Scalable String Analysis: An Experience Report

Kostyantyn Vorobyov
Yang Zhao
Padmanabhan Krishnan
Oracle Labs
Brisbane, Queensland, Australia
(kostyantyn.x.vorobyov,yang.yz.zhao,paddy.krishnan)@oracle.com

Abstract

In this paper we present OLSA – a tool for scalable static string analysis of Java programs. OLSA is based on intra-procedural string value flow graphs connected via call-graph edges. Formally, this uses a context-sensitive grammar to generate the set of possible strings. The analysis is focused on scalability and is thus not sound. This trade off is acceptable in the context of bug-finding in large web applications.

We evaluate our approach by using OLSA to detect SQL injections and unsafe use of reflection in DaCapo benchmarks and a large internal Java codebase and compare the performance of OLSA with JSA, one of state-of-the-art string analysers. The results indicate that OLSA can analyse industrial-scale codebases in a matter of hours, whereas JSA does not scale to many DaCapo programs. The set of potential strings generated by our string analysis can be used for checking the validity of the reported potential vulnerabilities.

CCS Concepts: **• Security and privacy → Software security engineering**; **• Software and its engineering → Automated static analysis**; *Software maintenance tools.*

Keywords: static analysis, local reasoning, context-sensitive grammar

ACM Reference Format:
Kostyantyn Vorobyov, Yang Zhao, and Padmanabhan Krishnan. 2021. Scalable String Analysis: An Experience Report. In *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3460946.3464321

1 Motivation

Static string analysis underpins many security-related analyses including detection of SQL injections, cross-site scripting and improper sanitisation. Even though string analysis has received much attention [3, 5, 11], its application to large commercial codebases is still unclear. One of the reasons for such results is that calculating possible values for a string variable is not straightforward. For instance, Costantini et al. [8] observe that the state-of-the-art in this field is still limited; one can use finite automata to model string values, but if the automata are precise they do not scale up.

A popular string analysis tool for Java™[1] programs that makes use of finite automata is the Java String Analyser (JSA) [5]. JSA aims to be sound, has been used in a number of projects and shown to produce useful results [6, 9, 10, 13, 16]. However, as shown by experiments with Violist [11], a string analysis framework for Java and Android, JSA does not scale to large codebases. The same experimentation shows that Violist outperforms JSA, however this evaluation is based on a handful of small Java and Android applications. OSA [4] is another string analysis tool for Java that uses abstract interpretation and allows context-sensitive handling of field variables (a feature not supported by JSA). Experiments with OSA indicated that due to the added context-sensitivity OSA was more precise then JSA but did not outperform JSA in terms of runtime.

In this paper, we tackle the problem of fast and practical string analysis for large Java applications. Specifically, given a callsite c that invokes function f, the goal of the analysis is to compute the set of string values f can produce at c. Even though this is similar to JSA, where a value flow graph (VFG) def-use edges is used to identify the string values that can reach a particular location, we show that our analysis scales to Java codebases that have millions of lines of code.

We report our experience in developing a scalable string analysis for Java in the context of the Parfait static analyser [7]. There are two main reasons for our string analysis. The first is to identify potential issues that may be missed by Parfait. Parfait's main focus is on precision (at the cost of accuracy). Therefore, we sacrifice soundness for scalability. This is in keeping with practice in industry [15]. The focus on

[1]Java ™is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

precision ensures that most of Parfait's reports are actionable. Therefore, by default, Parfait does not enable string analysis. Thus, we want to identify reports that Parfait could have potentially missed. The second is to enhance Parfait's taint analysis with specific information. That is, rather than just report that a string is potentially tainted, we aim to generate string-related expressions that can be used for debugging or error reporting.

1.1 High Level Approach

IFDS and CFL-reachability [14] are efficient techniques for context-sensitive data flow analysis. Inspired by these ideas, we propose a string analysis that presents a context-sensitive solution in the form of a formal context-sensitive grammar. Note that most of the rules in this grammar are context-free. The number of context-sensitive rules is linear in the number of callsites. In other words, we extend CFGs with special symbols to encode the context-sensitivity required for the analysis.

If a function f is involved in generating the string values for a variable, our string analysis generates grammar G_f^b for function f that takes into account only the body of f. An extended grammar G_f is then obtained by adding extra rules to G_f^b. G_f generates strings that capture the behaviour of other functions (including recursive behaviour). Thus, for each string in this extended grammar, we need to apply an algorithm (say $\mathcal{A}$) that interprets these special symbols to generate the actual strings of interest. That is, the set of possible strings at the callsite will be $\{\mathcal{A}(w) \mid w \in \mathcal{L}(G_f)\}$, where $\mathcal{L}(G_f)$ denotes the language of grammar G_f.

