Security Slicing for Auditing Common Injection Vulnerabilities  
Julian Thomé, Lwin Khin Shar, Domenico Bianculli, Lionel Briand  
SnT Centre for Security, Reliability and Trust  
University of Luxembourg, Luxembourg  
Email: {julian.thome, lwinkhin.shar, domenico.bianculli, lionel.briand}@uni.lu

Abstract  
Cross-site scripting and injection vulnerabilities are among the most common and serious security issues for Web applications. Although existing static analysis approaches can detect potential vulnerabilities in source code, they generate many false warnings and source-sink traces with irrelevant information, making their adoption impractical for security auditing.  
One suitable approach to support security auditing is to compute a program slice for each sink, which contains all the information required for security auditing. However, such slices are likely to contain a large amount of information that is irrelevant to security, thus raising scalability issues for security audits.  
In this paper, we propose an approach to assist security auditors by defining and experimenting with pruning techniques to reduce original program slices to what we refer to as security slices, which contain sound and precise information.  
To evaluate the proposed approach, we compared our security slices to the slices generated by a state-of-the-art program slicing tool, based on a number of open-source benchmarks. On average, our security slices are 76% smaller than the original slices. More importantly, with security slicing, one needs to audit approximately 1% of the total code to fix all the vulnerabilities, thus suggesting significant reduction in auditing costs.  

Keywords: Security auditing, static analysis, vulnerability, automated code fixing

1. Introduction  
Vulnerabilities in Web systems pose serious security and privacy threats such as privacy data breaches, data integrity violations, and denials of service. According to OWASP [38], injection vulnerabilities are the most serious vulnerabilities for Web systems. Among injection vulnerabilities, Cross-site scripting (XSS), SQL injection (SQLi), XML injection (XMli), XPath injection (XPathi), and LDAP injection (LDAPi) vulnerabilities are the most commonly found in Web applications and Web services. These vulnerabilities are usually caused by user inputs in security-sensitive program operations (sinks), which have no proper sanitization or validation mechanism.  
The majority of the approaches that deal with XSS, SQLi, XMli, XPathi, and LDAPi issues are security testing approaches [2, 6, 29, 59], and dynamic analysis approaches that detect attacks at runtime based on known attack signatures [32, 48, 44] or legitimate queries [55, 15, 52, 56]. However, a security auditor is typically required to locate vulnerabilities in source code, identify their causes and fix them. Analysis reports from the above-mentioned approaches, though useful, would not be sufficient to support code auditing as they only contain information derived from observed program behaviors or execution traces.
Approaches based on taint analysis [31, 24, 62, 42, 61, 20] and symbolic execution [26, 71] help identify and locate potential vulnerabilities in program code, and thus, could assist the auditor’s tasks. However, none of these approaches, except for the work reported in [42], seems to explicitly address XMLi, XPathi, and LDAPi. Hence, adapting these approaches to detect these types of vulnerabilities is a major need.

Furthermore, reports from taint analysis-based approaches only contain data-flow analysis traces and lack control-dependency information, which is essential for security auditing. Indeed, conditional statements checks are often used to perform input validation or sanitization tasks and, without analyzing such conditions, feasible and infeasible data-flows cannot be determined, thus causing many false warnings. Symbolic execution approaches reason with such conditions, but have yet to address scalability issues due to the path explosion problem [70]. Other approaches [68] report analysis results without any form of pruning (e.g., the whole program dependency graph), 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.

Program slicing [65] is one suitable technique that could help security auditors verify and fix potential vulnerabilities in source code. Like taint analysis, program slicing is also a static analysis technique, but it extracts all the statements that satisfy a given criterion, including control-flow and data-flow information, whereas taint analysis techniques only consider data-dependencies. However, there are also precision issues with slices since a large proportion of their statements may not be relevant to security auditing. Thus, without dedicated support, security auditing can be expected to be laborious, error-prone, and not scalable.

In this paper, our goal is to help security auditors, in a scalable way, to audit source code for identifying and fixing deficiencies in implemented security features. Our approach aims to systematically extract relevant security features implemented in source code. More precisely, to facilitate security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities in program source code, we apply static analysis to first identify the input sources (program points at which user inputs are accessed), and the sinks. Then, we apply program slicing and code filtering techniques to extract minimal and relevant source code that only contains statements required for auditing potential vulnerabilities related to each sink, pruning away other statements that do not require auditing.

The specific contributions of our approach include:

- **Sound and scalable security auditing.** We define a specific security slicing approach for the auditing of security vulnerabilities in program source code. Like taint analysis, our approach also uses static program analysis techniques, which are known to be scalable [61]. However, our analysis additionally extracts control-dependency information, which is often important for the security auditing of input validation and sanitization procedures. On the other hand, it filters out irrelevant and secure code from the generated vulnerability report. This ensures soundness and scalability.

- **Fully automated tool.** A tool called JoanAudit, which fully automates our proposed approach, has been implemented for Java Web systems based on a program slicing tool called Joana [16]. We have published the tool and the user manual online [58] so that our experiments can be replicated.

- **Specialized security analysis.** JoanAudit is readily configured for XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. In comparison, current program slicing tools are not specialized for such security needs; furthermore, most of the existing taint analysis tools do not readily support XMLi, XPathi, and LDAPi vulnerabilities.

- **Systematic evaluation.** We have evaluated our approach based on 43 programs from 9 Java Web systems, and analyzed 154 sinks from these Web programs. For each of them, a conventional slice was computed using Joana and a security slice was computed using our approach. Compared to the sizes of conventional program slices, our security slices are significantly smaller with reductions averaging 76%. Thus, the results show that our security slices are significantly more precise in terms of information relevant to security auditing. Based on manual verification, we also confirmed that the security slices are sound since all the information relevant to security auditing is extracted. From a practical standpoint, the results also show that by using our approach an auditor is required to audit approximately 1% of the total program code.
This paper is an extension of our prior work [60]. The main extensions include:

- **Types of vulnerabilities.** We address two more important types of vulnerabilities: XSS and LDAPi. XSS is currently the most common type of vulnerabilities in Web applications. LDAPi is also an important issue to address since LDAP directory services are increasingly used in enterprise Web applications.

- **Context analysis.** We provide a lightweight static analysis technique that extracts and analyzes path conditions from security slices to identify the context in which user inputs are used in a given sink and determine the appropriate sanitization procedures for securing those inputs. This information is used to fix some of the vulnerabilities automatically.

- **Experiments.** We conduct experiments on four additional Web systems to cover a larger variety of application domains, a wider system size range and new, additional types of vulnerabilities.

- **Detailed descriptions.** We provide detailed descriptions of the techniques (information flow control and automated code fixing) that we use to support code filtering. We also provide a detail description of the JoanAudit tool.

The paper is organized as follows: Section 2 illustrates some preliminary concepts; Section 3 gives an overview of the proposed security slicing approach; Section 4 presents the approach in detail; Section 5 discusses our prototype tool; Section 6 reports on the evaluation results; Section 7 discusses related work; Section 8 concludes the paper.

2. Preliminaries

In this section, we present some concepts used in the rest of the paper. We first provide a short overview of the injection vulnerabilities we address based on the definitions provided by OWASP [38], and introduce the concepts of input sources and sinks. We then discuss the program slicing techniques applied in our approach.

2.1. Injection Vulnerabilities

**XSS:** It is a code injection attack that injects client script code into the HTML code generated by the server program through user inputs, so that when a client visits the compromised Web page, the injected code is executed in the client’s Web browser, possibly accessing and transmitting client’s confidential information such as cookies. The injection is performed by inserting meta-characters or keywords specific to client-side script interpreters, such as `<script>` and `javascript`.

**SQL injection:** SQLi is an attack technique used to exploit applications that dynamically construct SQL queries by using user inputs to access or update relational databases. The attack makes use of meta-characters specific to SQL parsers, such as ‘,’ , #, and %, to alter the logic of the query.

**LDAP injection:** Similar to SQLi, LDAPi targets applications that dynamically build LDAP search filters using user inputs; the attack makes use of meta-characters specific to the LDAP search filter language [19], such as ( and & , to alter the logic of the query.

