JoanAudit: A Tool for Auditing Common Injection Vulnerabilities

Julian Thomé
University of Luxembourg
Luxembourg
julian.thome@uni.lu

Lwin Khin Shar
University of Luxembourg
Luxembourg
lwinkhin.shar@uni.lu

Domenico Bianculli
University of Luxembourg
Luxembourg
domenico.bianculli@uni.lu

Lionel Briand
University of Luxembourg
Luxembourg
lionel.briand@uni.lu

ABSTRACT

JoanAudit is a static analysis tool to assist security auditors in auditing Web applications and Web services for common injection vulnerabilities during software development. It automatically identifies parts of the program code that are relevant for security and generates an HTML report to guide security auditors audit the source code in a scalable way. *JoanAudit* is configured with various security-sensitive input sources and sinks relevant to injection vulnerabilities and standard sanitization procedures that prevent these vulnerabilities. It can also automatically fix some cases of vulnerabilities in source code — cases where inputs are directly used in sinks without any form of sanitization — by using standard sanitization procedures. Our evaluation shows that by using *JoanAudit*, security auditors are required to inspect only 1% of the total code for auditing common injection vulnerabilities. The screen-cast demo is available at `https://github.com/julianthome/joanaudit`.

CCS CONCEPTS

• **Security and privacy → Information flow control**; • **Software and its engineering → Automated static analysis**;

KEYWORDS

Security auditing, static analysis, vulnerability, automated code fixing

ACM Reference format:
Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. JoanAudit: A Tool for Auditing Common Injection Vulnerabilities. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122822

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122822

1 INTRODUCTION

The most common and serious threats for Web applications include Cross-site scripting (XSS), SQL injection (SQLi), XML injection (XMLi), XPath injection (XPathi), and LDAP injection (LDAPi) [27] which enable attackers to inject malicious input through an *input source* which is then relayed to another system by means of a *sink*. It is important to detect vulnerabilities at an early phase rather than later stages of software development.

Security auditing, i.e., the examination of the source code for the purpose of detecting vulnerabilities, helps to detect vulnerabilities during the early phases of software development. However, without proper automation, this task is laborious, error-prone and not scalable; hence, security auditors need *automated tools* to facilitate the auditing process.

Auditing tools face the following challenges [39]: (1) they have to help auditors locate the vulnerabilities quickly in the source code; (2) they need to scale to the size of realistic Web systems; (3) they should generate reports that provide control-dependency information to detail how injection vulnerabilities reach the sink in order to eliminate false alarms quickly; (4) these reports should only provide information relevant to security; (5) they need to support various types of vulnerabilities, such as XSS, SQLi, XMLi, XPathi or LDAPi; (6) they need to support security analysis for the Java programming language, one of the most commonly used technologies for Web development in industrial context [5].

Challenges 1 and 2 are addressed by approaches based on taint analysis [14, 18, 21, 23, 30, 41, 42]. However, reports generated by these approaches typically contain data-flow analysis traces and lack *control-dependency information* (challenge 3), which is essential for security auditing. Indeed, conditional statements checks are often used to perform input validation or sanitization tasks; without analyzing such conditions, feasible and infeasible data-flows cannot be determined, causing many false warnings.

Challenge 3 is addressed by approaches based on symbolic execution [19, 47] which helps to identify and locate potential vulnerabilities in program code, and thus, could assist the auditor's tasks. Though symbolic execution approaches reason with control-dependency information, they have yet to address scalability issues (challenge 2) due to the path explosion problem [45]. Other approaches [44] report analysis results without any form of pruning