This two-step process also allows us to instantiate $\mathcal{A}$ with different algorithms to generate client-specific strings. For instance, while detecting SQL injections a SQL parser that discards invalid strings can be used. If the requirements of the client analysis are not known, all possible strings (subject to resource requirements) can be generated.

In the following section we describe our analysis algorithm in greater detail and in Section 3 we present our experimental results of applying our approach to different systems.

2 Scalable String Analysis

Most existing string analysers focus on precision and require expensive computations. A spot check of our codebases indicated that such heavy-weight analysis was often not needed. For instance, SQL queries in Java are often encoded as constants concatenated with input values. Such expressions can be computed using simpler and more scalable techniques. In the remainder of this section we describe Oracle Labs String Analyser (OLSA) – our scalable string analyser.

OLSA first creates a JSA-style string value-flow graph (SVFG) for each method using only an *intra-procedural* analysis. That is, starting from a point of interest (i.e., a hotspot), OLSA uses backwards def-use analysis to build a separate SVFG for each method in the backward sequence. This process is extended with context-sensitivity to achieve better precision. The inter-procedural analysis (viz., the θ's in the grammar) is handled by special *switch* nodes described below. Once SVFG is constructed, OLSA utilises backwards graph traversal to compute string values per hotspot.

We first present an example to illustrate our ideas. Formal description of OLSA analysis is presented further in Algorithm 1.

2.1 Example

```
 1  String foo(String s1, String s2) {
 2    if (...) {
 3      String t1 = s1 + s2;
 4       return t1;
 5    } else if (...) {
 6      String t2 = s2 + "1";
 7      l1: String u1 = foo(s1, t2);
 8      return u1;
 9    } else {
10      String t3 = "0" + s1;
11      l2: String u2 = foo(t3, s2);
12      return u2;
13    }
14  }
15  String result = foo("a", "b");
```

Listing 1. Example program

The intra-procedural CFG as defined for JSA corresponding to the example in Figure 1 is developed below. The uppercase letter non-terminals correspond to program variables (e.g., $T1$ corresponds to `t1` defined on line 3) and RET_f denotes the return value of the function f.

$$T1 \longrightarrow concat(S1, S2) \tag{1}$$

$$RET_{foo} \longrightarrow T1 \mid U1 \mid U2 \tag{2}$$

$$T2 \longrightarrow concat(S2, \texttt{"1"}) \tag{3}$$

$$T3 \longrightarrow concat(\texttt{"0"}, S1) \tag{4}$$

Because OLSA does not address inter-procedural propagation of strings, no direct productions for $U1$ and $U2$ are defined. We extend the grammar to capture the different invocations of `foo` for $U1$ and $U2$ in the following way:

$$U1 \longrightarrow \theta_{l1}\ RET_{foo} \tag{5}$$

$$U2 \longrightarrow \theta_{l2}\ RET_{foo} \tag{6}$$

In general, the non-terminal symbol θ_l corresponds to the callsite at label l. The above two rules assign the return value of the function `foo` to $U1$ and $U2$ after processing of the callsites at labels $l1$ and $l2$ in Figure 1. In other words, θ represents the context and RET_{foo} captures the propagation of string values. We explain this processing by θ in greater detail below.

The analysis further requires to capture the flow of actual parameters to formal parameters of the function. Rules that capture this flow are shown below, where the non-terminals on the left-hand side correspond to the formal parameters and the non-terminals on the right-hand side correspond to the actual parameters.

$$\theta_{l1}\ S1 \rightarrow S1 \quad \theta_{l1}\ S2 \rightarrow T2 \tag{7}$$

$$\theta_{l2}\ S1 \rightarrow T3 \quad \theta_{l2}\ S2 \rightarrow S2 \tag{8}$$

Finally, we need another rule to summarise the instantiation at callsites. The following template rule captures the valid instantiation of a summary in a context given by θ, where op denotes any string operation with arity n.

$$\theta_l\ op(X_1, \ldots, X_n) \rightarrow op(\theta_l\ X_1, \ldots, \theta_l\ X_n) \tag{9}$$

Note that the generated grammar is not context-free since rules from (7) to (9) are context-sensitive because the non-terminals θ_{l1} and θ_{l2} appear on the left-hand side of the productions.