**XML injection:** XMLi is an integrity violation, where an attacker changes the hierarchical structure of an XML document by injecting XML elements through user inputs.

**XPath injection:** Similar to SQLi and LDAPi, XPathi is an attack technique used to exploit applications that construct XPath (XML Path Language) queries using user inputs to query or navigate XML documents. It can be used directly by an application to query an XML document as part of a larger operation, such as applying an XSLT transformation to, or executing an XQuery on, an XML document.
protected void doPost(HttpServletRequest req, ...) {
    String account = req.getParameter("account");
    String password = req.getParameter("password");
    String mode = req.getParameter("mode");
    if (mode.equals("login")) {
        allowUser(log, account, password);
    } else {
        createUser(log, account, password);
    }
}

protected boolean allowUser(String account, String password) {
    Document doc = builder.parse("users.xml");
    XPath xpath = xPathfactory.newXPath();
    String q = "/users/user[@nick='" + ESAPI.encoder().encodeForXPath(account) + "]/uni2423and/uni2423@password='" + ESAPI.encoder().encodeForXPath(password) + "]";
    NodeList nl = (NodeList)xpath.evaluate(q, doc, XPathConstants.NODESET);
}

protected void createUser(String account, String password) {
    String newUser = "<user/uni2423nick=" + ESAPI.encoder().encodeForXMLAttribute(account) + ">/uni2423password=" + ESAPI.encoder().encodeForXMLAttribute(password) + ">
    FileWriter fw = new FileWriter("users.xml");
    String newXML = "<users>
" + getPresentUsers() + newUser + "</users>
";
    fw.write(newXML);
}

Figure 1: Secure servlet with sanitization functions

2.2. Input Sources and Sinks

Input sources are operations that access external data that can be manipulated by malicious users. Specifically, in our approach, we define as input sources the accesses to: HTTP request parameters (e.g., getParameter()), HTTP headers, cookies, session objects, external files, and databases.

Sinks are operations that are sensitive to XSS, SQLi, XMLi, XPathi, or LDAPi. Specifically, we define the following elements as sinks:

- HTML document operations (e.g., javax.servlet.jsp.JspWriter.print());
- SQLi queries (e.g., java.sql.Statement.executeQuery());
- XML document operations (e.g., org.xml.sax.XMLReader.parse());
- XPath queries (e.g., javax.xml.xpath.XPath.evaluate());
- LDAPi queries (e.g., com.novell.ldap.LDAPConnection.search()).

We now illustrate XMLi and XPathi vulnerabilities and the concepts of input sources and sinks using the example in Figure 1 which we also use as running example throughout the paper.

The Java code snippet illustrated in Figure 1 grants or denies access to a Web application or service and/or creates a new user. The Java servlet interface implementation doPost() stores the values of three POST parameters (account, password, and mode) in variables that carry the same names. All the parameters are provided by the user of the Web application. If the mode parameter is equal to the string login, function allowUser() is called with account and password as parameters, to allow the user to access the application; otherwise, a new user account is created by invoking function createUser() with account and password as parameters. We assume that users credentials are stored in the XML document shown in Figure 2 and named users.xml.

The accesses to HTTP parameters at lines 2-4 are input sources. The XPath query at line 18 and the XML document processing operation at line 26 are sinks.

For granting or denying access, function allowUser() in Figure 1 executes the XPath query (sink) at line 18. This query compares the password—stored in the XML attribute password—for one of the entries...
in users.xml with the one accessed from an input source (the POST parameter password). In the example, the user inputs are sanitized at lines [16][17] by invoking methods from the OWASP Enterprise Security API (ESAPI) [39], which provides a rich set of sanitization functions for various vulnerability types. If the user input was used directly in the sink without such sanitization, the sink could be subject to XPathi attacks. For example, in the case of users.xml, by just knowing a user name, an attacker could launch a tautology attack using the value ‘or’1’=’1 as password, gaining access to the user’s credential data.

Likewise, in the absence of any sanitization, the operation at line 26 would be vulnerable to XMLi attacks. More specifically, at line 26 an XML tag is created with a user input using string concatenation. If the user inputs stored in account and password were not sanitized, as they are at lines 22–23, a user could compromise the integrity of the XML file by using one of the following meta-characters: < > / ‘ = ".

2.3. Program Slicing

Our terminology and definitions regarding security slicing are based on those of Hammer [16] since we rely on his program slicing approach and tool. Given a Web program, our security slices are extracted using program dependence graphs, system dependence graphs, backward program slices, and forward program slices of the program. The definitions for these concepts are provided below.

Definition 1. Program Dependence Graph [12]. A program dependence graph (PDG) is a directed graph $G = (N, E)$, where $N$ is the set of nodes representing the statements of a given procedure in a program,
and \( E \) is the set of control-dependence and data-dependence edges that induce a partial order on the nodes in \( N \).

Since a PDG can only represent an individual procedure, slicing on an PDG merely results in intraprocedural slices. For computing program slices from interprocedural programs, Horwitz et al. [18] defined system dependence graphs, which are essentially interprocedural program dependence graphs from which interprocedural program slices can be soundly and efficiently computed.

**Definition 2.** System Dependence Graph [18]. A system dependence graph consists of all the PDGs in the program, which are connected using interprocedural edges that reflect calls between procedures. This means that each procedure in a program is represented by a PDG. The PDG is modified to contain *formal-in* and *formal-out* nodes for every formal parameter of the procedure. Each call-site in the PDG is also modified to contain *actual-in* and *actual-out* nodes for each actual parameter. The call node is connected to the entry node of the invoked procedure via a call edge. The *actual-in* nodes are connected to their corresponding *formal-in* nodes via parameter-in edges, and the *actual-out* nodes are connected to their corresponding *formal-out* nodes via parameter-out edges. Lastly, summary edges are inserted between *actual-in* and *actual-out* nodes of the same call-site to reflect transitive data-dependencies that may occur in the called procedure.

Since an SDG provides an interprocedural model of a program—capturing interprocedural data-dependencies, control-dependencies, and call-dependencies—it is the ideal data structure for program analysis. Furthermore, program slices can be computed from it in a sound and efficient way in linear time [18, 57]. More specifically, the worst-case complexity of building a program slice from an SDG of \( N \) nodes is \( O(N) \); the worst-case complexity of building an SDG itself is \( O(N^3) \) [16].

![Figure 3](image.png) depicts the SDG of the program in Figure 1. The entry points of the methods `allowUser()`, `createUser()`, `encodeForXpath()`, `encodeForXMLAttribute()` and the main entry point `doPost()` are represented as SDG nodes (shaded boxes). The other nodes (white boxes), which represent the expressions of the program in Figure 1 are connected with control-dependence edges (black lines), data-dependence edges (black arrows) and summary edges (dotted black arrows). Call edges (dashed arrows with black arrowheads) connect call sites with their respective targets, whereas dashed arrows with white arrowheads denote parameter edges. Input sources are highlighted with a solid dashed frame, whereas sinks are highlighted with a blank dashed frame.

![Figure 4](image.png) The backward program slice with respect to the sink at line 18 in Figure 1.
Definition 3. Backward Program Slice \cite{18}. Given an SDG $G = (N, E)$, let $K \subseteq N$ be the set of identified sinks. The backward program slice of $G$ with respect to a target criterion $k \in K$, denoted with $bs(k)$, consists of all the statements that influence $k$, and is defined as $bs(k) = \{ j \in N \mid j \rightarrow k \}$, where $j \rightarrow k$ denotes that there exists an interprocedurally-realizable path from $j$ to $k$, so that $k$ is reachable through a set of preceding statements (possibly across procedures). The detailed algorithms for computing interprocedurally-realizable paths and backward slice are given in \cite{18}.

As illustrated in Figure 4, the backward program slice with respect to the sink at line 18 in Figure 1 contains all the program statements that influence (both intraprocedurally and interprocedurally) the operation of the sink.

Definition 4. Forward Program Slice \cite{9}. Given an SDG $G = (N, E)$, let $I \subseteq N$ be the source criterion. The forward program slice of $G$ with respect to $I$ consists of all the nodes that are influenced by $I$, and is defined as $fs(I) = \{ j \in N \mid i \rightarrow j \land i \in I \}$

The program in Figure 1 contains three input sources at lines 2-4. Figure 5 shows the forward program slice with respect to the input source $account$ at line 2.

