11-2015

Security slicing for auditing XML, XPath, and SQL injection vulnerabilities

Julian THOME

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Lionel BRIAND

Citation

THOME, Julian; SHAR, Lwin Khin; and BRIAND, Lionel. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities. (2015). *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE), Gaithersbury, USA, 2015 November 2-5*. 553-564.

Available at: https://ink.library.smu.edu.sg/sis_research/4778

Security Slicing for Auditing XML, XPath, and SQL Injection Vulnerabilities

Julian Thomé*, Lwin Khin Shar†, Lionel Briand‡
SnT Centre for Security, Reliability and Trust
University of Luxembourg, Luxembourg
Email: *julian.thome@uni.lu, †lwinkhin.shar@uni.lu, ‡lionel.briand@uni.lu

***Abstract*—XML, XPath, and SQL injection vulnerabilities are among the most common and serious security issues for Web applications and Web services. Thus, it is important for security auditors to ensure that the implemented code is, to the extent possible, free from these vulnerabilities before deployment. Although existing taint analysis approaches could automatically detect potential vulnerabilities in source code, they tend to generate many false warnings. Furthermore, the produced traces, *i.e.* data-flow paths from input sources to security-sensitive operations, tend to be incomplete or to contain a great deal of irrelevant information. Therefore, it is difficult to identify real vulnerabilities and determine their causes. One suitable approach to support security auditing is to compute a program slice for each security-sensitive operation, since it would contain all the information required for performing security audits (*Soundness*). A limitation, however, is that such slices may also contain information that is irrelevant to security (*Precision*), 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 pruning mechanism by using a number of open source benchmarks, we compared our security slices with the slices generated by a state-of-the-art program slicing tool. On average, our security slices are 80% smaller than the original slices, thus suggesting significant reduction in auditing costs.**

Keywords*—*Security auditing, static analysis, vulnerability

I. Introduction

Vulnerabilities in Web systems pose serious security and privacy threats such as the exposure of confidential data, loss of customer trust, and denial of service. According to OWASP [1], injection vulnerabilities are the most serious vulnerabilities for Web systems. Among the injection vulnerabilities, XML injection (XMLi), XPath injection (XPathi), and SQL injection (SQLi) vulnerabilities are commonly found in Web applications and Web services that use relational or XML databases. These vulnerabilities are usually caused by the use of user inputs in security-sensitive program statements (sinks) without proper sanitization or validation.

The majority of the approaches that deal with XMLi, XPathi and SQLi issues are security testing approaches [2], [3], [4], [5], and dynamic analysis approaches that detect attacks at runtime based on known attack signatures [6], [7], [8] or legitimate queries [9], [10], [11], [12]. 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 would only contain information derived from observed program behaviors or execution traces.

Approaches based on taint analysis [13], [14], [15], [16], [17], [18] and symbolic execution [19], [20] help identify and locate potential vulnerabilities in program code, and thus, could help with the auditor's tasks. Though none of these approaches, except for the work reported in [16], seems to explicitly address XMLi and XPathi, they could be adapted to detect these vulnerabilities.

However, reports from taint analysis-based approaches typically contain only data-flow analysis traces without *control-dependency information*, which may be essential for security auditing. Indeed, `if`-constructs or condition checks can be 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-based security analysis approaches have yet to address scalability issues due to the path explosion problem [21]. Other approaches [22] report analysis results without any form of pruning (*e.g.* the whole program dependency graphs), 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 [23] 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 affect a given criterion including control-flow and data-flow information, whereas taint analysis techniques only consider data-dependencies. However, this also causes a precision problem since a large chunk of a program slice may not be relevant to security auditing. Thus, without dedicated support, security auditing can be expected to be labor-intensive, 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 XMLi, XPathi, and SQLi vulnerabilities in program source code, we apply static analysis to first identify the sinks and sources, and then apply specific program slicing techniques to extract minimal and relevant source code that only contains statements required for auditing potential vulnerabilities related to each sink.

The specific contributions of this paper include:

978-1-5090-0406-5/15/$31.00 ©2015 IEEE

- *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 [17]. However, our analysis additionally extracts control-dependency information, which is often important for security auditing of input validation and sanitization procedures. Furthermore, it filters away 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* [24]. We have published the tool and the user manual online [25] so that our experiments can be replicated.
- *Specialized security analysis. JoanAudit* is readily configured for SQLi, XMLi and XPathi vulnerabilities. In comparison, current program slicing tools are not dedicated to such security needs while most of the existing taint analysis tools do not readily support XMLi and XPathi vulnerabilities.
- *Systematic evaluation.* We evaluated our approach based on 25 programs from five Java Web systems. We analyzed 84 sinks from those 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 80%. 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.

The paper is organized as follows: Section II provides background information for XMLi, XPathi, and SQLi vulnerabilities; Section III presents the proposed security slicing approach; Section IV evaluates the approach; Section V discusses related approaches; and Section VI concludes the study.

II. XML, XPath, and SQL Injection

In what follows, we give a short overview of the injection vulnerabilities we address based on the definitions provided by OWASP [1].

XML injection: XMLi attack is an integrity violation, where an attacker changes the hierarchical structure of an XML document by injecting XML elements through an *input source* (a program point at which data that can be manipulated by a malicious user is accessed).

XPATH injection: XPathi is an attack technique used to exploit applications that construct XPath (XML Path Language) queries using data from an input source 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 an XML document or applying an XQuery to an XML document.

SQL injection: Similar to XPath, SQLi is an attack technique used to exploit applications that construct SQL queries by using user inputs to access or update relational databases.

```
1  protected void doPost(HttpServletRequest req, /*...*/) {
2    String account = req.getParameter("account");
3    String password = req.getParameter("password");
4    String mode = req.getParameter("mode");
5    if(mode.equals("login")) {
6      if (allowUser(account, password)) // ...
7    } else { createUser(account,password) } // ...
8  }
9  protected boolean allowUser(String account,
10   String password)  { // ...
11   Document doc = builder.parse(XMLFILE);
12   XPath xpath = xPathfactory.newXPath();
13   String q = "/users/user[@nick='"+
        ESAPI.encoder().encodeForXPath(account) + "'␣and␣
        @password='" +
        ESAPI.encoder().encodeForXPath(password) + "']";
14   NodeList nl = (NodeList)xpath.evaluate(q, doc,
        XPathConstants.NODESET); // ...
15 }
16 protected void createUser(String account,
17   String password) {
18   String newUser = "<user␣nick=\"" +
        ESAPI.encoder().encodeForXMLAttribute(account) + "\"
        ␣password=\"" +
        ESAPI.encoder().encodeForXMLAttribute(password) + "
        \"␣/>";
19   FileWriter fw = new FileWriter(XMLFILE); // ...
20   bw.write("<users>\n" + getPresentUsers() + newUser + "\
        n</users>"); // ...
21 }
```

Listing 1. Secure servlet with sanitization functions.