Given the above rules, we now show we can derive a string for the callsite at label $l1$, assuming that the called function returns at Line 4.

$$\begin{array}{lll} RET_{foo} & \rightarrow U2 & \text{(from 2)} \\ & \rightarrow \theta_{l2}\ RET_{foo} & \text{(from 6)} \\ & \rightarrow \theta_{l2}\ T1 & \text{(from 2)} \\ & \rightarrow \theta_{l2}\ concat(S1, S2) & \text{(from 1)} \\ & \rightarrow concat(\theta_{l2}\ S1, \theta_{l2}\ S2) & \text{(from 9)} \\ & \rightarrow concat(T3, S2) & \text{(from 8)} \\ & \rightarrow concat(concat(\text{"0"}, S1), S2) & \text{(from 4)} \end{array}$$

Once the derivation is complete, the semantics of the string operations may be applied to generate the actual set of strings. For the variable `result` in Figure 1, JSA generates the string <XXXX> (i.e., it cannot resolve it). A brief investigation suggests that the presence of recursion results in JSA's alias analysis and inter-procedural analysis of string values being imprecise. OLSA generates strings "0ab", "0ab1" and "ab". OLSA does not generate "00ab11" because of the unfolding limits (loops are unfolded only once) in our implementation that are imposed for scalability. Note that we have replaced the inter-procedural alias analysis with a context-sensitive grammar rules (e.g., rules 7) unfolded on-demand. This results in a scalable but unsound analysis.

2.2 OLSA Algorithm

Algorithm 1 shows our technique to construct a grammar G for a given program P. The grammar G has non-terminals corresponding to program variables, say X for `x`. For the inter-procedural analysis we use the term $formal(l, x)$ to denote the formal parameter that corresponds to the actual parameter x at callsite l. The language $\mathcal{L}(X)$ of a non-terminal X is the same as the set of strings that the program variable x holds in a sound path-insensitive, flow and fully callsite-sensitive analysis. The algorithm generates the relevant grammar rules corresponding to each program statement. As each statement is handled independently, the algorithm is compositional by design.

```
input  : A program $P$
output : A context-sensitive grammar $G$
1  The grammar $G$ is defined as $(\Sigma, \mathcal{N}, \mathcal{R})$.
2  $\Sigma$ is the set of terminals containing the string
     constants in $P$, names of string operations and
     symbols '(', ')'and ','.
3  $\mathcal{N}$ is the set of non-terminals corresponding to
     program variables (denoted by uppercase letters),
     $RET_f$ corresponds to the return variable for
     function $f$ in $P$, $\overline{x}$ indicates that $x$ represents a
     sequence of variables and is used to capture
     polyadic operations, and $\theta_l$ for every callsite
     location $l$ in $P$.
4  $\mathcal{R} := \emptyset$
5  for every statement $s$ in $P$ do
6      if $s$ is $x := op(\overline{y})$ then
7          $\mathcal{R} := \mathcal{R} \cup \{X \rightarrow op(\overline{y})\}$
8      else if $s$ is $l : x := func(\overline{y})$ then
9          for every $X_i$, $op$ in $func$ add the following to
             $\mathcal{R}$ do
10             $\mathcal{R} := \mathcal{R} \cup \{X \rightarrow \theta_l\ RET_{func}\}$
               $\forall z \in \overline{y} : \mathcal{R} := \mathcal{R} \cup \{\theta_l\ formal(l, z) \rightarrow Z\}$
               $\mathcal{R} := \mathcal{R} \cup \{\theta_l\ op(X_1, \ldots, X_n) \rightarrow$
                                  $op(\theta_l\ X_1, \ldots, \theta_l\ X_n)\}$
11         end
12 end
```

Algorithm 1: Grammar construction.

In Algorithm 1 Line 7 handles string operations such as `concat` or `replace`. As we do not process such operations in the grammar generation phase, we note that X can derive only the result obtained from the variables used in the operation. The definitions starting at Line 9 in Algorithm 1 handle function calls and thus adds the necessary symbols to handle context-sensitivity. These rules are similar to any context-sensitive program analysis. That is, the variable X can derive the result of the function call at location l that is captured by θ_l. Similarly, the transfer of values from the actual parameters and the values in the local variables in the function are analysed context-sensitively using θ_l.