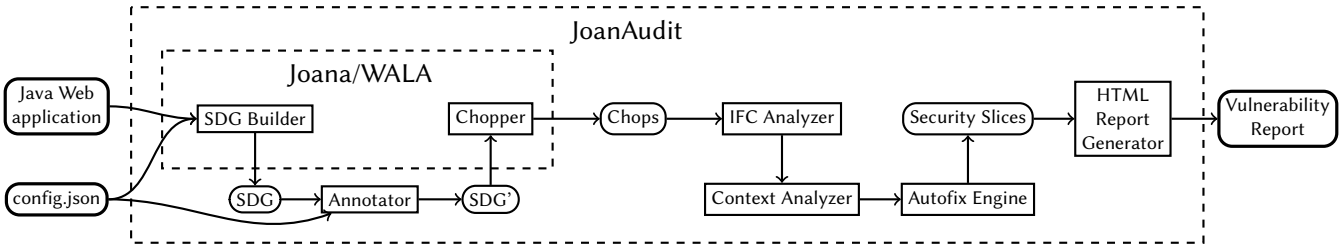

Figure 1: Architecture of *JoanAudit*

(challenges 1 and 4), thus containing a significant amount of information not useful to security auditing. As a result, an auditor might end up checking large chunks of code, which is not practical.

Challenge 5 is also not addressed by the majority of the above-mentioned approaches; the only exception is [30], which explicitly addresses XMLi, XPathi, and LDAPi.

Challenges 2, 4, and 5 are addressed by security testing approaches [1, 2, 16, 20, 37] and dynamic analysis-based security attack detection approaches [10, 24, 31–33, 35, 36]. These approaches can be used to detect XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. However, a security auditor is typically required to locate vulnerabilities in source code (challenge 1), identify their causes and fix them. Analysis reports from the above-mentioned approaches, though useful, are not sufficient to support code auditing since they only contain information derived from observed program behaviors or execution traces; they do not provide information about the location of the vulnerability in the source code.

Challenge 6 is generally addressed by black-box security testing approaches [2, 16, 37] because they are agnostic with respect to the programming language of the system under test. However, this is the same reason for which these approaches cannot locate vulnerabilities in the source code (challenge 1). Some security testing based approaches [10, 20] and static-analysis approaches [14, 30] do support Java but cannot meet challenges 1 and 3, respectively.

To facilitate security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities in program source code, in previous work we proposed an approach called *security slicing* [39, 40]. In this approach we first apply static analysis to identify the *input sources* and the sinks; afterwards, we apply program slicing and *code filtering* techniques to extract minimal and relevant source code that contains only statements required for auditing potential vulnerabilities related to each sink, pruning away other statements that do not require auditing.

In this paper, we present *JoanAudit*, a tool that implements our security slicing approach. It aims to provide security auditors a scalable way to locate and fix common injection vulnerabilities during the software development phase. *JoanAudit* addresses all of the above-mentioned challenges by (1) generating a vulnerability report that locates the vulnerabilities in the source code, (2) being scalable to Web systems realistic in size (52 kLOC), (3) extracting control- and data-dependency information from the program, (4) generating precise security slices that do not miss security-relevant information, (5) being readily configured for common injection vulnerabilities, and (6) targeting Java Web systems.

2 SECURITY SLICING WITH *JOANAUDIT*

Figure 1 illustrates the architecture of *JoanAudit*. The tool takes as input the bytecode of a Java Web application and a vulnerability catalogue (specified in the configuration file `config.json`), i.e., a pre-defined set of input source and sink signatures. For example, an input source could be the `getParameter()` function from the Java Servlet API for accessing HTTP POST parameters; a possible sink could be the `evaluate()` function from the `javax.xml` package which executes XPath queries on an XML database.

JoanAudit first constructs a System Dependence Graph (SDG), i.e., a graph model that captures inter-procedural data-, control-, and call-dependencies. An SDG is an ideal data structure for computing sound and precise program slices in linear time [12, 26]. The SDG is derived from the Java bytecode by the SDG builder. The latter also prunes functions that are irrelevant to security (e.g., logging libraries) or functions that are known or assumed to be free from security issues (e.g., standard security libraries). The list of irrelevant and known-good functions is predefined but can be configured in the vulnerability catalogue.