For example, the Java code snippet illustrated in Listing 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. The `mode` parameter can be either `login` if a user wishes to get access to the application, or ≠`login` if a user wants to create a new user account. In the former case, the function `allowUser()` is called with `account` and `password` as parameters, whereas in the latter case these two parameters are passed to another function `createUser()` which is in charge of creating a new user. User credentials are stored in an XML file as illustrated in Listing 2.

```
<users>
  <user nick="alice" password="alicepass"/>
  <user nick="bob" password="bobpass"/>
</users>
```

Listing 2. XML file (`XMLFILE`) to store user credentials.

For granting or denying access, the function `allowUser()` in Listing 1 executes an XPath query (sink) at line 14 to compare the password stored in the XML attribute `password` for one of the entries in Listing 2 with the one accessed from an input source: the POST parameter `password`. In the example, the user inputs are sanitized in line 13 by calls to the OWASP Enterprise Security API (ESAPI) [26], which provides a rich set of sanitization functions for various vulnerability types. The calls to those functions are highlighted. If the user input was used directly in the sink without such sanitization, the sink would be subject to XPathi attacks. In Listing 2, by just knowing the user name, an attacker could launch a tautology attack using the value `' or '1' = '1` as `password` and would get access to the user's credential data.

Likewise, in the absence of any sanitization, the XML document processing operation (sink) in the function `createUser()` at line 20 would be vulnerable to XMLi attacks. An XML tag is created with a user input using string concatenation in line 20. If the user inputs account and password were not sanitized, as they are in line 18, a user could compromise the integrity of the XML file by using one of the following metacharacters: < > / ' = ".

III. Approach

Our terminology and definitions regarding security slicing are based on those of Hammer [24] as we rely on his program slicing approach and tool.

Since we intend to provide practical support for the auditing of XMLi, XPathi, and SQLi vulnerabilities in Web applications and services, our security slicing approach is targeted towards specific Web technologies (J2EE) since our targeted types of vulnerabilities are commonplace in such systems.

When extracting security slices, we aim to achieve the following objectives:

1) *Soundness:* A security slice contains all the relevant program statements enabling any security violation to be audited.
2) *Precision:* A security slice contains only relevant program statements relevant to minimize the auditing effort.
3) *Scalability:* The security slicing algorithm can handle programs of realistic size.

Achieving these objectives is desirable but in practice one has to compromise between soundness and precision depending on the analysis goal. In our context, we prioritize soundness since finding all possible security violations is a priority for security auditing, though we also try to optimize precision to the extent possible.

Our proposed, fully automated approach includes six main steps. To clarify our contributions, we distinguish the steps where we rely solely on *Joana*.

1) Construct a system dependence graph (SDG) from the bytecode of a Java program. An SDG contains interprocedural dependency information of all the statements in the program. Thus, it provides a foundation for program analysis (*Joana*).
2) Identify the set of input sources I and sinks S from the SDG, *i.e.* all the input sources from which XMLi, XPathi, and SQLi attack values might come from, and all the sinks that are sensitive to these attacks.
3) Compute program chop $C(I, s)$ for each identified sink $s \in S$. Program chopping is a form of program slicing which contains interprocedural control- and data-dependence information from the source criteria to the target criteria; in this case, from I to s. As such, every sink in a program can be audited using these chops (*Joana*).
4) Perform information flow control (IFC) analysis to extract information flow traces along each path from $C(I, s)$.
5) Filter each chop $C(I, s)$ based on the extracted information flow traces to generate a concise and minimal chop for security auditing, referred to as a security slice $SS(I, s)$.
6) Extract path conditions from each security slice $SS(I, s)$ to facilitate security auditing (*e.g.* checking feasible conditions for security attacks).

The above steps are implemented in *JoanAudit* [25] and its implementation information is provided in Appendix A. The following sub-sections describe the steps in detail.

A. System Dependence Graph Construction

Constructing an SDG is the first step of our approach since program slicing and chopping can be computed from the SDG of a given program. For the sake of completeness, we provide definitions below.

An SDG is an ideal data structure for program analysis because program slices can be soundly and efficiently computed from it in linear time [27], [28]. In other words, the complexity of building a slice or a chop from an SDG of N nodes is $O(N)$ while the worst case complexity of building an SDG itself is $O(N^3)$ [24].

To compute an SDG, program dependence graphs representing each procedure in the program have to be computed first.

Definition 1 (Program Dependence Graph) [29]. 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 program, and E is the set of control-dependence and data-dependence edges which induce a partial order on the nodes in N.

However, as a PDG can only represent an individual procedure, slicing on a PDG merely results in intraprocedural slices. For computing program slices from interprocedural programs, Horwitz *et al.* [27] 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) [27]. 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. But the PDG is then modified to contain *formal-in* and *formal-out* nodes for every formal parameter of the procedure. Each call-site in a 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 that *actual-out* parameters in a call-site are dependent on *actual-in* parameters.

Hence, SDG provides an interprocedural model of a Java program, capturing interprocedural data-dependencies, control-dependencies, and call-dependencies. Fig. 1 illustrates a simplified sample SDG that partly resembles the program in Listing 1 (just the doPost() and allowUser() methods).

B. Identification of Input Sources and Sinks

After the construction of the SDG model of the Java program, our approach identifies two classes of nodes in the

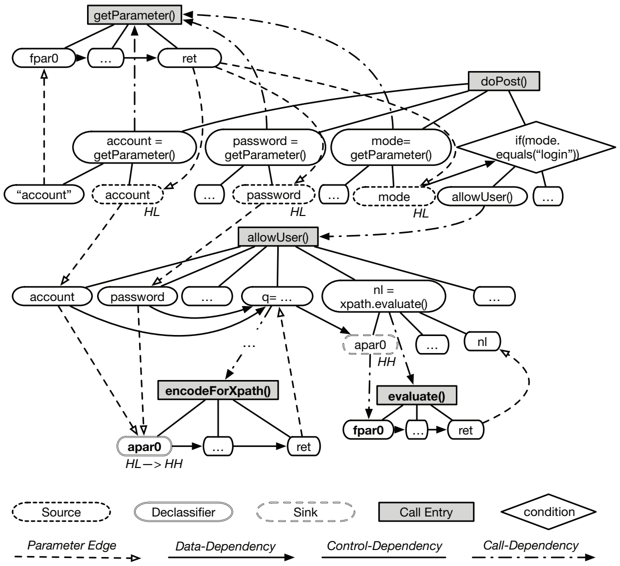

Fig. 1. Simplified SDG of the example program in Listing 1.

SDG that are required for program chopping which relies on a source criterion (*Input source*) and a target criterion (*Sink*):

1) *Input sources* are Web program functions or operations that access data which can be manipulated by malicious users. Specifically, in our approach, we define the following elements as input sources: accesses to HTTP request parameters (*e.g.* `getParameter()`), HTTP headers, cookies, session objects, external files, and databases.
2) *Sinks* are Web program functions or operations that are sensitive to XMLi, XPathi, or SQLi. Specifically, we define the following elements as sinks: XML document operations (*e.g.* `xmlobj.setTextContent()`), XPath queries (*e.g.* `xpath.evaluate()`), and SQLi queries (*e.g.* `sqlstmt.executeQuery()`).

In our prototype tool, the bytecode signatures of the above program functions and operations are predefined in configuration files to enable the tool to identify them from the SDG. In Fig. 1, the input sources that correspond to lines 2, 3 and 4 in Listing 1 are highlighted with solid dashed frames, whereas the sink that corresponds to line 14 is highlighted with a white dashed double stroke.