Theoretically, the generated grammar can be used to derive a string for any callsite in the program P. Due to its nondeterministic nature, we additionally construct a graph to represent the grammar and use a customised graph traversal to generate strings at a particular callsite. Figure 1 shows the internal representation of the grammar generated for program in Figure 1, where nodes represent non-terminals, and

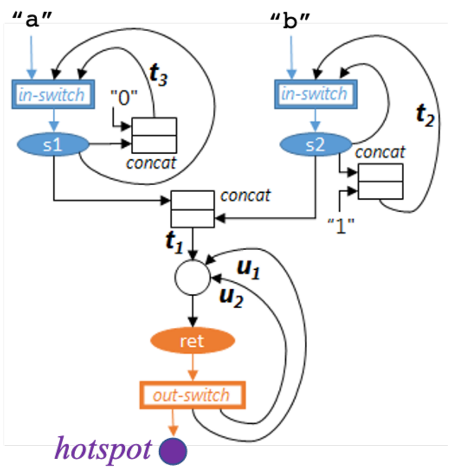

Figure 1. String value flow graph.

directed edges show dependencies between the nodes (non-terminals). The blue nodes labelled $s1$ and $s2$ in the graph represent the formal parameters of the function, while the brown node labelled *ret* represents its return node. To simplify the implementation, different return statements were joined into a single return node.

To capture the semantics of θ in the generated grammar we introduce switch nodes with sequences of slots that represent callsites. There is an *in-switch* node for every non-terminal representing a formal parameter and an *out-switch* node for every return non-terminal. The slots in the out-switch node correspond exactly to the slots in the in-switch node. In order to preserve this correspondence, we construct a linear order on the set of callsites of the function and arrange the in-switch and out-switch slots in that order. The non-terminals corresponding to actual parameters are connected to the in-switch slot (that is reserved for that callsite) instead of the formal parameter non-terminals.

To collect potential string values that can flow into a given node in the SVFG, the OLSA utilises a graph traversal algorithm to traverse the graph backwards. It can start from any node and stop at some end node that represents either a string constant or a user-input value. We explain the key aspects of the implementation using the example shown in Figure 1. For demonstration purposes, we only consider 6 types of nodes in this algorithm.

- **constant**: a node with only one outgoing edge, such as nodes "0", "1", "a" or "b".
- **simple**: a node with an arbitrary number of incoming and outgoing edges, e.g., node *ret* in Figure 1.
- **unary operation**: a node with one incoming edge and one outgoing edge. The incoming string values will be applied a unary operation before being sent out. An example of a unary operation is `trim`.
- **binary operation**: a node with two incoming edges and one outgoing edge that represents a binary operation. For instance, binary operation node `concat` in Figure 1 models string concatenation.
- **in-switch**: a node with one or more incoming edges (connecting to the corresponding actual arguments from different callsites) and one outgoing edge (connecting to a formal parameter node of the function).
- **out-switch**: a node with one incoming edge (from the unique return node of a function) and one or more outgoing edges (connecting to a different callsite).

Note that after the SVFG is constructed, the order of the incoming edges of an in-switch node must be matched with the order of the outgoing edges of the out-switch node in terms of different callsites of the function. We use a *Context* stack to distinguish string propagation at different call sites. This can be seen as a light-weight implementation of the construction of the supergraph in the IFDS algorithm [14].

For in-switch nodes, we pop the current context value from the *Context* stack, generate string values for the incoming edges and then push the current context value back into the *Context*. Analogously, for out-switch nodes, we first determine the index of the current outgoing edge, push it into the *Context* stack, and then calculate the string values from its incoming edges. After that, the top of the *Context* stack will be popped.

During the graph traversal to propagate string values, we maintain a map of each node that has been visited under a given context along with the generated values. If a node has already been visited in the same context, we will directly retrieve the saved result and stop the traversal. Therefore, each node in the string value flow graph will be only visited once in a particular context and each cycle in the string value flow graph is actually unfolded at most once.

3 Experimental Results

We now describe our results of the analysis on codebases that have more than 1 million lines of code. Because open benchmarks such as SecuriBench [12] are not representative of real applications, the main results we present are on large proprietary internal codebases.

To evaluate the performance of OLSA we also compare the results with JSA. Since JSA did not scale to our internal system, we present the results of our comparison on benchmarks from the DaCapo suite [2] and the unit tests associated with JSA. We were unable to compare OLSA against neither Violist nor OSA. Even after a considerable effort, we were unable to get Violist working in our environment, and the implementation of OSA is not publicly available.