Afterwards, the Annotator annotates the SDG with input sources, sinks, and declassifiers. Based on the annotations in the SDG, the tool generates a program chop for each sink. Each program chop contains all the program statements that influence a sink, starting from the input sources, possibly through different program paths. Sinks that are not affected by any input source are pruned from the SDG.

The block labeled IFC Analyzer performs Information Flow Control Analysis (IFC) on each chop to determine if there are paths in the chop that can be considered secure due to the proper usage of sanitization functions and thus pruned. IFC is a well-known technique that checks whether a software system conforms to a security specification [11]. This step relies on a pre-defined set of declassifiers (standard sanitization procedures for preventing common injection vulnerabilities), which are configured in the vulnerability catalogue.

The block labeled Context Analyzer performs context analysis on the remaining paths. As part of this analysis, the block Autofix Engine attempts to patch, when feasible, the source code with the required security API. More specifically, this step uses a lightweight static analysis called *context analysis* to identify the context in which the data from an input source is used in the sink. Based on the identified context, this technique is able to automatically fix a vulnerable input source by applying the appropriate sanitization function to it. This technique is always guaranteed to properly fix a given vulnerability because it applies a fix only 1) in case of a

id	Source	Sink	Len	Vulnerability
0	simple/Simple.java:96	simple/Simple.java:101	8	xss
1	simple/Simple.java:95	simple/Simple.java:117	18	xss
2	simple/Simple.java:96	simple/Simple.java:125	11	xpath injection
3	simple/Simple.java:95	simple/Simple.java:101	9	xss
4	simple/Simple.java:96	simple/Simple.java:117	17	xss
5	simple/Simple.java:95	simple/Simple.java:125	12	xpath injection
6	simple/Simple.java:95	simple/Simple.java:107	10	xss
7	simple/Simple.java:96	simple/Simple.java:116	16	xss
8	simple/Simple.java:96	simple/Simple.java:107	9	xss
9	simple/Simple.java:95	simple/Simple.java:116	17	xss

Figure 2: The overview page of the report generated by *JoanAudit* shows all potentially vulnerable paths found

direct data flow from an input source to a sink, and 2) if the context of the user input can be determined.

As output, the tool generates a report that guides the security auditor in auditing potentially vulnerable parts of the program. Figure 2 shows the main page of the report generated by *JoanAudit*, which gives an overview of all the paths from the security slices that were extracted by *JoanAudit*. The report indicates how many potentially vulnerable paths have been detected. Every row in the overview table represents a single path; it details the location of the sources and sinks in the source code, i.e., a combination of the scope or class in which the source/sink was found, and the line number of the source file. Moreover, the report indicates the path size (in terms of program statements) and vulnerability to which the path may be vulnerable.

After clicking on one of the rows in the overview table, the detailed information for the respective paths is displayed in an extra window (Figure 3), which shows the actual source code of the analyzed program and highlights the individual program statements belonging to the selected path. In this view, the source code line numbers are shown on the left; the scope, i.e., the class where the potential vulnerability has been found, is displayed at the top; source and sink, respectively, are the first and last highlighted statements in the code snippet. Notice that only the security-relevant parts are highlighted. This detailed view guides the security auditors from the input source to the potentially vulnerable sink.

3 IMPLEMENTATION

The implementation of *JoanAudit* comprises approximately 11 kLOC (excluding library code) and is based on *Joana* [9, 11] and IBM's *Wala* framework [15]; *Joana* provides APIs for SDG generation from Java bytecode, program slicing, and IFC analysis.

The tool is configured with the JSON file `config.json` which contains a list of Java bytecode signatures for input sources, sinks, and declassifiers. The `config.json` file also specifies the list of bytecode signatures for known-good and irrelevant APIs. Note that *JoanAudit* is highly customizable: based on their domain knowledge, developers can specify in `config.json` additional input sources, sinks, and custom declassifiers used in their applications. Thanks to this user-defined additional configuration, the tool will not skip analyzing other security-sensitive operations, and will not falsely report as insecure the paths containing custom declassifiers. The following excerpt from the `config.json` shows an example configuration for the sink corresponding to the function `evaluate` (from the `javax.xml` package).