C. Program Chopping

We are interested in whether data values accessed from the identified input sources are used in the sinks. We, therefore, aim to extract a program slice that contains the statements influenced by a set of input sources, which lead to the sink through possibly different program paths. This is done by the following steps:

From the SDG, we first compute the backward program slice for each sink s.

Definition 3 (Backward Program Slice) [27]. The backward program slice of an SDG $G = (N, E)$ with respect to the target criterion $s \in S$, where S is the set of identified sinks with $S \subseteq N$, consists of all the statements that influence s:

$$BS(s) = \{j \in N \mid j \rightarrow^{\star} s\}$$

where $j \rightarrow^{\star} s$ denotes that there exists an *interprocedurally realizable path* from j to s, so that s is reachable through a set of preceding statements (possibly across procedures). The detail algorithms for computation of interprocedurally realizable paths and the backward slice are given by Horwitz *et al.* [27].

Afterwards, just the slices influenced by user input leading to s have to be extracted from $BS(s)$ by means of forward program slicing and chopping.

Definition 4 (Forward Program Slice) [30]. The forward program slice of an SDG $G = (N, E)$ with respect to the source criterion $I \subseteq N$ consists of all the nodes that are influenced by I:

$$FS(I) = \{j \in N \mid i \rightarrow^{\star} j \ \wedge \ i \in I\}$$

Definition 5 (Program Chop) [31], [32]. The program chop of an SDG $G = (N, E)$ with the source criterion I and the target criterion s is defined as:

$$C(I, s) = FS(I) \cap BS(s)$$

Basically, backward slicing allows us to extract all those statements that could influence the execution of a security-sensitive statement while forward slicing allows us to extract all the statements to which potentially malicious data from input sources could flow to. Hence, program chopping, the intersection of the two slices, allows us to identify security-relevant nodes that are on the paths from I to s and, thus, involved in the propagation of potentially malicious data from input sources to a sink.

For example, a chop between the input sources `getParameter()` on lines 2, 3, and 4 and the sink `evaluate()` on line 14 in Listing 1 is shown in Fig. 2. As illustrated, chopping allows us to just focus on the parts of the SDG that are interesting for security auditors, *i.e.* all paths that connect input sources to a sink.

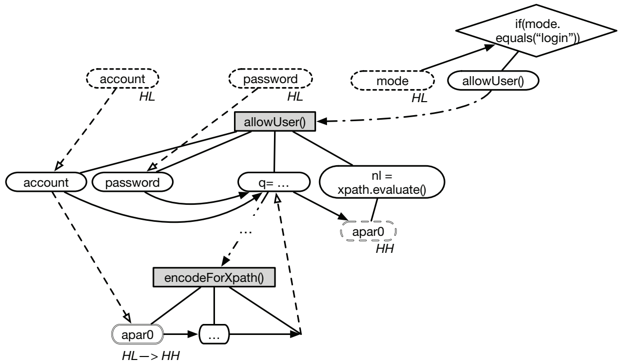

Fig. 2. The chop with the source criterion {2, 3, 4} and the target criterion {14} of the example program in Listing 1.

D. 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 [24], we adapt his generic flow-, context- and object-sensitive interprocedural IFC analysis framework to suit our specific information flow problem with respect to XMLi, XPathi, and SQLi. More specifically, we apply this technique to support the filtering mechanisms explained in the next sub-section.

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 partial-ordered set that expresses the relation between different security levels. We use a standard diamond lattice $\mathcal{L}_{LH}$ [33], as depicted in Fig. 3, that expresses the relation between four security levels HL, HH, LL, and LH. Every level $l = L_0L_1$ contains two components, *i.e.* a Confidentiality level L_0 and the Integrity level L_1. 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 [34].

Each input source and sink is annotated with a security label that enables the detection of allowed and disallowed information flow. Annotation is done automatically based on our predefined sets of input sources and sinks.

An input source, where data that is supposed to be secret but could be manipulated by an attacker is accessed, is labeled with HL – this data has the most restricted in usage as it cannot flow to any destination that has a different security label. In our approach, we label input sources like the `getParameter()` functions from the Java servlet API as HL since user confidential data is often obtained from such input sources, which can also be tampered with by an attacker.

Data labeled with HH is confidential and cannot be tampered with by an attacker. Data labeled with LH is non-confidential and also cannot be manipulated by an attacker. Hence, we use these two labels to annotate data that is expected to be secure in terms of integrity. In our approach, functions that access server environment variables read data from configuration files, *etc.* are labeled as HH, and functions that read time and date such as `getTime()` from `java.util.Calendar` are labeled as LH. The LL label is used for data that is non-confidential but could be influenced by an attacker, *e.g.* a function that monitors mouse-clicks.

A sink would be labeled with either LH or HH. Depending on whether the sink is allowed to handle user confidential data, the confidential label would be either L or H. But at all times, only high integrity data should be allowed to flow into the sink to prevent the flow of malicious input values causing security attacks. Thus, the integrity label is always H for the types of sinks we consider. In our approach, we label the sink functions that update or modify databases as HH since it is common to store highly confidential data in the back-end databases, whereas we label the sink functions that generate outputs to external environments (*e.g.* exception handling functions) as LH.

Based on annotations, we can trace information flow from one node to another and detect disallowed information flow and therefore security violations. For example, if there exists information flow from an LL input source to an HH sink, a security violation is detected.

However, one must also consider that program developers might use sanitization functions that properly validate data from an input source before using it in a sink. For our running example in Listing 1, the developer used proper sanitization functions (lines 13 and 18) between input sources and a sink. In the example, they used the OWASP security library [26] which provides sanitization functions for the vulnerability types we address. Such cases can be considered secure and do not need to be reported to an auditor.

The concept of declassification [35] can be used for this purpose. In our context, *declassifiers* are nodes in the SDG that represent sanitization functions. Whenever a user input passes through a declassifier, we modify its security level. In our case, as sanitization functions ensure the integrity of the data, the integrity level of the data that reaches the nodes corresponding to those functions would be changed to H.

For example, in Fig. 1, the input sources `account` and `password` that correspond to lines 2 and 3 of Listing 1 are annotated with the label HL. As these input values pass through the declassifiers at line 13 (highlighted in bold in Fig. 1), respectively, their security labels are changed to HH. Since the information flow from HH to HH is allowed according to the security lattice in Fig. 3, the use of those variables in the sink node `evaluate()`, at line 14 and highlighted in bold in Fig. 1, is considered secure.

However, if there were no sanitization functions, we would have two direct illegal flows (from `account` and `password` to the `evaluate()` call) and one illegal indirect flow (from `mode` to `evaluate()`) with $HL \rightarrow HH$, which is forbidden according to the security lattice in Fig. 3.

We assign declassifiers and sinks to different vulnerability categories. Depending on the vulnerability category of a sink, a corresponding vulnerability category of a declassifier is required to appropriately sanitize the input values used in the sink. For example, the declassifier in line 13 in Listing 1 is appropriate for the XPath function `xpath.evaluate()` in line 14, but is inappropriate for a sink of different vulnerability category, *e.g.* a SQL query operation. Table I lists vulnerability categories and their corresponding declassifiers from OWASP [26]. Our prototype tool is configured with a set of declassifiers provided by Apache [36] and OWASP [26]. It also recognizes `PreparedStatement` functions from the `java.sql` package as declassifiers corresponding to SQL sinks.

