

# An Integrated Approach for Effective Injection Vulnerability Analysis of Web Applications through Security Slicing and Hybrid Constraint Solving

Julian Thomé, Lwin Khin Shar, Domenico Bianculli, *Member, IEEE*, Lionel Briand, *Fellow, IEEE*

## Abstract—

Malicious users can attack Web applications by exploiting injection vulnerabilities in the source code. This work addresses the challenge of detecting injection vulnerabilities in the server-side code of Java Web applications in a scalable and effective way. We propose an integrated approach that seamlessly combines security slicing with hybrid constraint solving; the latter orchestrates automata-based solving with meta-heuristic search. We use static analysis to extract minimal program slices relevant to security from Web programs and to generate attack conditions. We then apply hybrid constraint solving to determine the satisfiability of attack conditions and thus detect vulnerabilities.

The experimental results, using a benchmark comprising a set of diverse and representative Web applications/services as well as security benchmark applications, show that our approach (implemented in the *JOACO* tool) is significantly more effective at detecting injection vulnerabilities than state-of-the-art approaches, achieving 98% recall, without producing any false alarm. We also compared the constraint solving module of our approach with state-of-the-art constraint solvers, using six different benchmark suites; our approach correctly solved the highest number of constraints (665 out of 672), without producing any incorrect result, and was the one with the least number of time-out/failing cases. In both scenarios, the execution time was practically acceptable, given the offline nature of vulnerability detection.

**Index Terms**—Vulnerability detection, constraint solving, static analysis, search-based software engineering



## 1 INTRODUCTION

SYMBOLIC execution and constraint solving represent a state-of-the-art approach used in security analysis to identify vulnerabilities in software systems. Symbolic execution executes a program with symbolic inputs and at the end generates a set of *path conditions*. Each of them corresponds to a constraint imposed on the symbolic inputs to follow a certain program path, i.e., a constraint characterizing a possible execution. By solving these constraints with a constraint solver, one can determine which concrete inputs can cause a certain program path to be executed.

In the context of security analysis this approach is used [1], [2], [3], [4] to detect *injection vulnerabilities*, i.e., program locations in which certain malicious inputs can alter the intended program behavior. Roughly speaking, this approach consists of solving the constraints obtained by

conjoining the path conditions (generated by the symbolic execution) with attack specifications provided by security experts. The main strength of this approach is that vulnerability detection yields a limited number of false positives, since the concrete inputs determined with constraint solving prove the existence of vulnerabilities. However, the effectiveness and precision of this approach are challenged by two main problems that affect symbolic execution and constraint solving [5]: 1) path explosion and 2) solving complex constraints (e.g., constraints involving regular expressions or containing string/mixed or integer operations). Notice that while these problems are independent from the context in which symbolic execution and constraint solving are applied, the solutions to mitigate them can be tailored for a specific context. Nevertheless, existing proposals [1], [2], [3], [4] in the context of vulnerability analysis have not fully addressed these problems.

The path explosion problem is triggered by the huge number of feasible program paths that symbolic execution has to explore in large programs. To mitigate this problem in the context of vulnerability analysis, in previous work [6] we proposed an approach to extracting security slices from Java programs. A security slice contains a concise and minimal sequence of program statements that affect a given security-sensitive program location (sink), such as an SQL query statement. Symbolic analysis can then be performed on security slices instead of the whole program; in this way

- J. Thomé, D. Bianculli, and L. Briand are with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg, Luxembourg.  
E-mail: [julian.thome@uni.lu](mailto:julian.thome@uni.lu), [domenico.bianculli@uni.lu](mailto:domenico.bianculli@uni.lu), [lionel.briand@uni.lu](mailto:lionel.briand@uni.lu)
- L.K.Shar is with the School of Computer Science and Engineering of the Nanyang Technological University, Singapore. He was affiliated with the Interdisciplinary Centre for Security, Reliability, and Trust (SnT) of the University of Luxembourg when this work was done.  
E-mail: [lkshar@ntu.edu.sg](mailto:lkshar@ntu.edu.sg)

Manuscript received Month XX, 201X; revised Month XX, 201X.

path conditions are analyzed only with respect to the paths leading to sinks instead of every path in the program. Since, according to our experience [6], the number of sinks in a program is low<sup>1</sup> and security slices are much smaller (approx. 1%) than the program containing them, this approach can effectively mitigate the path explosion problem.