Definition 5. Program Chop \cite{22, 46}. The program chop of an SDG $G = (N, E)$ with the source criterion $I$ and the target criterion $k$ is defined as $c(I, k) = bs(k) \cap fs(I)$.

Note that program chopping is defined as the intersection of backward slicing and forward slicing. It allows us to identify security-relevant nodes that are on the paths from $I$ to $k$ and, thus, involved in the propagation of potentially malicious data from input sources to a sink.

For example, Figure 6 shows a chop between the input sources $getParameter()$ on line 2 and the sink $xpath.evaluate()$ on line 18.

3. Overview of the Approach

Our fully-automated approach mainly targets Java-based Web applications, since the type of vulnerabilities it supports are commonplace in such systems. We emphasize that a specialized approach is necessary to provide practical support for the security auditing of Web applications and services developed using a specific technology.

When extracting security slices, we aim to achieve the following objectives:
Figure 6: The chop with the source criterion \{2, 3, 4\} and the target criterion \{18\} of the example program in Figure 1

1. **Soundness**: A security slice shall contain all the relevant program statements enabling the auditing of any security violation.

2. **Precision**: A security slice shall contain only the program statements relevant to minimizing the auditing effort.

3. **Performance**: The security slicing algorithm shall handle Web applications of realistic size.

Achieving all these objectives is desirable but in practice there is a trade-off between soundness and precision, depending on the analysis goal. In our context, we prioritize soundness because finding all the possible security violations is a priority for security auditing; nevertheless, we also try to optimize precision to the extent possible.

The pseudocode of the algorithm realizing our security slicing approach is shown in Figure 7. The algorithm takes as input: the bytecode $W$ of a Java program; a set $M_{(IR, KG)}$ of methods (custom functions or library API) that are either irrelevant to security analysis of XSS, SQLi, XMLi, XPathi, and LDAPi, or that may be relevant to security but are known (or assumed) to be correct or free from security issues; a set...

**Figure 7: Security slicing algorithm**
Figure 8: The security lattice used in our information flow control analysis

\begin{figure}
\centering
\includegraphics[width=0.2\textwidth]{security_lattice.png}
\caption{The security lattice used in our information flow control analysis}
\end{figure}

\(A_{(I,K,D)}\) of sources, sinks, and declassifiers (nodes in the SDG that represent sanitization procedures). The algorithm returns the set \(SS\) of security slices and associated path conditions extracted from \(W\).

The algorithm works as follows. After initializing \(SS\) to the empty set, it constructs the SDG from the bytecode \(W\) of the input program; this step is realized by using the API of Joana \cite{16}. The resulting SDG is then filtered by pruning nodes that contain methods belonging to \(M_{(IR,KG)}\); the details of this step are described in subsection 4.3. The next step identifies the set of input sources \(I\) and sinks \(K\) from the SDG. Afterwards, the algorithm iterates through the set \(K\); for all sinks \(k \in K\), it performs the following steps:

1. Computing the program chop \(c(I,k)\), to extract the program slice that contains the statements influenced by the set of input sources \(I\), which lead to sink \(k\) through possibly different program paths. This step is realized using the API of Joana.

2. Performing information flow control (IFC) analysis to identify how insecure the information flows along the paths in \(c(I,k)\) are. This step, partially supported by Joana, is described in subsection 4.1.

3. Performing context analysis to identify the context of sink and to understand whether input data is used in an insecure way in a sink. This analysis automatically patches vulnerable sinks with sanitization procedures if it is able to identify adequate procedures from the extracted path conditions \(PC\). If this is not possible, the extracted information can still be used to facilitate manual security auditing (e.g., checking feasible conditions for security attacks). This step is detailed in subsection 4.2.

Each of the last three steps is combined with a filtering procedure, based on the extracted information flow traces and path conditions; the filtering procedures are explained in subsection 4.3. Furthermore, each iteration terminates by computing a security slice \(ss(I,k)\) and its path conditions \(PC\), which are then added to set \(SS\).

4. Detailed steps

4.1. Information Flow Control Analysis

Information Flow Control Analysis (IFC) analysis is a technique that checks whether a software system conforms to a security specification. Relying on the work of Hammer \cite{16}, we adapt his generic flow-, context-, and object-sensitive interprocedural IFC analysis framework to suit our specific information flow problem with respect to XSS, SQLi, XMLi, XPathi, and LDAPi. Our goal is to trace how information from an input source can reach a sink, and then to analyze which paths in the chops are secure and which ones may not be secure.

We specify allowed and disallowed information flow based on a lattice called security lattice, i.e., a partially ordered set that expresses the relation between different security levels. We use the standard diamond lattice \(\mathcal{L}_{LL}\) [35], depicted in Figure 8, which expresses the relation between four security levels \(HL, HH, LL,\) and \(LH\). Every level \(l = L_0L_1\) contains two components: \(L_0\) denotes the confidentiality level while \(L_1\) denotes the integrity level. Confidentiality requires that information is to be prevented from flowing into inappropriate destinations or sinks, whereas integrity requires that information is to be prevented from flowing from inappropriate input sources \cite{49}. The element \(HL\) represents the most restricted usage, since any data labeled with it cannot flow to any destination that has a different security label. Data labeled with \(HH\) are confidential and cannot be manipulated by an attacker, whereas data labeled with \(LH\) are non-confidential.
and also cannot be manipulated by an attacker. The \textit{LL} label is used for data that are non-confidential but could be altered by an attacker.

All input sources and sinks are annotated with a security label that enables the detection of allowed and disallowed information flow. This annotation step is done automatically based on our predefined sets of input sources and sinks (see subsection 2.2). Input sources are labeled with \textit{HL} because data originating from them are supposed to be confidential but could be manipulated by an attacker. Sinks are labeled either with \textit{LH} or with \textit{HH}. The value of the confidentiality label is either \textit{L} or \textit{H}, depending on whether the sink is allowed or not to handle user confidential data. In any case, the integrity label for sinks is always \textit{H}, because only high-integrity data should be allowed to flow into the sinks, to prevent the flow of malicious input values causing security attacks. More specifically, in our approach we label as \textit{HH} the sink functions that update or modify databases—since it is common to store highly-confidential data in back-end databases—as well as the functions that access server environment variables, read data from configuration files or other sources. Moreover, we label as \textit{LH} the sink functions that generate outputs to external environments, such as exception handling functions, as well as functions that read time and date such as \texttt{getTime()} from \texttt{java.util.Calendar}. Finally, an example of function labeled with \textit{LL} is a function that monitors mouse-clicks.

Based on these annotations, the IFC analysis traces information flow from one node in the chop to another and detects disallowed information flow and, therefore, security violations. For example, a security violation is detected if there exists an information flow from an \textit{LL} input source to an \textit{HH} sink.

Notice that the annotation procedure must also take into account the fact that program developers might use sanitization procedures to properly validate data from an input source before using it in a sink. For instance, this is the case for our running example in Figure 1, where proper sanitization procedures (lines 16–17 and 22–23) taken from the OWASP security library \cite{39} are used between the input sources and a sink. Such cases can be considered secure and do not need to be reported to an auditor. To support the use of these functions, we rely on the concept of declassification \cite{50}. In our context, \textit{declassifiers} are nodes in the SDG that represent sanitization procedures. The integrity level of such nodes is annotated with an \textit{H} label since the sanitization procedure ensures the integrity of data. As we address five different vulnerability types, only the declassifiers relevant to the vulnerability type of a sink \textit{k} are annotated with the integrity level \textit{H}. Other declassifiers in the chop \textit{c(I, k)} and irrelevant for the vulnerability type of \textit{k} are ignored. For example, the declassifier at lines \texttt{16-17} in Figure 1 is relevant for the XPath function \texttt{xpath.evaluate()} at line \texttt{18} but is inappropriate for a sink of a different vulnerability category, e.g., an SQL query operation.