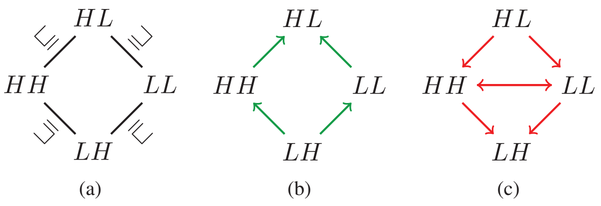

Fig. 3. Subfigure a highlights the partial order relation between the different security levels. Subfigure b shows the permitted information flow between security levels, whereas Subfigure c illustrates the disallowed information flow.

E. 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. Filter 1 and Filter 2 are applied concurrently during the SDG construction. Filter 3 is applied on the SDG, once constructed. Filter 4 and Filter 5 are applied to the program chops. 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 $C(I, s)$ 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 claim that by applying the filtering rules below, which remove statements that cannot be relevant to security auditing, we achieve better precision compared to the original program chops. However, we 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 will empirically demonstrate the soundness in the evaluation section.

Definition 6 (Filter 1: Irrelevant). Filter functions that are irrelevant to the security analysis of XMLi, XPathi, and SQLi. Let IR be the set of irrelevant functions. During the SDG construction, upon encountering a node that corresponds to a function f $\in IR$, 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*.

Definition 7 (Filter 2: Known-good). Filter functions with known-good security properties. Let KG be the set of known-good functions. During the SDG construction, upon encountering a node that corresponds to a function f $\in KG$, 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 XMLi, XPathi, and SQLi 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 soundness. For example, we observed that Java libraries responsible for invoking the HTTP GET and POST parameters are commonly present in the original program chops, though these libraries are known to be irrelevant for our security analysis purpose. We would also assume that input sanitization functions provided by Apache [36] and OWASP [26] are correct and do not require auditing. In our tool, we predefine 22 functions as *irrelevant* and 15 functions as *known-good*.

Definition 8 (Filter 3: No input). Filter those sinks that are not influenced by any input source. This means that if a sink s is not connected to any input source in I, it is removed from the SDG as well as all the edges leading to it.

Those sinks that are not influenced by any input sources would not cause any security issues and, thus, are not relevant to security auditing. Clearly, this implies that the resulting code, after applying Filter 3 to SDG, is still sound and yet more precise.

Definition 9 (Filter 4: Declassification). Filter the secure paths from chop $C(I, s)$. Let $D \subseteq N$ be the set of declassifier nodes in SDG, which corresponds to the type of sink s. Let P be a set of paths from input sources I to s. If there is a declassifier node $d \in D$ on a path $p \in P$, then the path p is removed from $C(I, s)$.

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

Note that this filtering process is performed using the IFC analysis technique discussed earlier. We use information flow control to filter out those paths, from the set of paths that are presented to the security auditor, that do not contain any violation according to the $\mathcal{L}_{LH}$ lattice.

Definition 10 (Filter 5: Automated fixing). Automatically fix those sinks that can be identified as definitely vulnerable and that can be properly fixed without user intervention. This means that, after fixing a vulnerable sink s, the whole chop $C(I, s)$ does not require to be audited.

A sink which directly uses the user input is definitely vulnerable. It can also be fixed by applying an adequate sanitization function on the input. For example, consider an XPath sink:

```
xpath.evaluate("/users/user[@account=’"+req.getParameter("
    account")+"’]")
```

The input `req.getParameter("account")` is directly used in the sink without going through any other program operations. Thus, while the sink is definitely vulnerable, it can also be automatically fixed by wrapping a standard sanitization routine around the input as below:

```
xpath.evaluate("/users/user[@account=’"+ESAPI.encoder().
    encodeForXPath(req.getParameter("account"))+"’]")
```

Clearly, automated fixing is not possible for all cases, especially when an input passes through functions and operations that cannot be reasoned with by our analysis. Fixing is also not possible when our analysis cannot determine the appropriate sanitization function to use according to the type of sink and how the input is used in the sink (*context*). Specifically, there are four types of cases where automated fixing is performed by our approach, which are listed in Table I.

First, the IFC analysis identifies the vulnerability category of a sink and the input which is directly used in the sink. It then extracts the query string from the sink statement and tries to find an appropriate sanitization function by matching the string with context patterns in Table I to identify the context in which the input appears (*e.g.* attribute or tag value). Extracting the query string may require tracking back the variables used in the sink, which is performed by traversing the nodes in the chop $C(I, s)$ backwards starting from s. No fix is made if pattern matching is unsuccessful. Fixes are applied in source code by replacing the original vulnerable statement with the modified,

TABLE I. AUTOMATED VULNERABILITY FIXING RULES

Vulnerability-Category of sink	Context	Context Pattern	Security API
SQLi	SQL attribute	`"Select attrb From table Where attrb="+input`	`ESAPI.encoder().encodeForSQL()`
XPathi	XPath attribute	`"/tag.../tag[@attrb|tag/text()="+input+"]"`	`ESAPI.encoder().encodeForXPath()`
XMLi	XML attribute	`"<element attrb="+input`	`ESAPI.encoder().encodeForXMLAttribute()`
XMLi	XML tag	`"<tag>"+input+"</tag>"`	`ESAPI.encoder().encodeForXML()`

secure statement as shown in Listing III-E. It is possible as we keep the mappings between the nodes in the chops and their corresponding source code line numbers.

Hence, as we filter only those cases that can be appropriately fixed, the resulting report after filtering them is still sound and yet more precise for security auditing.

The appropriate sanitization functions shown in Table I are from OWASP [26] and are configured in our tool. However, users may also choose to use their own set of sanitization functions by modifying its configuration file.

F. Path Condition

Our security slices provide an auditor with the information about *how* data from input sources could influence operations at sinks. However, information about *why* input sources could have an influence on sinks would be useful for security analysts to assess the risks.