[-] simple/Simple.java 95:125

```
95          String account = req.getParameter("account");
96      String pass = req.getParameter("pass");
97      String balance = allowUser(account, pass);
98  }
99  protected String allowUser(String account, String password) {
100     Document doc = null;
101     DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
102     DocumentBuilder builder = null;
103     try {
104         builder = factory.newDocumentBuilder();
105     } catch (ParserConfigurationException e2) {
106         // TODO Auto-generated catch block
107         e2.printStackTrace();
108     }
109     try {
110         doc = builder
111                 .parse("/Users/julian/Documents/workspace.safe/joana-to-simple-se
112     } catch (SAXException e1) {
113         // TODO Auto-generated catch block
114         e1.printStackTrace();
115     } catch (IOException e1) {
116         // TODO Auto-generated catch block
117         e1.printStackTrace();
118     }
119     XPathFactory xPathfactory = XPathFactory.newInstance();
120     XPath xpath = xPathfactory.newXPath();
121     String q = "/users/user[@nick='" + account + "' and @password='"
122             + password + "']";
123     try {
124         NodeList nl = (NodeList) xpath.evaluate(q, doc,
125                 XPathConstants.NODESET);
```

Figure 3: Security slice containing only the program statements relevant to security (highlighted statements)

```
"sinks":[{
  "name": "javax.xml.xpath.XPath.evaluate(Ljava/lang/String,Ljava/
       lang/Object;)Ljava/lang/String;",
  "labels": "1(H)"
}]
```

The configuration entry for the sink is a JSON object with a `name` attribute, i.e., the bytecode signature of the sink (in bytecode format), and a `labels` attribute that specifies the index of the parameter to be tracked and its security level. The security level is important for IFC in order to detect an actual security violation. In the example configuration, we label the first parameter of the `evaluate` function with security label `H` (high integrity) which requires that data arriving at the sink should not be tampered with. The configuration for sources and declassifiers is done similarly; the detailed overview of the configuration is available at `https://github.com/julianthome/joanaudit`.

4 SUMMARY OF THE EVALUATION RESULTS

We evaluated the precision, soundness, and scalability of *JoanAudit* on 9 Web applications/services, representing a selection of commonly used benchmarks (see [39, 40] for details); the average program size is 17 kLOC with the largest program having 52 kLOC, which is fairly typical for this type of systems. For space reasons, we only provide a summary of the evaluation results; the full description of the evaluation methodology and the detailed results are available in our previous work [39, 40].

To assess precision (i.e., the reduction of source code in terms of security-relevant statements), we compared the size of the slices produced by *JoanAudit* with the size of the slices produced by the state-of-the-art chopping implementation provided by *Joana*. Our

security slices are significantly smaller than their counterparts obtained through normal chopping: we achieved a mean reduction of 76% in terms of the number of SDG nodes. Furthermore, we could observe that on average, security slicing (for all the sinks in a given Web application) would require the audit of only approximately 1% of the whole application code. The size reduction of security slicing achieved with *JoanAudit* is directly correlated to the reduction of the manual effort required from security auditors for verifying vulnerable paths in the source code. Hence, these results clearly suggest that a significant reduction in code inspection can be expected when using *JoanAudit*.

Soundness requires our approach not to miss statements relevant to auditing XSS, SQLi, XMLi, XPathi and LDAPi vulnerabilities. We assessed soundness by manually comparing the security slices extracted by *JoanAudit* with their normal chop counterparts. We also manually inspected all the normal chops to determine whether *JoanAudit* had incorrectly dropped the whole chop from being reported. *JoanAudit* neither missed any information important for security auditing nor incorrectly dropped any chop.