In addition to annotating the integrity level of declassifier nodes with \textit{H}, we also change the integrity level of the \textit{data that reach these nodes} to \textit{H}. For example, as shown in Figure 3, the input sources \texttt{account} and \texttt{password} (lines \texttt{2-3} in Figure 1) are annotated with the label \textit{HL}. Since these input values pass through the declassifiers at line \texttt{16-17} (highlighted in bold in Figure 3), their security labels are changed to \textit{HH}. When performing IFC analysis, the use of these variables in the sink node \texttt{xpath.evaluate()} at line \texttt{18} will be considered secure, because the information flow from \textit{HH} to \textit{HH} is allowed. Our tool is configured with the declassifiers (mainly encoding and escaping functions) from two widely-used security libraries—Apache Common \cite{5} and OWASP \cite{39}. It also recognizes the \texttt{PreparedStatement} function from the \texttt{java.sql} package as a declassifier corresponding to SQL sinks.

Consider now the same example above, but without sanitization functions. In such a case, we would have at least two illegal flows (from \texttt{account} and \texttt{password} to the \texttt{xpath.evaluate()} call) from \textit{HL} to \textit{HH}. Hence, their corresponding paths would be determined as potentially insecure and will be subject to context analysis, explained in the next subsection.

4.2. Context Analysis

The IFC analysis illustrated above can tell if data from input sources may reach sinks. However, from a security auditing standpoint it is also necessary to understand the context of a sink, i.e., how the input data is used in a sink and if it is used in an insecure way.

In this section, we present context analysis, a lightweight technique for identifying the context (within a sink) in which the data of an input source is used. Based on the identified context, this technique is able to automatically fix a vulnerable input source by applying the most appropriate sanitization function to it.
Table 1 lists, for each type of vulnerability that we consider, the possible contexts (in the form of patterns, where <input> correspond to the data from an input source). For each context, we indicate the most appropriate security API (provided by OWASP [39]) that should be used in that specific context to sanitize the input data.

Context analysis is lightweight compared to symbolic evaluation and constraint solving approaches [26, 71] because it traverses only the paths leading to the sink rather than the whole program, and does not attempt to precisely reason about the operations performed in the path (e.g., by performing constraint solving). Instead, the analysis merely examines the path conditions, i.e., the necessary conditions for the presence of information flow from input sources I to a sink k via a program path. More specifically, context analysis relies on path condition analysis to rule out infeasible paths, and to reconstruct the string values in the sink, needed to identify the context of the input source. The identified context is matched with the context patterns of Table 1. In case of a match, context analysis applies the corresponding fix, by wrapping the input source causing the vulnerability with the proper security API. Otherwise, in case there is no match and the input source cannot be fixed automatically, the procedure yields the path conditions, which represent a valuable asset for security analysts to understand the cause of a vulnerability.

To explain this analysis, we use the code snippet shown in Figure 9 and extracted from one of our test subjects WebGoat/MultiLevelLogin1 (see Section 6). The code is vulnerable to XSS because the input data, which is accessed from a database (source at line 12) and displayed as content of an HTML page (sink at line 27), could be tampered with by an attacker before the data is stored in the database.

Context analysis uses static single-assignment (SSA) form [11], a standard intermediate representation used in program analysis. In SSA form, every variable in a program is assigned exactly once and every variable is defined before it is used. For join points, i.e., points in the program where different control flow paths merge together, a Φ-operation is added to represent the different values that a variable can take at that point. Figure 9(b) shows the equivalent SSA form for the program in Figure 9(a).

The pseudocode of our context analysis function is shown in Figure 10. It takes as input a security API and returns a list of context patterns that match the input data. The function first identifies the context pattern of the input data and then applies the corresponding security API to sanitize the input.

```
<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>No.</th>
<th>Context Pattern</th>
<th>Security API</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSS</td>
<td>1</td>
<td>HTML element content: &lt;tag&gt;input&lt;/tag&gt;</td>
<td>ESAPI.encoder().encodeForHTML()</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>HTML attribute value: &lt;div attr='input'&gt;</td>
<td>ESAPI.encoder().encodeForHTMLAttribute()</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>URL parameter value: &lt;a href='http://...?param= input'&gt;</td>
<td>ESAPI.encoder().encodeForURL()</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>JavaScript variable value: &lt;script&gt;var a='input'...&lt;/script&gt;</td>
<td>ESAPI.encoder().encodeForJavaScript()</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>CSS property value: &lt;div onclick='var a='input'&gt;</td>
<td>ESAPI.encoder().encodeForCSS()</td>
</tr>
<tr>
<td>SQLi</td>
<td>6</td>
<td>SELECT column From table WHERE row='input'</td>
<td>ESAPI.encoder().encodeForSQL()</td>
</tr>
<tr>
<td>XMLi</td>
<td>7</td>
<td>XML element content: &lt;node&gt;input&lt;/node&gt;</td>
<td>ESAPI.encoder().encodeForXML()</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>CDATA content: &lt;![CDATA[input]]&gt;</td>
<td>ESAPI.encoder().encodeForXML()</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>XML attribute value: &lt;node attr='input'/&gt;</td>
<td>ESAPI.encoder().encodeForXMLAttribute()</td>
</tr>
<tr>
<td>XPathi</td>
<td>10</td>
<td>XPath attribute value: //table[column='input']</td>
<td>ESAPI.encoder().encodeForXPath()</td>
</tr>
<tr>
<td>LDAPi</td>
<td>11</td>
<td>LDAP distinguished name: LdapName dn = new LdapName( input)</td>
<td>ESAPI.encoder().encodeForON()</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>LDAP search: search=(attr='input')</td>
<td>ESAPI.encoder().encodeForLDAP()</td>
</tr>
</tbody>
</table>
```
slice $ss$ in a dependence graph form; it uses two local variables: $PC$, representing the set of preconditions analyzed, and $PV$, representing the set of potentially vulnerable paths.

First, the input security slice $ss$ is transformed by function $\text{GENICFG}$ into its equivalent interprocedural control flow graph (ICFG) form \cite{icfg}, which shows the order of control flow executions across procedures. In this form, the control flow paths in the slice become explicit and can be easily extracted.

Afterwards, function $\text{COLLECTPATHS}$ extracts the control flow paths by traversing the ICFG in a depth-first search manner. For practicability (to avoid path explosion), loops and recursive function calls are traversed only once; both our experience and the evidence gathered during our experiments confirm that analyzing one iteration of loops and recursive calls is sufficient to detect vulnerabilities. To illustrate this step, we use the ICFG of the program from Figure 9(b), shown in Figure 11. Every control flow edge is labeled with a sequence number; outgoing predicate edges are annotated with $\text{TRUE}$ or $\text{FALSE}$. In the figure, three control flow paths can be observed: $\{(1,8), (1,2,3,6,7,8), (1,2,3,4,5,7,8)\}$. However, for this program, the IFC analysis described in subsection 4.1 would have already pruned the paths $\{(1,8), (1,2,3,6,7,8)\}$ from the security slice, since there is no insecure information flow in those paths. Hence, function $\text{COLLECTPATHS}$ will return, in variable $PV$, only one potentially vulnerable path: $PV = \{(1,2,3,4,5,7,8)\}$.

The next step of the context analysis procedure is a loop that iterates over the set $PV$. For each path $p \in PV$, function $\text{EVALPATH}$ tries to automatically fix the vulnerability contained in $p$, if possible. Function $\text{EVALPATH}$, which takes in input a path $p$, works as follows. First, the path conditions $pc$ and the

Figure 9: The Java source code (a) and the equivalent SSA form (b) of a sample program

Figure 10: Context analysis algorithm
context of the input source $ctx$ of path $p$ are extracted with the EVAL procedure, described further below. Afterwards, function AUTOFIX identifies the required sanitization procedure by matching the extracted context $ctx$ against one of the context patterns shown in Table 1. If there is a match for $ctx$, the security API corresponding to the matched context pattern is applied to the input source; this automated fixing procedure is further explained in subsection 4.3. If function AUTOFIX returns a fix, procedure REMOVEPATH is invoked to prune the fixed path from the security slice $ss$, and EVALPATH terminates returning null. If fixing the vulnerability in $p$ is not possible, the EVALPATH function returns the path condition $pc$ corresponding to path $p$. The path conditions returned after executing the loop over $P_V$ are available in the set $PC$, which can be used by security auditors for manual inspection.