The problems related to solving complex constraints are mainly due to the support for strings and their operations. In general, solving constraints that contain string operations requires to analyze the implementation of these operations, unless they can be treated as primitive functions in the constraint solver. However, there are typically hundreds of string operations in a given programming language that cannot be solved because their semantics is not known to the constraint solver (e.g., `java.lang.Object.hashCode()` and `java.lang.String.format(...)`); we denote these operations as *unsupported operations*. Existing approaches support only a limited number of string operations—such as concatenation, assignment, and equality—as primitive functions. More complex operations have to be analyzed and transformed into an equivalent set of basic constraints containing primitive functions. This task is often not trivial and requires proficiency in the input language of the solver. A constraint solver that supports a limited set of operations can fail in solving constraints that contain unsupported operations, resulting in missed vulnerabilities. To partially mitigate this problem, in previous work [7] we proposed a fallback mechanism to extend existing string constraint solvers for dealing with constraints with unsupported string operations. This mechanism, implemented in the *ACO-Solver* tool, used an off-the-shelf automata-based string constraint solver combined with a search-driven constraint solving procedure based on the Ant Colony Optimization meta-heuristic [8].

The goal of the work presented in this paper is to *provide a scalable approach, based on symbolic execution and constraint solving, to effectively find injection vulnerabilities in source code, which generates no or few false alarms, minimizes false negatives, and overcomes the path explosion problem and the one of solving complex constraints*.

We propose a new analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving.

We leverage our previous work on security slicing [6] to mitigate the path explosion problem, by generating during the symbolic execution only the constraints that characterize the security slices of the program under analysis. This step allows us to identify paths and statements in the program where vulnerabilities can be exploited; this helps make the remainder of the approach scalable. The generated constraints are then preprocessed in order to simplify the following step.

The next step uses a hybrid approach that orchestrates a constraint solving procedure for string/mixed and integer constraints with a search-based constraint solving procedure. The idea behind this hybrid solving strategy is to solve a constraint through a two-stage process:

- 1) First, our solving procedure solves all the constraints with supported operations, by leveraging automata-

based solving for solving string and mixed constraints, and linear interval arithmetic for solving integer constraints. In both cases, constraint solving rules are expressed using *recipes* that model the semantics of the operations. In particular, we provide recipes for many string/mixed operations, including 16 input sanitization operations from widely used security libraries [9], [10], and commonly used integer operations. In this way, the constraints involving supported operations can be efficiently solved, without transforming them into a set of primitive functions.

- 2) In the second stage, we use a search-driven solving procedure, which is based on and extends our previous work [7]. This procedure leverages the Ant Colony Optimization meta-heuristic to solve the remaining constraints which contain unsupported operations. The search space of this procedure is represented by the input domains as determined in the first stage; the search is driven by different fitness functions, depending on the type of the constraints.

The solver in the first stage is used to reduce (possibly in a significant way) the search space, i.e., the domains of the string and integer variables, for the search-driven solving procedure; hence, it makes the search in the second stage more scalable and effective.

Our approach constitutes a *targeted security analysis* method, since its target (i.e., the type of vulnerabilities to analyze) can be specified by providing the corresponding *threat models*, i.e., generalized attack specifications (also called attack patterns) associated with different types of sinks. The approach analyzes the satisfiability of a threat model in conjunction with the path condition that leads to a given sink. In this sense, our approach can be considered *general*, since it can detect any type of vulnerability whose threat model can be described using regular expressions. To show this generality, in this paper we provide the threat models for five common types of vulnerabilities: cross-site scripting (XSS), SQL injection (SQLi), XPath injection (XPathi), XML injection (XMLi), LDAP injection—LDAPi). Moreover, the approach is also *language-independent*, since the modeling of string/mixed and integer operations proposed in this paper—although provided in the context of the Java language—can be easily ported to other languages.

Our integrated technique achieves high effectiveness in detecting vulnerabilities. In particular, we assessed the vulnerability detection capability of our tool (*JOACO*) by comparing it, using a benchmark comprising a set of diverse and representative Web applications/services as well as security benchmark applications, with *SFlow* [11] and *LAPSE+* [12]—two state-of-the-art security analysis tools for Java—and also with our previous work *JoanAudit+CVC4+ACO-Solver*, i.e., the combination of our security slicing tool *JoanAudit* [6] with *CVC4+ACO-Solver* [7]. *CVC4+ACO-Solver* is the combination of the *CVC4* solver with *ACO-Solver*, which was shown [7] to be the best performing constraint solver (in the context of vulnerability detection) when compared with *CVC4*, *Z3-str2*, and *Z3-str2+ACO-Solver*.