Such information could be obtained by extracting *path conditions*. A path condition $PC(i, s)$ states the necessary condition for the presence of information flow from an input source $i \in I$ to a sink $s \in S$ via one or more paths. It can be computed from the security slice $SS(I, s)$ using the algorithm given by Snelting [37]:

1) compute all the paths P from I to s in $SS(I, s)$.
2) for every node $n \in p$ on a path $p \in P$, compute the execution condition $E(n)$. The execution condition $E(n)$ is a necessary condition for the execution of n, which can be computed by traversing the incoming control-dependence edges and collecting the predicates of the ancestor nodes, until a root node of $SS(I, s)$ is reached. Typically, $E(n)$ includes conditions from `if-`, `for-`, or `while-` statements.
3) a path condition for p is the conjunctive combination of the execution conditions:

$$PC(p) = \bigwedge_{n \in p} E(n)$$

4) when more than one path exist between i and s, the path condition of multiple paths is the disjunctive combination of the path conditions for individual paths:

$$PC(i, s) = \bigvee_{p = i \rightarrow^{\star} s} PC(p)$$

If the auditor finds that the condition is impossible, the corresponding path could be safely ignored for security auditing. On the other hand, if the conditions are satisfiable, the analyst could verify whether they allow insecure information flow and, thus, determine the causes of security vulnerabilities. To illustrate, in our running example program in Listing 1, a path exists from an input source at line 4 to the sink at line 14. Assuming that the program does not contain any sanitization function, the following path condition would be computed and reported to the auditor:

$$E(5) = \texttt{mode.equals("login")}$$
$$PC(4, 14) = E(5) = \texttt{mode.equals("login")}$$

IV. EVALUATION

A. Research Questions

To evaluate whether our approach achieves precision, soundness and scalability in providing assistance to security auditing, we aim to answer the following research questions:

1) Question 1. (Precision) How much reduction can be expected from security slicing in terms of source code to be inspected? Is the reduction practically significant?
2) Question 2. (Soundness) Do we extract all the statements that are relevant to auditing XMLi, XPathi, and SQLi vulnerabilities?
3) Question 3. (Scalability) Does the tool scale to realistic systems in terms of run-time performance?

B. Test Subjects

Table II shows the five Web applications/services that we used in our evaluation. It reports the sizes of the test subjects in terms of lines of code (LOC). The test subjects have an average size of 28 kLOC, and the largest one has 52 kLOC, which is fairly typical for that type of systems. The third column in Table II 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 XML, XPath, and SQL. Some sinks are very general and are exploitable in various ways (*e.g.* sinks that allow attackers to load arbitrary classes on server side). Due to their universality, we also considered them in our evaluation and their number is listed in column "others" in Table II.

WebGoat [38] is a deliberately in-secured 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* [39] is a blogging application that supports thousands of users and blogs. It also contains Web service APIs. *Pebble* [40] is also a blogging application that offers some Web service capabilities, *e.g.* an RPC API for blog post notifications. *Regain* [41] is a search engine that allows users to search for files over a Web front-end. *PubSubHubbub* (*PubSub*) [42] is the implementation of an open protocol for distributed publish/subscribe communication on the Internet. We selected *WebGoat*, Apache *Roller* and

Pebble since they are commonly used as benchmarks for security [15], [43], [44], [13], [45], and *Regain* since it is used in practice by *dm*, one of the biggest drug stores in Europe. These test subjects, together with our tool, can be obtained from our Website [25].

TABLE II. TEST SUBJECTS

	Java LOC	#Prog.	#Sources	#Sinks XML	#Sinks XPath	#Sinks SQL	#Sinks others	#Declassifiers XML	#Declassifiers XPath	#Declassifiers SQL
WebGoat 5.2	24,608	14	40	3	1	29	13	0	0	25
Roller 5.1.1	52,433	3	14	13	0	0	0	11	0	0
Pebble 2.6.4	36,592	3	6	7	0	0	0	3	0	0
Regain 2.1.0	23,182	1	1	1	0	0	0	3	0	0
PubSub 0.3	1,964	4	16	13	4	0	0	4	0	0

C. Experiment

1) Experimental design: To answer the first question, we compare the sizes of the slices produced by our security slicing method and state-of-the-art chopping (using *Joana*'s chopping functionality) in terms of the numbers of nodes and edges. That is, for each sink s, we compute a slice using our approach and a normal, unfiltered chop with the criterion (I, s). We use the *Wilcoxon signed-rank test* over the slice sizes across Web programs so as to determine whether the differences in sizes of the two types of slices are statistically significant. But what is more important is whether this difference is of practical significance, *i.e.*, does it save significant auditing effort?

To answer the second question, we checked all the security slices produced by our method against the source code to determine whether they omit any statement relevant to auditing XMLi, XPathi, and SQLi vulnerabilities.

To answer the last question, we evaluate our tool on realistic test subjects, such as Apache *Roller* and *Pebble* ($>$ 36 kLOC), and report its runtime performance results.

2) Results: As shown in Table II, we analyzed 25 Web programs from the five test subjects. For each Web program, an SDG was constructed. We computed normal chops and security slices from each SDG. The numbers of normal chops and security slices extracted from each Web program/SDG are given in the third column (Chops) of Table III. No chopping and security slicing was performed for 11 of the sinks in Table II because the tool determined that those sinks are not influenced by any input source. Overall, we computed 73 normal chops from 77 sources and 84 sinks, and 21 security slices from 25 Web programs.

The sizes (nodes and edges) of SDGs, normal chops, and security slices are reported in Table III. The last column in Table III 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 when, for example, all the paths in a slice are detected to be secured by the use of declassifiers.

To determine how much reduction security slicing achieves compared to normal chopping, we compute the relative size reduction of security slices compared to the unfiltered standard chop. The results (in percentage) are given in Table III (in brackets). We can observe that, both in terms of the number of nodes and edges, our security slices are significantly smaller than their normal counterparts. With mean and median reductions above 80% and 73%, respectively (shown in the last two rows of Table III), one can expect significant practical benefits. Not surprisingly, Wilcoxon signed-rank tests over 25 observations (#Prog.) show that the size reductions achieved with security slices are statistically significant at the 99% level.

The above comparison result reports the benefit of security slicing over chopping using a tool (*Joana*) that is not easy to configure and use for standard engineers. Furthermore, for situations where security auditors have no program chopping tool they know how to use or have access to, we can also check the percentage of the entire program code that needs to be audited with security slices. Comparing the security slice sizes and the SDG sizes in Table III, we can observe that on average security slicing would require the audit of only 0.8% of the code for all the sinks in a given Web program.

Since the unfiltered standard chops and the security slices are both based on the control-flow paths between sinks and sources, the size reduction of security slicing as compared to normal chopping in Table III is directly correlated to the reduction of manual effort required from security auditors for verifying vulnerable paths in the source code. Hence, these results answer our first research question by clearly suggesting that a significant reduction in code inspection can be expected.

Next, for all of the security slices from each test subject, we manually checked if those slices miss any information important for auditing their security vulnerability. Listing 3 shows the code corresponding to a security slice reported by *JoanAudit* which we use as an example to illustrate how slices were inspected. Since our approach relies on a predefined set of signatures of sinks, sources and declassifiers, an auditor knows where to start with the manual inspection. In our example, an auditor would first look at the sink function at line 5 and the parameters used in the sink. Then she would analyze the path condition at line 3 reported by the tool and track back the parameter used in the sink (`query` at line 5) to the source call (`getParameter()` at line 1). By doing so, an auditor would be able to determine the vulnerability condition of the sink. Following a similar process, we verified that all security slices provided sufficient information for security auditing, which addresses our second question.

```
1 String accountNumber = s.getParameter(ACCT_NUM, "101");
2 String query = "SELECT * FROM user_data WHERE userid = " +
        accountNumber;
3 if (accountNumber.toString().equals(answer_results.
      getString(1))) { /* ... */ } else {
4   Statement statement = conn.createStatement(/*...*/);
5   ResultSet results = statement.executeQuery(query);
6 }
```

Listing 3. Slice from the *WebGoat* (`BlindNumericSqlInjection`).