In terms of scalability, *JoanAudit* took on average 27 s to analyze the individual test subjects; the slowest run took 124 s. These results show that *JoanAudit* exhibits good run-time performance, which makes it suitable to analyze Java Web applications similar in size to our test subjects, which is the case for many such systems.

5 TOOL AND DATA AVAILABILITY

The tool and the data-sets used for the evaluation, the installation and user manuals alongside with the screen-cast are available on the tool website: `https://github.com/julianthome/joanaudit`.

The tool is available as a Java executable, included in a Docker container, and can be freely used for research purposes.

6 RELATED WORK

The main static analysis tools and frameworks which can be used for security analysis of Java Web applications are *FindBugs* [3, 13], *SFlow* [14], *FlowTwist* [21], *LAPSE+* [30], *TAJ* [42], *Andromeda* [41], *Indus* [17], *Soot* [43], *Joana* [11], *Wala* [15]; for a more comprehensive list see [28].

FindBugs [3, 13] is a widely-used static analysis tool for finding bugs in Java programs based on a set of bug-patterns; vulnerabilities can be identified from its bug report.

SFlow, *FlowTwist*, *LAPSE+*, *TAJ* and *Andromeda* are static taint analysis tools that track the data-flows between sources and sinks. *SFlow* and *FlowTwist* require the developers to specify sources and sinks, but this task can be laborious and error-prone. *LAPSE+* is already configured for the most common sources, sinks and sanitization operations. *TAJ* and *Andromeda* have been shown to be effective in their evaluation reports [41, 42], but they are not publicly available.

Furthermore, all of the above-mentioned approaches are based on taint analysis and, thus, do miss control-dependency information (challenge 3 in Section 1). *JoanAudit*, by contrast, is based on security slicing which leverages both control- and data-flow information from the program, and filters irrelevant information in order to identify program parts that are relevant for security.

Joana, *Indus*, and *Soot* are general-purpose slicing frameworks. *Joana* relies on the *Wala* libraries and provides a sound and precise approach for computing slices and chops. *Indus* is built on top of *Soot*, but is less precise than *Joana*, since it does not fully support interprocedural slicing [11]. As our approach and tool are built on top of *Joana*, we have the same advantages. Since these are general-purpose tools, their users have to provide slicing criteria suitable for their analysis goals (such as checking information flow and debugging). The generated slices may contain large amount of irrelevant information.

By contrast, *JoanAudit* is a specialized tool that targets security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. It is already configured with sources and sinks corresponding to these injection vulnerabilities, effectively defining the slicing criteria. Furthermore, it minimizes the size of the generated slices by applying various filtering techniques that prune irrelevant or secure code, making security auditing scalable and practical.

There are other static analysis tools for security analysis that target PHP systems: *RIPS* [6], *DEKANT* [25], *Stranger* [46], *PHPAspis* [29], and *Pixy* [18]. However, they are not applicable to Java systems due to the technical differences between the two languages.

7 CONCLUSION

We presented *JoanAudit*, a security slicing tool for auditing Java Web services and Web applications for common injection vulnerabilities, namely XSS, SQLi, XMLi, XPathi and LDAPi. Our current implementation is readily configured with a rich set of Java sources, sinks, and declassifiers. It can also be easily extended with new sources, sinks, and declassifiers for analyzing new types of vulnerabilities.

Our evaluation indicates that *JoanAudit* scales to realistic Web systems and significantly minimizes the amount of information extracted from source code, ultimately reducing the amount of manual effort required for security auditing.

As part of future work, we will adapt *JoanAudit* to widely-used Java Web frameworks such as *Spring* [34] or *Play Framework* [22], and we plan to integrate *JoanAudit* into build management tools like *Maven* [8] or *Gradle* [7]. We will also integrate the security slicing technique implemented by *JoanAudit* with symbolic execution [4] and string constraint solving techniques [38] to provide comprehensive tool support for the detection of injection vulnerabilities. Furthermore, we will conduct a user study to assess the usefulness of our tool in industrial settings.