*JOACO* achieved a high recall of 98% and 100% precision, detecting 93 vulnerabilities (out of which 8 were previously unknown), missing only 2 vulnerabilities, without producing any false alarm. *JOACO* performed much bet-

1. Our experiments show that, on average, there are only 3 sinks in a Web program, related to the type of vulnerabilities we consider.

ter than state-of-the-art vulnerability detection tools, yielding a higher recall (ranging between +70pp and +73pp, with pp=percentage points) and precision (ranging between +16pp and +27pp), with no failing cases. This high effectiveness in vulnerability detection comes at the cost of a higher execution time, which is however practically acceptable. Compared with our previous work *JoanAudit+CVC4+ACO-Solver*, *JOACO* detected more vulnerabilities, had much less time-out cases, and was faster.

We also assessed the string constraint solving capability of our approach, since string constraint solving is widely recognized by the research community as a means to enable vulnerability detection. More specifically, we compared the constraint solving capabilities of *JOACO* when used in the stand-alone solver mode (dubbed *JOACO-CS*) with three state-of-the-art constraint solvers: *CVC4* [13], *Z3* [14], and our previous work *CVC4+ACO-Solver* [7]. We used a benchmark collection comprising 672 constraints, obtained from six different sources.

*JOACO-CS* performed similarly to or better (+7%–+14% more correctly solved cases) than state-of-the-art string constraint solvers, including our previous work, depending on the benchmark considered. It correctly solved the highest number of constraints, without producing any incorrect result, and was the one with the least number of time-out/failing cases. In particular, *JOACO-CS* fared best with constraints derived from Web applications containing actual injection vulnerabilities, which are the target of this work. In terms of execution time, *JOACO-CS* was  $1.7\times$  slower than the most effective, state-of-the-art constraint solver (*CVC4*). However, since *JOACO-CS* can solve more cases and constraint solving is typically an offline activity, with no stringent time requirements, we consider this slowdown as practically acceptable.

To sum up, the specific contributions of this paper are:

- An integrated analysis technique for injection vulnerabilities, which leverages the synergistic combination of security slicing with hybrid constraint solving. This technique is general and language-independent.
- The application of this technique to detect the five most common types of injection vulnerabilities (XSS, SQLi, XMLi, XPathi, LDAPi) in the context of Java applications.
- The implementation of the proposed technique in a fully functional tool called *JOACO*, publicly available [15].
- An extensive empirical evaluation on automated vulnerability detection, which assesses two separate and complementary aspects: 1) vulnerability detection effectiveness, precision, and run-time performance, using a benchmark comprising a set of diverse Web applications/services and security benchmark applications; 2) the effectiveness and run-time performance of the constraint solving part of our overall solution, using a benchmark collection comprising 672 constraints.

As a separate contribution, we also make available the artifacts used in the evaluation.

This paper can be considered an extension of our previous work [7]; the major differences are:

**Hybrid constraint solving technique.** The hybrid constraint solving technique described in this paper and implemented in *JOACO* (and *JOACO-CS*) is a revised and im-

proved version of the one described in [7] (and implemented in *ACO-Solver*) along the following lines:

- *Constraint pre-processing.* *JOACO* includes a pre-processing step that applies: a) *derived constraint generation*, which adds additional constraints to reduce the input domain and solve the constraints more efficiently; b) *constraint refinement*, to simplify the constraint network, detect trivially inconsistent constraints, and avoid unnecessary and expensive constraint solving.
- *No dependency on an external solver.* *ACO-Solver*, since it implemented a fallback mechanism, first invoked an external solver (i.e., the solver for which the fallback mechanism was provided) to attempt to solve the input constraint. When this invocation failed (e.g., because the external solver could not solve a constraint with an unsupported operation), *ACO-Solver* had to restart the constraint solving from scratch, since it could not reuse or benefit from any intermediate result determined by the external solver before the failure. In this work, *JOACO* does not depend on any external solver, since it orchestrates the two-stage process sketched above.
- *Built-in support for a larger set of string operations.* *ACO-Solver* relied on the *Sushi* [3] constraint solver to compute solution automata for string constraints, before calling the search-based solving procedure. However, *Sushi* supports very few string operations (concat, contains, equals, trim, substring, replace, replaceAll, and matches). In this paper, we built our automata-based and interval constraint solver on top of *Sushi* and extended it with support for 38 new operations, including 16 input sanitization operations from the Apache Commons Lang 3 [9] and OWASP [10] standard security libraries. By supporting more operations in our built-in automata-based and interval constraint solver, we are able to minimize the number of invocations to the search-based constraint solving procedure.
- *Unified treatment of integer and string constraints.* In [7] we did not describe how numeric constraints were solved and focused only on string constraints. In this paper we provide a unified treatment for string and integer constraints, by converting integer ranges into their automaton representation; in this way, both types of constraints can be handled by our search-based solving procedure (which requires every variable domain to be represented in the form of a solution automaton).