During our manual inspections of security slices, we also observed that filtering rules have different effects on different test subjects. For example, for Apache *Roller*, *Regain* and *PubSub*, the larger part of the slice size reduction is mainly due to the declassification filter and the filtering of known-good library files whereas for *WebGoat*, the majority of the slice size reduction is due to declassification, the filtering of irrelevant library files and automated fixing (four vulnerable sinks were fixed automatically).

Last, as shown in Table IV, we measured the time taken for computing each step in the generation of security slices

TABLE III. COMPARISON OF NORMAL CHOPPING AND SECURITY SLICING

	Program Name	Chops	SDG		Chopping		SecSlicing				
			Nodes	Edges	Nodes	Edges	Nodes	(%)	Edges	(%)	SecSlices to be audited
WebGoat		**43**	**160,573**	**923,709**	**7,980**	**3,975**	**1,533**	**(81)**	**764**	**(81)**	**14**
1	`BackDoors`	3	11,196	63,350	640	319	359	(44)	179	(44)	2
2	`BlindNumericSqlInjection`	2	9,573	52,262	173	86	124	(28)	62	(28)	1
3	`BlindScript`	5	21,558	140,134	1,534	765	0	(100)	0	(100)	0
4	`BlindStringSqlInjection`	2	9,616	52,580	173	86	124	(28)	62	(28)	1
5	`InsecureLogin`	3	11,998	68,257	1,450	725	0	(100)	0	(100)	0
6	`MultiLevelLogin1`	5	13,525	80,281	874	435	0	(100)	0	(100)	0
7	`MultiLevelLogin2`	5	12,546	71,773	1,290	641	0	(100)	0	(100)	0
8	`SqlAddData`	3	10,565	58,219	198	98	157	(21)	78	(20)	2
9	`SqlModifyData`	5	10,623	58,350	409	203	368	(10)	183	(10)	4
10	`SqlNumericInjection`	3	13,576	77,717	239	119	58	(76)	29	(76)	1
11	`SqlStringInjection`	3	12,155	69,502	437	217	113	(74)	56	(74)	1
12	`WsSAXInjection`	1	8,075	45,164	140	70	140	(0)	70	(0)	1
13	`WsSqlInjection`	2	9,191	49,232	333	166	0	(100)	0	(100)	0
14	`XPATHInjection`	1	6,376	36,888	90	45	90	(0)	45	(0)	1
Roller		**12**	**16,361**	**142,811**	**1,492**	**743**	**43**	**(97)**	**21**	**(97)**	**1**
15	`CommentDataServlet`	1	11,119	115,398	128	64	0	100	0	(100)	0
16	`AuthorizationServlet`	1	752	3,578	43	21	43	0	21	(0)	1
17	`OpenSearchServlet`	10	4,490	23,835	1,321	658	0	100	0	(100)	0
Pebble		**5**	**1,605**	**7,824**	**79**	**39**	**56**	**(29)**	**28**	**(28)**	**1**
18	`ImageCaptchaServlet`	1	829	4,033	56	28	56	0	28	0	1
19	`SecurityUtils`	3	236	1,128	18	9	0	100	0	100	0
20	`XmlRpcController`	1	540	2,663	5	2	0	100	0	100	0
Regain		**1**	**43,197**	**622,748**	**100**	**50**	**0**	**(100)**	**0**	**(100)**	**0**
21	`FileServlet`	1	43,197	622,748	100	50	0	(100)	0	(100)	0
PubSub		**12**	**37,567**	**307,390**	**1,209**	**603**	**530**	**(56)**	**264**	**(56)**	**5**
22	`Published.Discovery`	2	160	726	53	26	53	(0)	26	(0)	2
23	`PuSHhandler`	2	35,968	299,363	850	425	410	(52)	205	(52)	1
24	`PubSubHubbub.Discovery`	2	182	843	67	33	67	(0)	33	(0)	2
25	`Subscriber`	6	1,257	6,458	239	119	0	(100)	0	(100)	0
Total		73	259,303	2,004,482	10,860	5,410	2,162	(80)	1,077	(80)	21
Mean		3	10,372	80,179	434	216	86	(80)	43	(80)	1
Median		2	9,616	52,580	198	98	53	(73)	26	(73)	1

and normal chops. We observe that the SDG construction takes longer than other analysis steps. This is because SDG construction has a worst case time complexity of $O(N^3)$ with N being the size of the underlying dependence graph, whereas the rest of the analysis algorithms are much less complex (*e.g.* program chopping can be performed in linear time on the number of nodes in SDG). More importantly, we observe that *JoanAudit* took an average of 50*s* to analyze an entire test subject and required a maximum of 2 minutes to analyze the largest one. This shows that our tool can be practically run on Java Web systems that are in the same ball park size range as our test subjects, which is the case for many such systems. Given that security auditors typically have to manually audit large chunks of code in practice, our tool can be a great asset.

TABLE IV. RUNTIME PERFORMANCE (IN MILLISECONDS)

	SDG Generation	Source/Sink Identification	Chopping	Filtering	Total
WebGoat	124,301	504	12,266	694	137,765
Roller	23,815	56	763	69	24,703
Pebble	4,570	20	128	53	4,771
Regain	44,311	40	285	30	44,666
PubSub	39,213	85	965	153	40,416

D. Threats to Validity

Our empirical evaluation is subject to threats to validity. The results were obtained from five selected Web applications/services, and hence, they cannot necessarily be generalized to all Web services or Web applications. However, by choosing test subjects that vary in sizes and functionalities, and by picking realistic Java projects (with 28 kLOC on average) that are well-known benchmarks in the context of security, we minimized this threat.

Since our security slicing approach and tool are targeted towards Java Web systems, the approach may not produce the same results for Web systems based on other languages. However, since the fundamental principles of our approach are not programming language specific, they can be adapted to other languages such as C++ using C++ program slicing tools (*e.g.* CodeSurfer [46]).

V. RELATED WORK

Our work is most closely related to *static* taint analysis and program slicing approaches.

A. Taint analysis

Taint analysis approaches label data from input sources as tainted data and then, detect vulnerabilities if the tainted data flows into sinks without passing through any sanitization function (declassifier).

Almorsy *et al.* [47], Livshits and Lam [13], Pérez *et al.* [16], Tripp *et al.* [15], [17], and Huang *et al.* [18] developed taint analysis tools that support Java Web systems.

In general, there are three key differences between static taint analysis approaches and our security slicing approach. First, these analyses typically deal with only one security property (integrity) whereas our IFC analysis, by means of a security lattice, can deal with multiple security properties (in our case, integrity and confidentiality). Second, taint analysis does not perform control-dependency analysis. This information could be essential for correctly identifying vulnerabilities or auditing the correctness of input sanitization procedures since `if`-constructs are often used to check user inputs. For example, consider a simplified example from one of our test subject *WebGoat* below:

```
1 String employeeId = req.getParameter('id');
2 if(Integer.parseInt(employeeId) == EMPLOYEE_ID))
3   ResultSet results = stmt.executeQuery("SELECT * FROM
      employee WHERE userid =" + employeeId); //SQL sink
```

In the above example, a taint analysis approach would falsely report a vulnerability since there is a data-flow from the input source at line 1 to the sink at line 3, without considering the sanitization through a call to `parseInt()` at line 2 that does not 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 `parseInt()` declassifier and, thus, does not report a vulnerability. It is common for software engineers to use sanitization procedures as in the example above. Hence, if these procedures need to be audited, our approach would be more suitable than a taint analysis approach.