ACKNOWLEDGMENTS

This work is supported by the National Research Fund, Luxembourg FNR/P10/03, INTER/DFG/14/11092585, and the AFR grant FNR9132112.

REFERENCES

[1] Nuno Antunes and Marco Vieira. 2013. SOA-Scanner: An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures. In *Proceedings of SCC 2013*. IEEE Computer Society, Washington, DC, USA, 280–287.

[2] Dennis Appelt, Cu Duy Nguyen, Lionel C. Briand, and Nadia Alshahwan. 2014. Automated testing for SQL injection vulnerabilities: an input mutation approach. In *Proceedings of ISSTA 2014*. ACM, New York, NY, USA, 259–269.

[3] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Experiences Using Static Analysis to Find Bugs. *IEEE Softw.*

25, 5 (2008), 22–29.
[4] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.
[5] Stephen Cass. 2016. The 2016 Top Programming Languages. `http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages`. (2016).
[6] Johannes Dahse. 2016. *Static detection of complex vulnerabilities in modern PHP applications.* Ph.D. Dissertation. Ruhr University Bochum.
[7] Adam Hans Dockter, Szczepan Murdoch, Peter Faber, Daz Niederwieser, Luke Daley Deboer, and Rene Gröschke. 2017. The Gradle Build Tool. `https://gradle.org`. (2017).
[8] Apache Software Foundation. 2017. The Apache Maven Project. `https://maven.apache.org/`. (2017).
[9] Jürgen Graf, Martin Mohr, Martin Hecker, Simon Bischof, and Tobias Blaschke. 2017. Joana - Information Flow Control for Java. `https://github.com/joana-team/joana`. (2017).
[10] William G. J. Halfond, Alessandro Orso, and Pete Manolios. 2008. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Trans. Softw. Eng.* 34, 1 (2008), 65–81.
[11] Christian Hammer. 2009. *Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs.* Ph.D. Dissertation. Karlsruhe Institute of Technology.
[12] Susan Horwitz, Thomas W. Reps, and David Binkley. 1990. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.* 12, 1 (1990), 26–60.
[13] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (2004), 92–106.
[14] Wei Huang, Yao Dong, and Ana Milanova. 2014. Type-Based Taint Analysis for Java Web Applications. In *Proceedings of FASE 2014*. Springer, New York, NY, USA, 140–154.
[15] IBM. 2017. T. J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`. (2017).
[16] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. 2016. Automated and Effective Testing of Web Services for XML Injection Attacks. In *Proceedings of ISSTA 2016*. ACM, New York, NY, USA, 12–23.
[17] Ganeshan Jayaraman, Venkatesh Prasad Ranganath, and John Hatcliff. 2005. Kaveri: Delivering the Indus Java Program Slicer to Eclipse. In *Proceedings of FASE 2005*. Springer, Berlin, Heidelberg, 269–272.
[18] Nenad Jovanovic, Christopher Krügel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of SP 2006*. IEEE Computer Society, Washington, DC, USA, 258–263.
[19] Adam Kiezun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. 2009. Automatic creation of SQL Injection and cross-site scripting attacks. In *Proceedings of ICSE 2009*. IEEE Computer Society, Washington, DC, USA, 199–209.
[20] Nuno Laranjeiro, Marco Vieira, and Henrique Madeira. 2014. A Technique for Deploying Robust Web Services. *IEEE Trans. Serv. Comput.* 7, 1 (2014), 68–81.
[21] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of SIGSOFT FSE 2014*. ACM, New York, NY, USA, 98–108.
[22] Lightbend and Zengularity. 2017. The Play Framework. `https://www.playframework.com/`. (2017).
[23] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of USENIX Security 2005*. USENIX Association, Berkeley, CA, USA, 18–18.
[24] Christian Mainka, Meiko Jensen, Luigi Lo Iacono, and Jörg Schwenk. 2013. Making XML Signatures Immune to XML Signature Wrapping Attacks. In *Proceedings of CLOSER 2013*. Springer, New York, NY, USA, 151–167.