The extraction of the path conditions and of the context of a path is done through function EVAL, which works as follows. It traces, in reverse control-flow order starting from the sink, all the statements (in the SSA form) on which the sink variable is data- or control-dependent. Function TRACEBACKWARDS collects all the variables, their assignments and their interdependencies (stored in the map $Vmap$), including the conditions $Cond$ imposed on the variables at predicate statements. Function RESOLVEARIABLES resolves all variables until a fixed point is reached; the variables used in the sink are resolved as a concatenation of the program-defined values and the input variables. The result of the fix-point iteration is stored in the map $Vmap'$, which is then used by the GETSRCSNKPARAMS function to determine: 1) the variables that are associated with the input source $srcpar$, i.e., the value that is returned by the source operation; 2) the sink parameter $snkpar$, i.e., the string that is passed to the operation in the sink. With this information, function GETCONTEXT extracts the context of the input source with respect to the sink. The context is
returned together with the conjoined conditions in Cond to the evalPath procedure and stored in variables ctx and pc.

For example, after applying the eval procedure on the path $p = (1, 2, 3, 4, 5, 7, 8)$ in Figure 11 variable out$_8$ at the sink at line 27 is resolved to '<html><p>0', where u$_4$ represents the input variable assigned with the data from the input source at line 12. By matching this context against the context patterns of Table 1, it is identified as an input used as the context of an HTML element. The corresponding security API ESAPI.encoder.encodeForHTML() is then used to patch the input source at line 12 in Figure 9 resulting in the new statement String u = ESAPI.encoder.encodeForHTML(r.getString(1)).

Consider now the case in which the above vulnerable path $p = (1, 2, 3, 4, 5, 7, 8)$ could not be fixed by function autoFix. The following path condition pc would be reported:

DriverManager.getConnection(DB).prepareStatement("SELECT * ...").executeQuery().next() ∧ ¬u.isEmpty().

Based on this information, a security auditor may easily identify that the path is feasible as long as there are user data in the database. Hence, she may conclude that a security attack is feasible since there is no sanitization of the user input.

Note that our approach filters known-good classes (explained in the next subsection) such as those belonging to database drivers and database queries from the SDG. During SDG construction, those classes are replaced with stub nodes. Therefore, for the example above, the paths in the methods called by the DriverManager are not explored in our analysis. The considerable reduction of the number of analyzed path improves the scalability of our approach, and results in a simplified path condition, from which an auditor can still assess its feasibility.

4.3. Filtering

In this section, we describe the five filtering mechanisms applied to generate minimal slices for security auditing. For efficiency reasons, the filters are applied at different stages of our approach (as shown in our security slicing algorithm in Figure 7). Filter 1 and Filter 2 are applied concurrently during the SDG construction. Filter 3 is applied during program chopping. Filter 4 and Filter 5 are applied to the program chops in sequence. We mentioned earlier that the goal of our work is to achieve the highest possible precision while preserving soundness so that security auditing is scalable.

The original program chops $c(I, k)$ without filters are sound with respect to the types of input sources and sinks we consider, since all the statements related to those sources and sinks are extracted. It is straightforward to prove that by applying the filtering rules illustrated below, which remove statements that cannot be relevant to security auditing, we achieve better precision compared to the original program chops. However, we also need to demonstrate that we maintain soundness by not removing any statement that might be relevant to security auditing when filtering rules are applied. Therefore, when defining the filtering rules below, we provide arguments on how we preserve soundness. Further, we empirically demonstrate the soundness in Section 6.

The five filtering mechanisms used in JoanAudit are:

Filter 1: Irrelevant. It filters functions (custom functions or library APIs) that are irrelevant to the security analysis of XSS, SQLi, XMLi, XPathi, and LDAPi. Let $M_{IR}$ be the set of irrelevant functions. During the SDG construction, upon encountering a node that corresponds to a function $f \in M_{IR}$, a stub node is generated instead of the PDG that represents $f$. By doing so, all the nodes and edges that correspond to $f$ are filtered while not affecting the construction of the SDG. For security auditing purposes, the stub node is annotated with the name of the function and labeled as irrelevant.

Filter 2: Known-good. It filters functions with known-good security properties. Let $M_{KG}$ be the set of known-good functions. During the SDG construction, upon encountering a node that corresponds to a function $f \in M_{KG}$, a stub node is generated instead of the PDG that represents $f$. Therefore, like the filter above, all the nodes and edges that correspond to $f$ are filtered in such a way as not to affect the construction of SDG. For security auditing purposes, the stub node is annotated with the name of the function and labeled as known-good.

Basically, the above two filters correspond to 1) functions that are known to be irrelevant to the auditing of XSS, SQLi, XMLi, XPathi, and LDAPi issues; and 2) functions that may be relevant to security but are
known (or assumed) to be correct or free from security issues. Hence, it is clear that filtering such functions does not affect the soundness of our approach.

For example, we observed that Java methods belonging to classes responsible for retrieving the HTTP GET and POST parameters (e.g., those implementing the `javax.servlet.HttpServletRequest` interface) are commonly present in the original program chops; however — differently from the parameters they retrieve — these methods are irrelevant for our security analysis purpose because they contain neither input sanitization operations nor security-sensitive operations concerning XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. Example functions excluded by the known-good filter are the ones provided by widely-used security libraries, such as Apache [6] and OWASP [39] (e.g., the methods of the classes implementing the `org.owasp.esapi.Encoder` interface); these functions are assumed to be correct and thus do not require auditing.

In our tool, we predefine 12 functions as irrelevant and 50 functions as known-good. Program developers or security auditors may need to extend these sets of functions based on their domain knowledge; these sets can be easily defined in our tool through a configuration file.

**Filter 3: No input.** It filters sinks that are not influenced by any input source. This filtering is easily done by performing the program chopping with the source criterion \( I \) and the sink criterion \( k \). The resulting chop \( c(I, k) \) would be empty.

The sinks that are not influenced by any input sources cannot cause any security issues; thus, they are not relevant to security auditing. This implies that the resulting code, after applying Filter 3, is still sound and yet more precise.

**Filter 4: Declassification.** It filters out the secure paths from chop \( c(I, k) \). Let \( D_k \subseteq N \) be the set of declassifier nodes in SDG that corresponds to the type of sink \( k \). Let \( P \) be a set of paths from input sources \( I \) to \( k \). If there is a declassifier node \( d \in D_k \) on a path \( p \in P \), then the path \( p \) is removed from \( c(I, k) \).

The presence of a declassifier on a path \( p \) in \( c(I, k) \), which is adequate for securing the sink, ensures that values from input sources are properly validated and sanitized before being used in \( k \), as far as path \( p \) is concerned. Hence, the resulting code after filtering such paths is still sound and yet more precise.

This filter is applied using the IFC analysis discussed in [subsection 4.1](#). We use information flow control to filter out — from the set of paths that are returned to the security auditor — the paths that do not contain any violation according to the \( L_{IH} \) lattice.

**Filter 5: Automated fixing.** It automatically fixes the paths from input sources \( I \) to sink \( k \) that can be identified as definitely vulnerable and that can be properly fixed without user intervention. Let \( P \) be the set of remaining paths from chop \( c(I, k) \) after applying Filter 4. If a path \( p \in P \) identified as vulnerable can be fixed by applying an adequate security API, then the path \( p \) is removed from \( c(I, k) \). This filter corresponds to the \textsc{autoFix} procedure described in [subsection 4.2](#).

Automated fixing is not possible for all cases, especially when an input passes through complex string operations, like `substring()` and `replace()`, which are not addressed by our analysis. This is because there might be custom sanitization on the path using operations like `replace()` and in that case, applying another sanitization procedure on the path could affect the integrity of the input data and may not fix the security issue as intended. Therefore, automated fixing is only applied for the inputs directly used in the sink or for the inputs that only pass through simple string operations like `concat()`, `toUpperCase()`, and `trim()`, which do not have any (sanitization) effect on the input. For example, as discussed in [subsection 4.2](#) for the program in Figure 9 the fixing is applied to the input at line [12] because it only passes through the `concat()` and `toUpperCase()` operations before it is used in the sink. Fixing is also not possible when our analysis cannot determine the appropriate sanitization procedure to use, for example when it cannot identify the matching context due to complex code.