For our scalability experimentation we selected an internal enterprise system composed of a number of web applications. The entire system has 32 million lines of Java code calculated using `cloc` [1]. Because the analysis time depends on the number of hotspots, we also report the number of hotspots. All loops in the resulting grammar are traversed only once and the implementation supports only a few typical string operations such as `concat`, `replace` and `substring`. For all experiments reported in this section we output all possible strings computed by the analysis.

Our experiments were carried out on a 8-core 2.60GHz E5-2690 Intel Xeon processor with 128GB RAM, running 64-bit Oracle Linux™[2]. All runtimes are in seconds and ⊥ indicates that the analysis did not terminate after 10 minutes.

3.1 DaCapo

In the experiments using programs from the DaCapo suite we set the hotspots to arguments of print methods from the `java.io` and `javax.servlet.jsp` packages and additionally `java.class.forName` function.

Table 1 presents the results of the comparison with JSA over 15 selected DaCapo benchmarks. We used benchmarks from all available revisions of DaCapo excluding programs that failed to build to the intermediate representation used by OLSA. Both OLSA and JSA were run on all selected benchmarks. Of these JSA was able to complete analysis (within 10 minute limit) only for 3 programs (shown via Table 1), whereas OLSA analysed all used benchmarks in less then 9 seconds. Similarly to Li et al. [11] we attribute the scalability issues of JSA to inter-procedural alias analysis that often led to memory exhaustion in larger DaCapo benchmarks.

In addition to runtime, we compared the number of strings both tools were able to resolve (see Table 2), where a fully resolved string corresponds to a string literal, whereas a partially resolved string has a mixture of constants and unknown components. The constants in partially resolved strings provide useful information to the developer to determine if the report is actionable. For instance, the partially resolved string `"SELECT␣*␣FROM␣<XXXX>"` at an SQL command will need further investigation while partial string `"document.write(<img␣src=␣<XXXX>>"` is clearly a false positive. In sunflow both tools resolved (fully or partially) approximately 70% of the hotspots. In avrora and sunflow benchmarks, however, OLSA resolved more strings overall but the majority of these strings were resolved only partially. JSA, on the other hand was able to fully resolve slightly more strings then OLSA but had fewer partial strings. Such results demonstrate effectiveness of OLSA that was able to infer entire or partial strings in most cases and additionally was able to analyse over 4 thousand of hotspots in less then 9 seconds which makes this tool highly usable in practice.

3.2 Internal Codebases

For experiments with our internal codebase we used methods related to SQL injection (e.g., `java.sql.executeQuery`) and unsafe use of reflection (`java.class.forName`) as hotspots. They were motivated by a client security analysis where string arguments should not be tainted, i.e., controlled by a potential attacker.

Tables 3 and 4 summarise the OLSA performance over different web applications from our internal codebase. They present the results on all the web applications as well as the four largest (by number of hotspots) applications. The tables show the size of the application, the number of relevant hotspots and the percentage of hotspots that were fully resolved in these cases. Results show that OLSA string analysis is able to analyse enterprise-scale codebases. OLSA was able to generate potential strings for SQL injection and unsafe reflection over the entire codebase in slightly over one hour and in less than three hours respectively. Because many of the strings are fully resolved (i.e., as constants), the percentage of hotspots that need manual inspection (of around 20%) is acceptable for an in-depth security analysis.

Table 1. Comparing OLSA and JSA over DaCapo.

Codebase	Hotspots	JSA Runtime	OLSA Runtime
Entire codebase	4,304	⊥	8.77
xalan	637	⊥	0.82
derby	542	⊥	2.67
cassandra	301	⊥	0.49
bloat	748	324.35	0.66
avrora	40	941.25	0.25
sunflow	91	98.47	0.08

Table 2. JSA and OLSA string resolution over DaCapo.

	JSA Resolved		OLSA Resolved	
Codebase	Fully	Partially	Fully	Partially
xalan	⊥	⊥	40%	36%
derby	⊥	⊥	30%	47%
cassandra	⊥	⊥	34%	24%
bloat	53%	30%	36%	58%
avrora	43%	28%	38%	45%
sunflow	65%	4%	64%	4%

Table 3. Runtime SQL injection.

Size(KLoC)	Hotspots	OLSA Runtime	% Resolved
Entire codebase	33,966	8,735	78.8
3,048	5,896	711	61.5
1,821	3,270	371	85.1
953	2,248	897	81.5
858	2,059	590	55.3

[2]Oracle Linux™ is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Table 4. Runtime unsafe reflection.