Jovanovic *et al.*'s taint analysis tool [14] reported five false positives due to such cases. Tripp *et al.* [17] reported 40% false positives on analyzing *WebGoat*. From our manual inspection of some of the *WebGoat* source code, missing control-dependency information seems to be responsible for at least five of their false positive cases. Likewise, in [48] it was reported that Livshits and Lam's taint analysis approach [13] yielded 20% false positives due to missing control-dependency information.

Last, our approach filters irrelevant and secure code whereas taint analysis approaches typically report all the data-flow traces without any form of filtering. Furthermore, our approach is dedicated to XMLi, XPathi and SQLi vulnerabilities. Among the current taint analysis approaches, to the best of our knowledge, only Pérez *et al.* [16] readily address XMLi and XPathi vulnerabilities. However, since Pérez *et al.*'s work is not evaluated in [16], it is difficult to verify the effectiveness of their tool. It is also possible to adapt the other approaches to support XMLi and XPathi 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, declassifiers with respect to XMLi, XPathi and SQLi and thus, can be used out-of-the-box.

B. *Program slicing*

Krinke [49] proposed 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 full Java: *Indus* [50] and *Joana* [24]. *Indus* is built on *Soot* [51], a Java bytecode analysis framework, and is less precise than *Joana* as it does not fully support interprocedural slicing [24]. As discussed in our approach section, *Joana* provides a sound and precise approach for computing slices and chops. As our approach and tool are built on *Joana*, we have the same advantages. However, *Joana* only generates slices for general purposes like checking information flow and debugging. By contrast, we additionally provide techniques for pruning statements in the slices produced by *Joana* and target the security auditing of vulnerabilities. *Joana* is, therefore, our baseline of comparison.

Yamaguchi *et al.* [22], [52] also proposed methods that assist security auditing for C/C++ programs by using machine learning to classify functions as vulnerable/non-vulnerable based on the absence/presence of sanitization [52], or by applying intraprocedural analysis on the code property graph (a combination of AST, CFG and PDG) of a program [22]. Besides the fact that we focus on Java instead of C/C++, our approach is based on interprocedural analysis which takes the call-return and parameter-passing mechanisms of the program into account.

The key difference between our approach and the above approaches is that they do not focus on minimizing the size of code extracted since 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 code extracted by filtering irrelevant or secure code so that security auditing is scalable and practical.

VI. Conclusion and Future Work

Injection vulnerabilities are among the most common and serious security threats to Web applications and services. A number of approaches have been developed to 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 present an approach, based on state-of-the-art program slicing, to assist the security auditing of common injection vulnerabilities, namely XMLi, XPathi, and SQLi. For every security-sensitive sink in the program, we extract 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 be complementary to current vulnerability detection approaches by helping the auditor identify false positives and negatives. A prototype tool that automates our approach was fully implemented and was used to generate 21 security slices from 25 Web programs. In comparison with conventional program slices, we observed that our security slices are 80% smaller on average while still retaining all the information relevant for verifying XMLi, XPathi, and SQLi vulnerabilities. We also made the tool and the test subjects available online so that researchers can validate and build on our results.

In the future, we intend to enhance our current approach by automating the vulnerability verification task. In particular, we aim to develop techniques that scale symbolic execution in order to make it applicable to the feasibility analysis of path conditions in conjunction with security threat conditions.

Acknowledgment

We would like to thank Jürgen Graf and Martin Mohr from Karlsruher 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 and FNR9132112).

REFERENCES

[1] OWASP, "OWASP Top 10," https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2013.

[2] N. Antunes and M. Vieira, "Soa-scanner: An integrated tool to detect vulnerabilities in service-based infrastructures," in *Services Computing (SCC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 280–287.

[3] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan, "Automated testing for sql injection vulnerabilities: An input mutation approach," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610403

[4] N. Laranjeiro, M. Vieira, and H. Madeira, "A technique for deploying robust web services," *Services Computing, IEEE Transactions on*, vol. 7, no. 1, pp. 68–81, Jan 2014.

[5] J. Thomé, A. Gorla, and A. Zeller, "Search-based security testing of web applications," in *Proceedings of the 7th International Workshop on Search-Based Software Testing*, ser. SBST 2014. New York, NY, USA: ACM, 2014, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/2593833.2593835

[6] C. Mainka, M. Jensen, L. L. Iacono, and J. Schwenk, "Making xml signatures immune to xml signature wrapping attacks," in *Cloud Computing and Services Science*. Springer, 2013, pp. 151–167.

[7] T. M. Rosa, A. O. Santin, and A. Malucelli, "Mitigating xml injection 0-day attacks through strategy-based detection systems," *Security & Privacy, IEEE*, vol. 11, no. 4, pp. 46–53, 2013.

[8] A. Razzaq, K. Latif, H. F. Ahmad, A. Hur, Z. Anwar, and P. C. Bloodsworth, "Semantic security against web application attacks," *Information Sciences*, vol. 254, pp. 19–38, 2014.

[9] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 372–382. [Online]. Available: http://doi.acm.org/10.1145/1111037.1111070

[10] W. Halfond, A. Orso, and P. Manolios, "Wasp: Protecting web applications using positive tainting and syntax-aware evaluation," *Software Engineering, IEEE Transactions on*, vol. 34, no. 1, pp. 65–81, Jan 2008.

[11] H. Shahriar and M. Zulkernine, "Information-theoretic detection of sql injection attacks," in *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, Oct 2012, pp. 40–47.

[12] Z. Tao, "Detection and service security mechanism of xml injection attacks," in *Information Computing and Applications*. Springer, 2013, pp. 67–75.

[13] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251398.1251416

[14] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, May 2006, pp. 6 pp.–263.

[15] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 87–97. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542486

[16] P. M. Pérez, J. Filipiak, and J. M. Sierra, "LAPSE+ static analysis security software: Vulnerabilities detection in java ee applications," in *Future Information Technology*. Springer, 2011, pp. 148–156.

[17] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 210–225. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37057-1_15

[18] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *Fundamental Approaches to Software Engineering*, ser. Lecture Notes in Computer Science, S. Gnesi and A. Rensink, Eds. Springer Berlin Heidelberg, 2014, vol. 8411, pp. 140–154. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54804-8_10

[19] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst, "Automatic creation of sql injection and cross-site scripting attacks," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, May 2009, pp. 199–209.

[20] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Software Engineering (ICSE), 2013 35th International Conference on*, May 2013, pp. 652–661.

[21] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 1, pp. 3:1–3:42, Oct. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629536

[22] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 590–604. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.44

[23] M. Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81. Piscataway, NJ, USA: IEEE Press, 1981, pp. 439–449. [Online]. Available: http://dl.acm.org/citation.cfm?id=800078.802557

[24] C. Hammer, "Information flow control for java - a comprehensive approach based on path conditions in dependence graphs," Ph.D. dissertation, Universität Karlsruhe (TH), Fak. f. Informatik, Jul. 2009, iSBN 978-3-86644-398-3. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012049

[25] J. Thomé, "JoanAudit: a security slicing tool," http://wwwen.uni.lu/snt/research/software_verification_and_validation_lab/tools_from_svv_lab, 2015.

