we propose two types of static node-based strategies: a *less-optimized* and a *fully-optimized* solution. The less-optimized solution optimizes the number of sanitizer positions only when there is a common node for all paths from a source to a sink. Other than that, it follows the same approach as Livshits et al [6]. The fully-optimized solution, however, always reduces the number of sanitizer positions by selecting nodes common to many paths requiring the same sanitizer as a sanitization positions. This optimization removes code duplication and the occurrence of inconsistent multiple-sanitization error that could appear as the result of flow-insensitive static analysis. Due to space limitation, we present detailed explanations only for the fully-optimized solution. However, the result of both approaches is evaluated in Section 4.

Table 1. S_i-possible and S_i-exclusive nodes for Figure 2

Sanitizers	Possible	Exclusive
S_1	1, 3, 4, 7, 8, 9, 11, 12, 13, 16, 17	3, 7, 8, 9, 11, 12, 13
S_2	2, 5, 10, 14, 16, 17	5, 10, 14
S_3	1, 2, 4, 6, 15, 19	19
S_4	1, 4, 15, 18	$\emptyset$
$\perp$	2, 6, 15, 18	$\emptyset$

3.1 Fully Optimized Approach

For i, j ranging from 1 to k where k is the number of sanitizer types, a node n is called *S_i-possible* if it is found on a path from source s to sink t that requires sanitizer S_i, i.e., $P(s,t) = S_i$. This implies that at least some of the data traversing from source s to sink t through node n needs to be sanitized with sanitizer S_i. Likewise, a node n is called *S_i-exclusive* if it is S_i-possible and not S_j-possible for all $i \neq j$ [6].

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

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 $P(s,t) = S_i$.

For the dataflow graph example (Figure 2), the possible and exclusive nodes for sanitizers S_1, S_2, S_3, S_4 and $\perp$ are given in Table 1. According to the correctness definition (Definition 1), sanitizer nodes must be selected from the S_i-exclusive sets. However, sanitizer S_4 has no exclusive nodes, hence values flowing from source type ◯ to sink type ● cannot be sanitized statically. This is due to the inability of static analysis to identify the source and/or destination of input data at nodes that are not exclusive to a sanitizer type. Additionally the correctness definition prohibits placing the same sanitizer more than once on a single path. Hence, our solution selects the node that provides most optimal placement if there is more than one exclusive node on a single path.

In addition to the S_i-possible and S_i-exclusive sets, we need the following definitions to elaborate our approach.

Definition 4. The *S_i-node-frequency* for a node $n \in N$ is the number of paths through n that satisfy $P(s,t) = S_i$, for a source node s and s sink node t,

Definition 5. Node $n \in N$ is *S_i-exclusive-p_j-previous* if it is S_i-exclusive and it is found on any of the paths traversed before path p_j during the depth first path search of path analysis.

Definition 6. Node $n \in N$ on a path p_j is *S_i-exclusive-p_j-exclusive* if it is S_i-exclusive and not S_i-exclusive-p_j-previous.

Definition 7. Node $n \in N$ is called the *S_i-position* if it is selected to be sanitization node for sanitizer S_i.

Definition 8. *S_i-backtracking-map* collects the mapping of S_i-position nodes, at every path iteration, to paths requiring sanitizer S_i.

For each path p_j our fully-optimized placement algorithm traverses the S_i-exclusive-p_j-exclusive nodes and selects the last node on that path with max-

imum S_i-node-frequency as the temporary[1] S_i-position. However, more than one sanitizer could be included on a single path with this strategy alone, which violates the correctness definition (Definition 1). To resolve this problem, we consider three conditions based on the number of previously identified sanitizer position nodes S_i on the current path p_j.

First, if there is no S_i-position on p_j, we apply the strategy described above and add p_j into the S_i-backtracking-map.

Second, there is exactly one S_i-position on p_j. Thus the current path p_j already has a S_i-position node, hence we do nothing. However, that S_i-position node might be removed later due to backtracking in which case p_j would lack a sanitizer. For this reason, p_j is entered into a map of skipped paths which will be re-added to the iteration if its corresponding S_i-position node is removed.