Anyway, since we apply the filter only on the paths that can be appropriately fixed, the resulting report after this filter is still sound and yet more precise for security auditing.

5. Implementation

We implemented our approach in a command-line tool called \textit{JoanAudit}, written in Java and publicly available [59]. It comprises approximately 11 kLOC, excluding library code. The tool is based on \textit{Joana} [10],
which is based on IBM’s Wala framework [21]. Joana provides APIs for SDG generation from Java bytecode, program slicing, and IFC analysis. Our tool also directly uses Wala’s APIs for some functionalities like ICFG generation and code optimization.

The tool is configured with two XML files, config.xml and lattice.xml. The first file contains a list of Java bytecode signatures for 74 input sources, 58 sinks, and 27 declassifiers; this list corresponds to the set $\Lambda(I,K,D)$ in the security slicing algorithm shown in Figure 7. The config.xml file also specifies the list of bytecode signatures for 50 known-good APIs and 12 irrelevant APIs; this list corresponds to the set $M_{IR,KG}$ in Figure 7 used in Filter 1 and Filter 2. The lattice.xml file specifies a configuration for the security lattice explained in subsection 4.1. Both files are configurable by users to suit their security analysis needs. For example, based on their domain knowledge, developers can specify in config.xml 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. Similarly, different security lattices (e.g., with finer-grained security levels) can be defined in lattice.xml.

Figure 12 illustrates the architecture of the tool. Given a Java Web application, JoanAudit performs the analysis steps presented in Sections 3–4. The bytecode of the application is converted to an intermediate representation based on the SSA form, which is then processed by the analysis steps. However, to facilitate security auditing, the tool outputs the security slice in source code format. The block labeled SDG Builder corresponds to the step at line 3 in Figure 7, which generates the SDG of the input program. The block labeled Annotator corresponds to the step at line 5 in Figure 7, which annotates the SDG with input sources, sinks, and declassifiers. Based on the annotations in SDG, the tool generates a program chop for each sink (line 7 in Figure 7). Sinks that are not influenced by any input source are filtered upon chopping (Filter 3). The block labeled IFC Analyzer performs on each chop the IFC analysis described in subsection 4.1. After computing the ICFG from the annotated SDG by means of the ICFG builder block (based on Wala’s API), for each chop, JoanAudit extracts the corresponding ICFG subgraph, from which the secure paths determined from the IFC analysis are filtered (Filter 4). The block labeled Context Analyzer performs context analysis (described in subsection 4.2) 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, as described in subsection 4.3 (Filter 5). As output, the tool generates a report that guides the security auditor in auditing potentially vulnerable parts of the program. An excerpt of report generated for one of our test subjects (WebGoat, see section 6) is shown in Figure 13.

The report contains potentially vulnerable paths (denoted as sequences of line numbers) and highlights the control-flow, data-dependencies, control-dependencies, and path conditions along these paths. The scopes (i.e., the classes to which the line numbers refer to) are parenthesized with squared brackets. The

\[\text{Figure 12: The architecture of our tool JoanAudit}\]
path conditions extracted during context analysis are returned in the format shown in Figure 14, in which sink and source variables are denoted with \textit{snk} and \textit{input}, respectively.

6. Evaluation

6.1. Research Questions

To evaluate whether our approach achieves precision, soundness and run-time performance when providing assistance to security auditing, we aim to answer the following research questions:

RQ1 (Precision) How much reduction can be expected from security slicing in terms of source code to be inspected? Is the reduction practically significant?

RQ2 (Soundness) Do we extract all the statements that are relevant to auditing XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities?

RQ3 (Performance) Does the tool scale to realistic systems in terms of run-time performance?

6.2. Test Subjects

Table 2 shows the 9 Web applications/services that we used in our evaluation. WebGoat \cite{WebGoat} is a deliberately insecure Web application/service for the purpose of teaching security vulnerabilities. It contains various realistic vulnerabilities that are commonly found in Java Web applications. Apache Roller \cite{Roller} and Pebble \cite{Pebble} are blogging applications that also expose a Web service APIs. Regain \cite{Regain} is a search engine that allows users to search for files over a Web front-end. PubSub \cite{PubSub} is the implementation of the open protocol PubSubHubbbub for distributed publish/subscribe communication \cite{PubSubHubbbub}, which is supported by many blogging applications and also used to access newsfeeds on the Internet. rest-auth-proxy \cite{rest-auth-proxy} is an LDAP-based Web service that authenticates users against an LDAP directory.

We selected WebGoat, Apache Roller, and Pebble since they are commonly used as benchmarks for security \cite{WebGoatBenchmark, RollerBenchmark, PebbleBenchmark}. The choice of Regain was driven by the fact that it is used in a production-grade system by dm, one of the biggest drugstore chains in Europe. TPC-App, TPC-C, and TPC-W are
Table 2: Test subjects

<table>
<thead>
<tr>
<th></th>
<th>LOC</th>
<th>#Prog.</th>
<th>#Sources</th>
<th>#Sinks</th>
<th>#Declassifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XML</td>
<td>XPath</td>
<td>XSS</td>
<td>LDAP</td>
<td>SQL others</td>
</tr>
<tr>
<td>WebGoat 5.2</td>
<td>24,608</td>
<td>14</td>
<td>34</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Roller 5.1.1</td>
<td>52,433</td>
<td>3</td>
<td>14</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Pebble 2.6.4</td>
<td>36,592</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Regain 2.1.0</td>
<td>23,182</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PubSub 0.3</td>
<td>1,964</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>TPC-App</td>
<td>2,082</td>
<td>6</td>
<td>22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TPC-C</td>
<td>9,184</td>
<td>6</td>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TPC-W</td>
<td>2,470</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rest-auth-proxy</td>
<td>442</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>152,957</td>
<td>43</td>
<td>106 (57)</td>
<td>4 (66)</td>
<td>3 (11)</td>
</tr>
</tbody>
</table>

Table 2 also reports the sizes of the test subjects in terms of lines of code (LOC), excluding the library code. The test subjects have an average size of 17 kLOC, and the largest one has 52 kLOC, which is fairly typical for that type of systems. The third column in Table 2 shows the numbers of Web programs (#Prog.), i.e., JSP, Java servlets and classes, contained in each test subject and analyzed by our tool JoanAudit. The table also reports the numbers of input sources (#Sources), sinks (#Sinks), and declassifiers (#Declassifiers) that JoanAudit identified. For sinks and declassifiers, the numbers are shown separately with respect to XSS, SQLi, XMLi, XPathi, and LDAPi. Some sinks are very general and are exploitable in various ways (e.g., sinks that allow attackers to load arbitrary classes server-side). Due to their universality, we also considered them in our evaluation and their number is listed in column others in Table 2.

All these test subjects can be obtained from the tool website [58].

6.3. Results

We ran our evaluation on an Apple MacBook Pro with an Intel Core i7 (2 GHz) and 8 GB of RAM, running Mac OS X 10.11, JVM version 25.31-b07, Joana rev. 688, Wala v.1.1.3, and OWASP ESAPI 2.0.

6.3.1. Precision

To answer RQ1, we compared the size of the slices produced by JoanAudit (hereafter referred to as “security slices”) with the size of the slices produced by the state-of-the-art chopping implementation provided by Joana (hereafter referred to as “normal chops”) extended with source/sink identification capabilities; in terms of size, we considered both the number of nodes and the number of edges. More specifically, for each sink $k$, we computed a security slice using our approach and a normal chop with the criterion $(I, k)$. We used the Wilcoxon signed-rank test over the slice sizes across Web programs in order to determine whether the differences in sizes of the two types of slices were statistically significant. We also discuss whether this difference is of practical significance in terms of auditing effort.