[25] Ibéria Medeiros, Nuno Neves, and Miguel Correia. 2016. DEKANT: A Static Analysis Tool That Learns to Detect Web Application Vulnerabilities. In *Proceedings of ISSTA 2016*. ACM, New York, NY, USA, 1–11.
[26] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. In *Proceedings of SIGSOFT/SIGPLAN PSDE 1984*. ACM, New York, NY, USA, 177–184.
[27] OWASP. 2017. OWASP Top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`. (2017).
[28] OWASP. 2017. Static Code Analysis. `https://www.owasp.org/index.php/Static_Code_Analysis`. (2017).
[29] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *Proceedings of WebApps 2011*. USENIX Association, Berkeley, CA, USA, 2–2.
[30] Pablo Martín Pérez, Joanna Filipiak, and José María Sierra. 2011. LAPSE+ Static Analysis Security Software: Vulnerabilities Detection in Java EE Applications. In *Proceedings of FutureTech 2011*. Springer, Berlin, Heidelberg, 148–156.
[31] Abdul Razzaq, Khalid Latif, Hafiz Farooq Ahmad, Ali Hur, Zahid Anwar, and Peter Charles Bloodsworth. 2014. Semantic security against Web application attacks. *Inf. Sci.* 254 (2014), 19–38.
[32] Thiago Mattos Rosa, Altair Olivo Santin, and Andreia Malucelli. 2013. Mitigating XML Injection 0-Day Attacks through Strategy-Based Detection Systems. *IEEE Secur. & Priv.* 11, 4 (2013), 46–53.
[33] Hossain Shahriar and Mohammad Zulkernine. 2012. Information-Theoretic Detection of SQL Injection Attacks. In *Proceedings of HASE 2012*. IEEE Computer Society, Washington, DC, USA, 40–47.
[34] SpringSource. 2017. The Spring Framework. `https://spring.io/`. (2017).
[35] Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in Web applications. In *Proceedings of POPL 2006*. ACM, New York, NY, USA, 372–382.
[36] Zhao Tao. 2013. Detection and Service Security Mechanism of XML Injection Attacks. In *Proceedings of ICICA 2013*. Springer, Berlin, Heidelberg, 67–75.
[37] Julian Thomé, Alessandra Gorla, and Andreas Zeller. 2014. Search-based security testing of Web applications. In *Proceedings of SBST Workshop 2014*. ACM, New York, NY, USA, 5–14.
[38] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Search-driven String Constraint Solving for Vulnerability Detection. In *Proceedings of ICSE 2017*. ACM, New York, NY, USA, 198–208.
[39] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. 2017. Security slicing for auditing common injection vulnerabilities. (2017). `https://doi.org/10.1016/j.jss.2017.02.040`
[40] Julian Thomé, Lwin Khin Shar, and Lionel C. Briand. 2015. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities. In *Proceedings of ISSRE 2015*. IEEE Computer Society, Washington, DC, USA, 553–564.
[41] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Proceedings of FASE 2013*. Springer, Berlin, Heidelberg, 210–225.
[42] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of Web applications. In *Proceedings of PLDI 2009*. ACM, New York, NY, USA, 87–97.
[43] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of CASCON 1999*. IBM, Indianapolis, Indiana, USA, 13.
[44] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of SP 2014*. IEEE Computer Society, Washington, DC, USA, 590–604.
[45] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 3:1–3:42.
[46] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2010. STRANGER: An Automata-based String Analysis Tool for PHP. In *Proceedings of TACAS 2010*. Springer, Berlin, Heidelberg, 154–157.
[47] Yunhui Zheng and Xiangyu Zhang. 2013. Path sensitive static analysis of Web applications for remote code execution vulnerability detection. In *Proceedings of ICSE 2013*. IEEE Computer Society, Washington, DC, USA, 652–661.