Third, when there is more than one S_i-position node on p_j, we backtrack to find the paths that provided these nodes and assign a different sanitization node for all except one path. To select that one, we consider the intersection of the current path with all paths that have provided the S_i-position nodes causing multiple sanitization errors. Then, taking two intersection sets at a time starting from the sink, we compare the number of incoming edges to the top node in the bottom intersection set with the number of outgoing edges from the bottom node in the top intersection set. After that, the path whose S_i-position node in the intersection set had the lower number is removed and another intersection set is compared with the one that had the higher number. When all intersection sets are considered, only the path that results in the maximum number remains and serves as the source of S_i-position for the current path, as well. The reason behind this is that if the S_i-position node is removed with the path, we have to apply S_i at all its previous/next nodes depending on its relative position in the two intersection sets. Thus, it is preferable to remove the ones that have lower in-/out-degrees. Lastly, these removed paths as well as the skipped paths that contain these removed S_i-positions are again included to the path iteration to select another S_i-position node.

To exemplify our fully-optimized solution, we use the S_i-*exclusive* graph shown in Figure 3. The path numbers in the graph indicate the iteration order during the analysis to find a single S_i-position node on every path. For p_1 n_3 is selected as sanitization node according to the first condition since it has node frequency *3*. Path p_2 already contains n_3 (second condition). Next, node n_8 is selected as S_i-position for p_3 (first condition).

The fourth iteration results in backtracking, as p_4 contains two sanitization nodes: n_3 and n_8. From the intersection of p_4 with p_3, we get the top node n_5 and from the intersection of p_4 with p_1 we get the bottom node n_3. Comparing the out-degree of n_3 (3 edges) with in-degree of n_5 (2 edges), n_3 is chosen to remain sanitizer node. Finally n_4 is selected as new sanitization node for path p_3 (subpath excluding n_5 and n_8) using condition one. Placing S_i at nodes n_3 and n_4 is sufficient to sanitize all values that flow along these paths.

[1] The S_i-position for path p_j can be changed later due to backtracking.

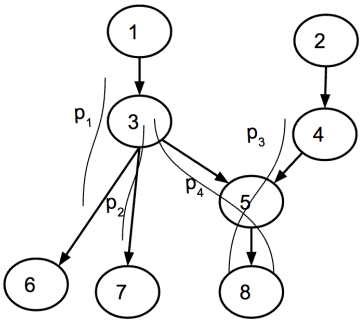


Fig. 3. S_i-exclusive graph example for fully-optimized solution explanation

Table 2. Results of placement approaches

	Fully-optimized				Less-optimized			
	Sanitization			Analysis	Sanitization			Analysis
N	**coverage**	**errors**	**numbers**	**time (m)**	**coverage**	**errors**	**numbers**	**time (m)**
1	50 %	0%	5	2:31	50 %	20%	8	2:40
2	90 %	0%	9	2:09	90 %	40 %	14	2:01
3	80 %	0%	8	2:44	90 %	40 %	14	2:42
4	80 %	0%	9	4:46	90 %	40 %	14	5:19
5	90 %	0%	9	9:30	90 %	40 %	14	9:02

In the end, the S_i-position nodes using our fully-optimized solution for the dataflow graph and the policy in Figure 2 are: S_1 at nodes n_{11}, n_8, n_9, S_2 at node n_5 and S_3 at node n_{19}. S_4 and $\perp$ have no exclusive node, hence sanitization is not statically possible. Using our less-optimized solution, n_{12} and n_{13} would be selected for S_1 instead of n_8, similar to previous approaches [6].

4 Evaluation

We evaluate our approaches using WALA [10] on an application whose call graph consists of 15,214 nodes, 119,026 edges, 14,070 methods, and 724,806 bytes. Sanitization coverage, sanitization errors, the number of sanitizer positions and time taken by the analyses are used as evaluation parameters. The sanitization coverage indicates the ratio of untrusted inputs passed through the correct sanitizers. The sanitization errors are multiple sanitization errors due to WALA's flow-insensitive pointer analysis that decomposes one path to multiple paths. Table 2 reports the result of the fully-optimized and less-optimized approaches on a MacBook Pro with a 2.9 GHz Intel Core i7 processor and 16 GB RAM. N in the first column indicates a call-string based context-sensitivity parameter. The ratio of sanitization errors is relative to the required correct sanitizer positions.

Maximum sanitization coverage was achieved for $N = 2$ for both approaches, and we took that for comparison as missing sanitization may have devastating

effects. The missing 10% are caused by paths lacking sanitizer-exclusive nodes. However, for these paths runtime tracking of input data can be applied instead of static sanitizer placement [6]. For example, in Figure 2 one might want to add code that registers whether data was flowing to node 15 from node 4 or from node 6. Then in node 18 one would be able to decide whether sanitizer S_4 needs to be applied (flow from 4) or not (flow from 6).