[26] OWASP, "OWASP ESAPI," https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API, 2015.

[27] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[28] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, no. 5, pp. 177–184, Apr. 1984. [Online]. Available: http://doi.acm.org/10.1145/390011.808263

[29] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[30] J.-F. Bergeretti and B. A. Carré, "Information-flow and data-flow analysis of while-programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 1, pp. 37–61, 1985.

[31] D. Jackson and E. J. Rollins, "Chopping: A generalization of slicing," DTIC Document, Tech. Rep., 1994.

[32] T. Reps and G. Rosay, "Precise interprocedural chopping," in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 4. ACM, 1995, pp. 41–52.

[33] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification and qualified robustness," *J. Comput. Secur.*, vol. 14, no. 2, pp. 157–196, Apr. 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1150577.1150580

[34] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, no. 1, pp. 5–19, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2002.806121

[35] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. IEEE, 2005, pp. 255–269.

[36] Apache, "StringEscapeUtils," https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/StringEscapeUtils.html, 2015.

[37] G. Snelting, "Combining slicing and constraint solving for validation of measurement software," in *Static Analysis*, ser. Lecture Notes in Computer Science, R. Cousot and D. Schmidt, Eds. Springer Berlin Heidelberg, 1996, vol. 1145, pp. 332–348. [Online]. Available: http://dx.doi.org/10.1007/3-540-61739-6_51

[38] OWASP, "OWASP WebGoat project," https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project, 2015.

[39] Apache, "Apache Roller blogging application," http://roller.apache.org/, 2015.

[40] Pebble, "A lightweight, open source, java ee blogging tool," http://pebble.sourceforge.net/, 2015.

[41] Regain, "Regain search engine," http://regain.sourceforge.net/, 2015.

[42] PubSubHubbub, "A simple, open, webhook based pubsub protocol & open source reference implementation," https://code.google.com/p/pubsubhubbub/, 2015.

[43] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "Aside: Ide support for web application security," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 267–276. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076770

[44] Y. Liu and A. Milanova, "Practical static analysis for inference of security-related program properties," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009, pp. 50–59.

[45] A. Møller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," *Transactions on Software Engineering and Methodology*, vol. 23, no. 4, August 2014, earlier version in Proc. 34th International Conference on Software Engineering (ICSE) 2012.

[46] T. Teitelbaum, "Codesurfer," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 1, pp. 99–, Jan. 2000. [Online]. Available: http://doi.acm.org.proxy.bnl.lu/10.1145/340855.341076

[47] M. Almorsy, J. Grundy, and A. S. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 100–109.

[48] L. K. Shar and H. B. K. Tan, "Auditing the XSS defence features implemented in web application programs," *IET Software*, vol. 6, no. 4, pp. 377–390, 2012.

[49] J. Krinke, "Slicing, Chopping, and Path Conditions with Barriers," *Software Quality Journal*, vol. 12, no. 4, pp. 339–360, Dec. 2004. [Online]. Available: http://link.springer.com/10.1023/B:SQJO.0000039792.93414.a5

[50] G. Jayaraman, V. P. Ranganath, and J. Hatcliff, "Kaveri: Delivering the indus java program slicer to eclipse," in *Fundamental Approaches to Software Engineering*. Springer, 2005, pp. 269–272.

[51] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[52] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 499–510. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516665

APPENDIX A. IMPLEMENTATION

We implemented our approach as a Java application called *JoanAudit*, which is publicly available [25]. The tool makes use of *Joana* [24] which is based on IBM's WALA framework (http://wala.sourceforge.net). The API is already capable of generating SDGs from Java bytecode and performing IFC-based slicing. However, as *Joana* is not specifically designed for our security analysis purposes, the additionally required functionalities for security slicing are implemented and incorporated in *JoanAudit*, as explained below.

Fig. 4 illustrates the architecture of the tool. Given a Java Web program (*i.e.* JSP or Java servlet), *JoanAudit* performs the six analysis steps discussed in Section III.

As input, *JoanAudit* requires the bytecode of the program to be analyzed. The tool contains two XML configuration files– one specifies a rich set of the Java bytecode signatures of input sources, sinks and declassifiers, and the other specifies a configuration for our security lattice explained in Section III-D. Based on the first configuration file, the tool identifies input sources, sinks and declassifiers and annotates them in the SDG automatically. Using the annotations, *JoanAudit* generates a program chop for each sink. Based on the security lattice configuration file, it performs IFC analysis on the program chops and prunes the secure paths. It then extracts path conditions from the remaining paths in the chops to help guide the security auditing.

As output, the tool generates a report that leads the security auditor to potentially vulnerable parts of the program. A sample report is shown in Listing 4. The report contains potentially vulnerable paths (sequences of line numbers) and highlights the control-flow, data-depenencies, control-dependencies, and path conditions along these paths. The scopes (the classes to which the line numbers refer to) are parenthesized with squared brackets. The tool runs standalone and can be executed on the command line. Specifying sanitization procedures (declassifiers) is also straightforward by just adding the corresponding bytecode method signature to the `config.xml` file.

For the automated fixing filter, we implemented a rudimentary symbolic execution engine that supports simple string operations. Whenever there is a direct flow from a source to a sink, we compute the string s used in the sink by symbolically executing all those operations on the path that have an impact on s. The resulting string contains constant and variable parts (symbolic input variables) that represent the input source values (*e.g.* `/users/user[@nick='v1' and @password='v2']`, where `v1` and `v2` are symbolic input variables). Afterwards, we apply the patterns from Table I to determine the appropriate sanitization functions.

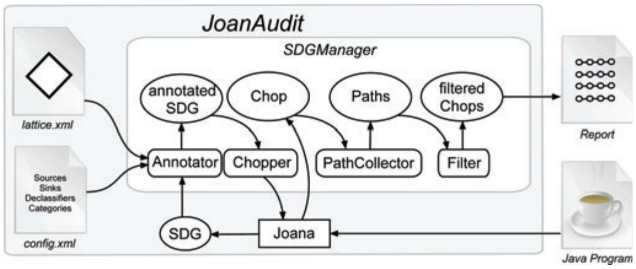

Fig. 4. Architecture of *JoanAudit*.

```
For sink xpath injection (snk_xi)(145):
* Control Flow:[org/owasp/webgoat/lessons/XPATHInjection.
      java]
      131->132->138->139->140->141->142->143->144->145
* Data Flow:([org/owasp/webgoat/lessons/XPATHInjection.
      java] 143->145)(141->145)(144->143)(143->144)
      (131->143)(139->142)(140->141)(138->139)(131->132)
* Control Dependencies:([org/owasp/webgoat/lessons/
      XPATHInjection.java] 143->145)(144->143)(143->144)
      (142->143)(141->142)(140->141)(139->140)(138->139)
      (132->138)
* Conditions :[org/owasp/webgoat/lessons/XPATHInjection.
      java] 132
```

Listing 4. Sample report generated from *WebGoat*.