As shown in Table 2, we analyzed 43 Web programs from the 9 test subjects. For each Web program, an SDG was constructed. We computed normal chops and security slices from each SDG. The results are shown in Table 3. Overall, we computed 154 normal chops (#ch) and 39 security slices (#ss) from 106 sources and 154 sinks. The size (in terms of #nodes and #edges) of SDGs, normal chops, and security slices are shown in columns SDG, Chopping, and SecuritySlicing, respectively. Column #ss reports the final output of JoanAudit, i.e., the numbers of remaining security slices that require auditing after filtering has been performed. Some of the computed security slices are completely filtered (i.e., #ss=0) when, for example, all the paths in a slice are detected to be secured because of the presence of declassifiers. Furthermore, the
last four columns in Table 3 show the effectiveness of the five different filters presented in subsection 4.3 in terms of the number of nodes that are filtered.

To determine the amount of reduction achieved by security slicing when compared to normal chopping, we computed the relative size reduction of security slices with respect to (unfiltered) normal chop. The results (in percentage) are given in the columns (N%) and (E%) in Table 3. These results show that our security slices are significantly smaller than their counterparts obtained through normal chopping, in terms of both the number of nodes and the number of edges. As shown in the last two rows of the table, our approach achieved mean and median reductions of 76% and 100%, respectively, in terms of the number of nodes, and 79% and 100%, respectively, in terms of the number of edges. More importantly, 115 chops were completely dropped by the filters, meaning that only 39 out of total 154 chops require manual auditing (see columns #ch and #ss). Hence, one can expect significant practical benefits by adopting our approach.

The Wilcoxon signed-rank tests over 43 observations (#Prog.) show that the size reductions achieved with security slices are statistically significant at a 99% level of significance.

From the last four columns in Table 3, we can also observe how much each type of filters contributed. The known-good and irrelevant library-code-filters (F1+F2) significantly reduced the SDG size for all the test subjects. This can be explained by the fact that applications typically contain a large chunk of library code. The no input filter (F3) also significantly pruned many nodes (74,776 nodes in total) since those nodes are not influenced by any input source. The declassification filter (F4) significantly pruned many nodes from the standard chops (3,645 nodes in total), for all the test subjects except rest-auth-proxy. The automated fixing filter (F5) was significant for WebGoat, PubSub, and TPC-W (751 nodes were pruned in total).

To conclude, by comparing the security slice sizes and the SDG sizes in Table 3, we can observe that on average security slicing would require the audit of approximately 1% of the code for all the sinks in a given Web application. Since the security slices computed by our approach are based on the control-flow paths between sinks and sources, 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 answer RQ1 by clearly suggesting that a significant reduction in code inspection can be expected when using our approach.

We also remark that the above comparison shows the benefit of security slicing over normal chopping, with the latter performed by using a tool (Joana) that is also not easy to configure and use for standard engineers. Furthermore, for situations where security auditors have no access to program chopping tools, our approach can also indicate the percentage of the entire program code that has to be audited with security slices.

6.3.2. Soundness

To answer RQ2, we manually inspected all the security slices (39) returned by JoanAudit and compared them to their normal chop counterparts, to determine whether our security slicing approach had pruned any information relevant to auditing XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities.

To illustrate this manual inspection process, we use the simplified code excerpt below, which corresponds to a security slice extracted from the rest-auth-proxy/LdapAuthService program by JoanAudit.

```java
public AuthResponse authenticatePost(
    @FormParam("user") String user, // SOURCE
    @FormParam("pass") String pass
) {
    // ...
    LdapAuthentication ldap = getLdap(user);
    // ...
    ldap.authenticate(user,pass);
    // ...
}

private LdapAuthentication getLdap(String user) {
    // ...
    String sfilter = Configuration.get(Keys.LDAP_SFILTER);
    LdapAuthentication ldap = new LdapAuthentication();
    // ...
    if (!StringUtils.isEmpty(sfilter))
        ldap.setSearchFilter(sfilter.replaceAll("{user}", user)); // SINK
    ldap.setSearchFilter(sfilter.replaceAll("{user}", user)); // SINK
}
```
In the code above, function authenticatePost() can be called by a user to request authentication with the rest-auth-proxy web service; its inputs are the username (user, line 19) and the password (pass, line 20). Function getldap() creates an ldapAuthentication object, which manages all the communications with
Table 4: Execution time of the individual steps in JoanAudit (in ms)

<table>
<thead>
<tr>
<th></th>
<th>SDG Generation</th>
<th>Source/Sink Identification</th>
<th>Chopping</th>
<th>Filtering</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebGoat</td>
<td>21,774</td>
<td>201</td>
<td>59,427</td>
<td>42,278</td>
<td>123,680</td>
</tr>
<tr>
<td>Roller</td>
<td>5,079</td>
<td>64</td>
<td>16,125</td>
<td>1,241</td>
<td>22,509</td>
</tr>
<tr>
<td>Pebble</td>
<td>2,949</td>
<td>21</td>
<td>234</td>
<td>40</td>
<td>3,244</td>
</tr>
<tr>
<td>Regain</td>
<td>4,315</td>
<td>20</td>
<td>758</td>
<td>354</td>
<td>5,447</td>
</tr>
<tr>
<td>PubSub</td>
<td>2,876</td>
<td>41</td>
<td>367</td>
<td>224</td>
<td>3,508</td>
</tr>
<tr>
<td>TPC-App</td>
<td>16,297</td>
<td>112</td>
<td>2,157</td>
<td>4,349</td>
<td>22,509</td>
</tr>
<tr>
<td>TPC-C</td>
<td>8,089</td>
<td>63</td>
<td>3,931</td>
<td>6,664</td>
<td>18,747</td>
</tr>
<tr>
<td>TPC-W</td>
<td>7,590</td>
<td>31</td>
<td>313</td>
<td>3,044</td>
<td>10,978</td>
</tr>
<tr>
<td>rest-auth-proxy</td>
<td>945</td>
<td>6</td>
<td>6,220</td>
<td>25,765</td>
<td>32,936</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td><strong>7,768</strong></td>
<td><strong>62</strong></td>
<td><strong>9,948</strong></td>
<td><strong>9,329</strong></td>
<td><strong>27,107</strong></td>
</tr>
</tbody>
</table>

the LDAP backend server and stores configuration attributes that are important for user authentication (e.g., distinguished name, search filter, LDAP host address, port). First, the pre-configured search filter is loaded from the configuration file (line 14); then, an LdapAuthentication object is created (line 15). The pre-configured search filter can contain placeholders surrounded by curly brackets that are replaced with concrete values. For example, given the search filter \( \{\text{objectClass} = \text{inetOrgPerson} \& \text{uid} = \{\text{user}\} \) \( \), the placeholder \( \{\text{user}\} \) is replaced with the value provided with parameter user at line 18, and then the result is stored in the LdapAuthentication object through the setSearchFilter() method.

We started our manual inspection process at the sink (line 18), to determine the variables it uses (sfilter in the example above). Then, we tracked back its dependent statements to identify how the variables were processed. We determined that there was an unsanitized input at line 2 on which the sink in line 18 is data dependent. Hence, a user could alter the semantics of the search filter sfilter by injecting LDAP filter fragments such as \( ( ) * & | \) through the user variable at line 2. There was no known LDAPi vulnerability reported before for rest-auth-proxy; by using our tool, we detected a new LDAPi vulnerability and reported it to the developers.

In addition to inspecting security slices, we also manually inspected all the normal chops (154 chops) to determine if our security slicing had incorrectly dropped the whole chop from being reported (i.e., generating a false negative). Following a similar process, we verified that our security slicing approach neither missed any information important for security auditing nor incorrectly dropped any chop: this answers RQ2. The chops and their security slice counterparts are available on the tool website [58].

6.3.3. Performance

To answer RQ3, we measured the time taken for performing each step in the generation of security slices and normal chops; the results are shown in Table 4. JoanAudit took an average of 27s to analyze individual test subjects and required a maximum of 124s to analyze the largest one. 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.

Furthermore, we remark that the sum of the values in the columns “SDG Generation”, “Source/Sink Identification”, and “Chopping” corresponds to the execution time of the state-of-the-art chopping implementation provided by Joana extended with source/sink identification capabilities (i.e., normal chopping). The difference between this approach and ours lies only in the extra time taken by the filtering step, which on average accounts for 33% of the total time.