As expected, the number of sanitizer positions is reduced for the fully-optimized approach since it always attempts to find common nodes for paths that require the same sanitizer type. Unlike the less-optimized approach, which is is only slightly more precise than previous work [6], the fully-optimized approach does not result in sanitization errors. The analysis time is almost identical for each context-sensitivity value N. But the less-optimized solution can provide its maximum coverage at a lower value of N, hence analysis time could be lower.

Note that the less-optimized solution follows exactly the same approach as *Livshits et al* [6] in its non-optimized variant. Hence, the evaluation of the fully-optimized solution with respect to the less-optimized solution also elaborates on how our fully-optimized solution resolves the multiple sanitization error and code duplication problems of existing research. The correctness (sanitization coverage) of both our solutions is in sync with *Livshits et al* [6], since we use the same node-based static analysis approach, i.e., every piece of data which flows through a path that has at least one exclusive node is sanitized. This has been confirmed using several test cases although we use only one application for a final evaluation. The optimality of the fully-optimized approach is also confirmed in a similar way, i.e., it finds the least sufficient number of sanitizer positions.

To the best of our knowledge, previous research does not consider optimization of sanitizer placement and this paper is the first to report that such an optimization has benefits beyond code clone elimination, namely a reduction in the sanitization error rate.

5 Related Work

The automatic sanitizer placement of Livshits et al. [6] is closely related to our approach. They propose two solutions: a static node-based solution, and an edge-based solution based on static analysis and runtime taint tracking. The former is similar to ours but does not attempt to optimize the number of sanitizer positions. Another research that shares our goal is presented by Samuel et al. [8]. They provide static type inference-based approach to automatically apply sanitizers. It is unclear whether type inference based analysis is scalable as they focus on a small program (less than 4000 lines of code) in Google Closure. Saxena et al. [9] propose *SCRIPTGARD* that can detect and repair incorrect placement of sanitizers in ASP.NET applications. *SCRIPTGARD* assumes that developers have manually applied context-sensitive sanitization correctly. A dynamic analysis detects and auto-corrects context-inconsistency errors in sanitization. In contrast, our approach can correctly apply sanitizers in an application com-

pletely lacking developer-supplied annotations. Additionally, we leverage static analysis in order to eliminate runtime overhead of dynamic analysis.

6 Conclusion

This paper presents an automatic sanitizer placement mechanism that optimizes the number of sanitizer positions. Due to the reduced number of sanitizer positions we did not experience any inconsistent multiple sanitization errors. Hence, the fully-optimized algorithm is a valuable solution for real world applications. However, in some cases runtime information is required to identify the valid sanitizer type. Thus, a hybrid analysis that leverages runtime tracking in such situations is proposed to guarantee full sanitization coverage.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project SimoBA (16KIS0440).

References

1. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proc. of Network and Distributed System Security, p.12 (2007)
2. Halfond, W., Viegas, J., Orso, A.: A Classification of SQL-Injection Attacks and Countermeasures. In: IEEE International Symposium on Secure Software Engineering (ISSSE), pp. 13–15. IEEE(2006)
3. Xie, Y., Aiken, A.: Static Detection of Security Vulnerabilities in Scripting Languages. In: 15th USENIX Security Symposium, pp. 179-192 (2006)
4. Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, and G.: Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: Symp. on Security and Privacy, pp. 387-401. (2008)
5. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with BEK. In: 20th USENIX conference on Security, pp. 1-1. USENIX Association (2011)
6. Livshits, B., Chong, S.: Towards fully automatic placement of security sanitizers and declassiefiers. In: ACM SIGPLAN Notices, pp. 385–398, ACM (2013)
7. Weinberger, J., Saxena, P., Akhawe, D., Finifter, M., Shin, R., Song, D.: A systematic analysis of XSS sanitization in web application frameworks. In: European Symposium on Research in Computer Security, pp. 150-171. Springe (2011)
8. Samuel, M., Saxena, P., Song, D.: Context-sensitive auto-sanitization in web templating languages using type qualifiers. In CCS, pp. 587-600. ACM (2011)
9. Saxena, P., Molnar, D., Livshits, B.: SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In CCS, pp. 601–614. (2011)
10. T. J. Watson Libraries for Analysis (WALA), `http://wala.sourceforge.net`