**Empirical evaluation.** This paper includes a much larger and diverse empirical evaluation. First, we assess the vulnerability detection capability of *JOACO* when analyzing the source code of Web applications and compare it with state-of-the-art vulnerability detection tools for Java Web applications; this analysis is completely new. Second, we assess the constraint solving capability of our approach by running an experimental study which extends the one from [7] by comparing with different string constraint solvers and by using a larger benchmark collection, which includes five additional benchmarks, used in previous studies, as well as an extended version of our home-grown benchmark.

The rest of the paper is organized as follows. Section 2 provides some background on the concepts and techniques used in the paper. Section 3 discusses the motivations for this work. Sections 4–7 illustrate our approach. Section 8

outlines the implementation of our *JOACO* tool. Section 9 presents the evaluation of our approach; section 10 discusses its limitations. Section 11 surveys related work. Section 12 concludes the paper and gives directions for future work.

## 2 BACKGROUND

In this section we present several background concepts used in the rest of the paper. We first provide a short introduction to constraints and constraint networks (§ 2.1) and the Ant Colony Optimization meta-heuristic (§ 2.2). Afterwards, we define the application scope of our work, by giving an overview of injection vulnerabilities (§ 2.3) and of the associated threat models (§ 2.4).

### 2.1 Constraints and Constraint Networks

The following definitions are based on the ones presented in the constraint solving literature [16], [17].

Let  $Y = y_1, \dots, y_k, k > 0$  be a finite sequence of variables and  $D_1, \dots, D_k$  a sequence of domains, with each variable  $y_i$  ranging over the respective domain  $\text{dom}(y_i) = D_i$ . A *constraint*  $c$  over  $Y$  is a relation over  $Y$ , i.e.,  $c \subseteq D_1 \times \dots \times D_k$ ;  $Y$  is also called the *scope* of the constraint and  $k$  is its arity. Informally, a constraint  $c$  on some variables is a subset of the cartesian product over the variable domains that contains the combination of values that satisfy  $c$ .

A *constraint network*  $\mathcal{R}$  is a triple  $(X, D, C)$ , where  $X$  is a finite sequence of variables  $x_1, \dots, x_n$ , each associated with a domain  $D_1, \dots, D_n$ , and  $C = \{c_1, \dots, c_t\}$  is a set of constraints; the scope of each constraint  $c_i$ , denoted by  $S_i$ , is a subsequence of  $X$ .

A constraint network  $\mathcal{R} = (X, D, C)$  can be represented as a *hypergraph*  $H = (V, S)$  where the set of nodes  $V$  corresponds to the set of variables  $X$  of the network, and  $S = S_1, \dots, S_t$  is the set of hyperedges that group variables belonging to the same scope.

The *union* ( $\cup$ ) and *intersection* ( $\cap$ ) operators for constraint networks return another constraint network and are defined as follows: given two constraint networks  $\mathcal{R}_1 = (X_1, D_1, C_1)$  and  $\mathcal{R}_2 = (X_2, D_2, C_2)$ ,  $\mathcal{R}_3 = \mathcal{R}_1 \oplus \mathcal{R}_2$ , where  $\oplus \in \{\cup, \cap\}$ ,  $\mathcal{R}_3 = (X_3, D_3, C_3)$  with  $X_3 = X_1 \oplus X_2$ ,  $D_3 = D_1 \oplus D_2$ , and  $C_3 = C_1 \oplus C_2$ .

$\mathcal{R}_1$  is a *subgraph* of  $\mathcal{R}_2$ , denoted by  $\mathcal{R}_1 \subseteq \mathcal{R}_2$ , if and only if  $\mathcal{R}_1 \cap \mathcal{R}_2 = \mathcal{R}_1$ .

An *instantiation* of a constraint network  $\mathcal{R} = (X, D, C)$  assigns to each variable  $x_i$  of  $X$  a value from  $\text{dom}(x_i)$ . A *solution*  $\sigma$  of a constraint network  $\mathcal{R} = (X, D, C)$  is an instantiation of  $\mathcal{R}$  that satisfies all the constraints in  $C$ ; we say that  $\sigma$  satisfies  $\mathcal{R}$  and that  $\sigma$  is a model of  $\mathcal{R}$ . The tuple of values assigned by  $\sigma$  to a set of variables  $I$  with  $I \subseteq X$  is denoted by  $\mathcal{S}(\sigma, I)$ . When  $\mathcal{R}$  possesses at least one model,  $\mathcal{R}$  is said to be *satisfiable*; otherwise,  $\mathcal{R}$  is said to be *unsatisfiable*.