6.4. Threats to Validity

Our empirical evaluation is subject to threats to validity. The results were obtained from 9 selected Web applications, and hence, they cannot necessarily be generalized to all Web applications. We minimized this threat by choosing test subjects that vary in sizes and functionalities, and by picking realistic Java projects, which in many cases represent well-known benchmarks in the context of security.

We compared our approach, in terms of size reduction and performance, with a state-of-the-art chopping implementation provided by Joana extended with source/sink identification capabilities. Note that we
expect, however, to achieve similar results when comparing with other Java program slicing/chopping tools (e.g., Indus [23]) since our approach works on top of program chopping and is independent from the specific chopping tool we use.

Lastly, since our security slicing approach and tool are targeted towards Java Web applications, the approach may not produce the same results for Web applications written in other languages. Nevertheless, the fundamental principles of our approach are not language-specific and can be adapted to other languages using the corresponding program slicing tools (e.g., CodeSurfer [57] for C++).

7. Related Work

Related work that deal with the security auditing of XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities can be broadly categorized into two areas: static taint analysis and program slicing approaches.

7.1. Static Taint analysis

Static taint analysis approaches label data from input sources as tainted data and then detect vulnerabilities if the tainted data flows into sinks — which may be exploited by tainted data — without passing through any sanitization function (declassifier). Implementation of static taint analysis are available for Java Web systems [1, 31, 52, 63, 62, 20], for PHP Web systems [24, 47, 54, 50, 53], and for Android systems [7].

In general, there are three key differences between static taint analysis approaches and our security slicing approach. First, static taint analysis approaches tend to focus on data-flow based tainting only, and do not consider control-dependency information. This information is often essential for correctly identifying vulnerabilities or auditing the correctness of input sanitization procedures, since selection statements are often used to check user inputs. For example, consider the code snippet below, corresponding to a sampled, simplified slice, extracted from WebGoat:

```
1 String employeeId = req.getParameter('id'); // SOURCE
2 if (Integer.parseInt(employeeId) == EMPLOYEE_ID)
3     results = stmt.executeQuery("SELECT/uni2423*/uni2423FROM/uni2423employee/uni2423WHERE/uni2423userid/uni2423=" + employeeId); // SINK
```

In the above example, a taint analysis approach would falsely report a vulnerability. More specifically, it would detect a data-flow from the input source at line 1 to the sink at line 3, without considering that the sanitization achieved through the call to `parseInt()` at line 2 would have an impact on the value of `employeeId` itself. By contrast, our approach correctly identifies the path from line 1 to line 3 as secure due to the presence of the `parseInt()` declassifier; hence, it does not report a vulnerability. In general, lack of support for control-flow dependencies can be the source of many false positive results: Jovanovic et al.’s taint analysis tool [24] reported five false positives; Tripp et al. [61] reported 40% false positives on analyzing WebGoat; Shar and Tan [53] also reported that Livshits and Lam’s taint analysis approach [51] yielded 20% false positives due to missing control-dependency information. Although there are some taint analysis approaches ([10, 27, 51, 25, 72] that analyze control-dependency information, but they support programming languages different from Java and/or do not address injection vulnerabilities (with the exception of Dytan [10], which addresses SQLi in the context of dynamic taint analysis for x86 code).

Second, declassification is the only form of filtering provided by taint analysis approaches (e.g., as in [36]) whereas our approach additionally filters irrelevant and known-good library functions and also fixes some of the vulnerabilities automatically.

Last, our approach specifically targets XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities. Current taint analysis-based approaches address only SQLi and/or XSS. To the best of our knowledge, only Pérez et al. [42] readily address XMLi, XPathi, and LDAPi for Java Web systems. However, since Pérez et al.’s work is not evaluated, it is difficult to verify its effectiveness. Medeiros et al. [53] readily address XPathi and LDAPi but for PHP Web systems. It is possible to adapt existing approaches to support XMLi, XPathi, and LDAPi and even equip them with our proposed filtering mechanisms. However, since developers are often not security experts, these tasks may not be trivial. By contrast, our tool is already configured with an extensive library of input sources, sinks, and declassifiers specific to these vulnerabilities and thus, it can be used out-of-the-box.
7.2. Program slicing

Krinke [28] proposes barrier slicing approaches that could allow auditors to filter specific parts of the program that are known to be correct. Our approach makes use of this idea to prune Java libraries that are irrelevant to our security auditing purposes.

Despite the various slicing approaches proposed in the literature, in practice there are only two slicers that can handle all Java features: Indus [23] and Joana [16]. Indus is built on top of Soot [63], a Java bytecode analysis framework, and is less precise than Joana, since it does not fully support interprocedural slicing [16]. As discussed in section 3, Joana provides a sound and precise approach for computing slices and chops. As our approach and tool are built on top of Joana, we have the same advantages. However, Joana only generates slices for generic tasks like checking information flow and debugging. By contrast, we provide additional techniques for pruning statements in the slices produced by Joana and target security auditing of vulnerabilities. Therefore, Joana represents our baseline for comparison.

Shar and Tan [53] propose a program slicing-based approach for auditing the implemented defense features to prevent XSS. The approach of Yamaguchi et al. [68, 69] extracts abstract syntax trees and program dependence graphs relevant to auditing buffer overflow vulnerabilities in C/C++ code. The key difference between these approaches and ours is that they do not focus on minimizing the size of the extracted code, because their main objective is to extract all the possible defense features. By contrast, we extract all the features relevant for security auditing and yet, we also minimize the size of the extracted code by filtering irrelevant or secure code, making security auditing scalable and practical.

Backes et al. [8] present a program slicing-based approach for auditing privacy data leakage issues in Android code. Similarly to our approach, they also reduce SDG size by filtering known-good and irrelevant library code. But unlike our approach, they do not consider declassification and automated fixing. Further, as our objectives are different, the specifications of sources, sinks, and library APIs are also different. Hassanshahi et al. [17] propose an approach for detecting Web-to-App Injection (W2AI) attacks, an attack type where an adversary can exploit a vulnerable app through the bridge that enables interaction between the browser and apps installed on Android phones. Like our approach, they also make use of program slicing based on the ICFG in conjunction with a pre-defined set of sources and sinks. However, the main objective of their work is the detection of 0-day W2AI vulnerabilities rather than helping security analysts to audit source code for finding and fixing vulnerabilities of various kind.

8. Conclusion and Future Work

Injection vulnerabilities are among the most common and serious security threats to Web applications. A number of approaches have been developed to help identify many of those vulnerabilities in source code, such as taint analysis. However, they still generate too many false alarms to be practical, or miss some vulnerabilities. Therefore, they cannot effectively support security auditing by identifying and fixing vulnerabilities in source code in a scalable manner.

In this paper, we have presented an approach based on state-of-the-art program slicing, to assist the security auditing of common injection vulnerabilities, namely XSS, SQLi, XMLi, XPathi, and LDAPi. For every security-sensitive operation in the program, the approach extracts a sound and precise slice, along with path conditions, to help analysts perform security auditing on minimal chunks of source code. This is meant to complement current vulnerability detection approaches, by helping the auditor identify false positives and negatives. We implemented our approach in the JoanAudit tool, which we evaluated on 43 Web applications, generating 39 security slices. In comparison with conventional program slices, we observed that our security slices are 76% smaller on average, while still retaining all the information relevant for verifying common vulnerabilities. The tool and the test subjects used for the evaluation are available online [58].

As part of future work, we plan to enhance our approach by automating the verification of vulnerabilities. In particular, we aim to develop techniques that can make symbolic execution scale up, so that it can be applied for the feasibility analysis of the conjunction of path conditions with security threat conditions. 

23
Acknowledgment

We would like to thank Jürgen Graf and Martin Mohr from Karlsruhe Institute of Technology (KIT) for their kind and valuable help regarding Joana. This work is supported by the National Research Fund, Luxembourg FNR/P10/03, INTER/DFG/14/11092585, and the AFR grant FNR9132112.

References

References