Size(KLoC)	Hotspots	OLSA Runtime	% Resolved
Entire codebase	38,189	8,988	86.6
853	6,153	1137	78.0
953	4,501	802.1	87.3
593	3,047	572.5	77.9
269	2,444	385.8	79.2

3.3 JSA Unit Tests

Finally, to evaluate precision of our analysis, we analysed 303 small test programs from the JSA unit test suite. Each program contains a single hotspot and hard-coded inputs, that is, these programs have the exact set of strings an ideal analysis should compute.

Overall, JSA was more precise and produced ideal results for 32% of programs and identified strings partially in another 30% of cases. OLSA performed worse and could not correctly identify strings (either fully or partially) in 68% of cases. The reason OLSA has incorrect answers is because it is not field sensitive and ignores constructs involving arrays and other data structures. In principle OLSA can be extended to support such features, however experiments in other contexts suggested that the increase in runtime performance is also combined with lower precision. Because the hard-coded inputs used in the JSA unit tests are not representative of features in the larger codebases, OLSA was not extended to support field-sensitivity and data structures.

4 Conclusion

In this paper we have presented our experience in developing OLSA – a practical string analyser that scales to large enterprise-level codebases and produces useful results.

The key insight into OLSA is that alias analysis is very expensive and thus limits scalability and precision. Performing only intra-procedural data-flow analysis combined with a limited form of context-sensitive analysis gives us the necessary scalability in the context of bug-finding.

The results of experiments with JSA using our internal codebase show that OLSA can analyse large software using security-related configurations in a matter of hours and yield useful results with almost 80% of strings resolved. Spot checks indicate that unresolved strings reported by OLSA can in principle be solved by supporting more features such as field-sensitivity. The impact of adding field-sensitivity to scalability is unknown at this stage.

The experimentation comparing precision of OLSA with JSA over its unit tests shows the trade off between scalability and precision of the two tools, where JSA correctly identified more strings. Many of the strings incorrectly reported by OLSA are due to lack of support of arrays, loop productions, field sensitivity and aliasing, features that are likely to increase analysis runtime. Such unit tests, however, may not be representative of real programs. Further experiments with DaCapo benchmarks show that the precision of the tools is comparable. JSA, however, was only able to analyse of 3 programs within the allocated time budget.

Overall, we have observed that computing precise string expressions, often viewed as a desirable outcome for a string analysis tool, is not always useful. Obtaining precise results often requires costly analyses that may fail to scale to large codebases. This is especially the case for specific problems. For instance, as shown by our experiments, in computing queries that typically involve constant strings (e.g., static part of SQL queries), a simpler and faster analysis suffices.

Acknowledgements. We thank Raghavendra Ramesh for his initial ideas to this work. We also thank the anonymous referees for their detailed feedback.

References

[1] cloc. https://github.com/AlDanial/cloc, 2008.
[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190. ACM, 2006.
[3] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin. *String Analysis for Software Verification and Security*. Springer, 2017.
[4] T-H. Choi, O. Lee, H. Kim, and K-G. Doh. A practical string analyzer by the widening approach. In *APLAS*, volume LNCS, 4279, pages 374–388. Springer, 2006.
[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, volume LNCS 2694, pages 1–18. Springer, 2003.
[6] M. Christodorescu, N. Kidd, and W-H. Goh. String analysis for x86 binaries. In *PASTE*, pages 88–95. ACM, September 2005.
[7] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo. Transitioning Parfait into a development tool. *S&P*, 10(3):16–23, May/June 2012.
[8] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *ICFEM*, volume LNCS, 6991, pages 505–521. Springer, 2011.
[9] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654. IEEE, 2004.
[10] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *ASE*, pages 174–183. ACM, 2005.
[11] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond. String analysis for Java and Android applications. In *FSE*, pages 661–672. ACM, 2015.
[12] B. Livshits. Stanford SecuriBench. https://suif.stanford.edu/~livshits/securibench/download.html, 2005.
[13] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS*, volume LNCS, 3780, pages 139–160. Springer, 2005.
[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61. ACM, 1995.
[15] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, pages 598–608. IEEE Computer Society, 2015.
[16] A. Steinhauser and F. Gauthier. JSPChecker: Static detection of context-sensitive cross-site scripting flaws in legacy web applications. In *PLAS*, pages 57–68. ACM, 2016.