Two constraint networks  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are *equisatisfiable* (written as  $\mathcal{R}_1 \equiv_{\text{sat}} \mathcal{R}_2$ ) if and only if  $\mathcal{R}_1$  is satisfiable whenever  $\mathcal{R}_2$  is satisfiable. Equisatisfiability is a reflexive, transitive, and symmetric relation.

Given a set of variables  $I$ , and two constraint networks  $\mathcal{R}_1 = (X_1, D_1, C_1)$  and  $\mathcal{R}_2 = (X_2, D_2, C_2)$ ,  $\mathcal{R}_1$  *subsumes*  $\mathcal{R}_2$  modulo  $I$  (written as  $\mathcal{R}_1 \models_I \mathcal{R}_2$ ) if for all models  $\sigma_1$  of  $\mathcal{R}_1$  there exists a model  $\sigma_2$  of  $\mathcal{R}_2$  such that  $\mathcal{S}(\sigma_1, I \cap X_1 \cap X_2) \subseteq \mathcal{S}(\sigma_2, I \cap X_2)$ .

Given two constraint networks  $\mathcal{R}_1$  and  $\mathcal{R}_2$  and a set of variables  $I$ ,  $\mathcal{R}_1$  is *equivalent to  $\mathcal{R}_2$  modulo  $I$*  (written as  $\mathcal{R}_1 \equiv_I \mathcal{R}_2$ ) iff  $\mathcal{R}_1 \models_I \mathcal{R}_2$  and  $\mathcal{R}_2 \models_I \mathcal{R}_1$ .

Two constraint networks  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are *equivalent* (denoted by  $\mathcal{R}_1 \equiv \mathcal{R}_2$ ) if they are defined on the same set of variables and express the same set of solutions. Two equivalent constraint networks are also equisatisfiable.

### 2.2 Ant Colony Optimization

Ant Colony Optimization (ACO) [8] is a widely used meta-heuristic search techniques for solving combinatorial optimization problems. It is inspired by the observation of the behavior of real ants searching for food. Real ants start seeking food randomly; when they find a source of food, they leave a chemical substance (called *pheromone*) along the path that goes from the food source back to the colony. Other ants can detect the presence of this substance and are likely to follow the same path. This path, populated by many ants, is called *pheromone trail* and serves as a guidance (e.g., positive feedback) for the other ants. In ACO, these observations are translated into the world of *artificial ants*, which can cooperate to find a good solution to a given optimization problem. The optimization problem is translated into the problem of finding the best path on a weighted graph. *Artificial pheromone trails* are numeric parameters that characterize the graph components (i.e., nodes and edges); they encode the “history” in approaching the problem (and finding its solutions) by the whole ant colony. ACO algorithms also implement a mechanism, inspired by real pheromone evaporation, to modify the pheromone information over time so that ants can forget the (search) history and start exploring new search directions. The artificial ants build their solutions by moving step-by-step along the graph; at each step they make a stochastic decision based on the pheromone trail.

### 2.3 Injection vulnerabilities

*Injection vulnerabilities* are program locations in which certain malicious input can be “injected” into the program to alter its intended behavior or the one of another system. An injection may occur when the user input is passed through the program to an interpreter or to an external program (e.g., a shell interpreter, a database engine) and the input data contain malicious commands or command modifiers (e.g., a shell script, an additional constraint of an SQL query). An injection vulnerability arises when the input is not properly validated or sanitized in correspondence of a sink.

Injection vulnerabilities can cause serious damage to a system and its users. For example, an attacker could compromise the systems underlying the application or gain access to a database containing sensitive information. The “OWASP (Open Web Application Security Project) Top 10 2013” report [18] shows that injection vulnerabilities are the most common application security risk for Web applications.

There are several types of injection vulnerabilities. In this paper we focus on the following five types, for which we give a short overview and an example based on the CWE (Common Weakness Enumeration) dictionary [19].

### 2.3.1 Cross-site scripting (XSS)

It is an attack technique that injects malicious scripts into a trusted Web application. It can be accomplished by inserting untrusted, browser-executable data (e.g., JavaScript code, HTML tags) through a web request. When these data are used to dynamically generate a page requested by an end-user, the malicious script (injected through the untrusted data) will be executed in the user's browser, which is misled to consider the script as coming from a trusted source (and thus safe to execute). This untrusted code, once executed in the browser, may access and transmit to the attacker confidential information such as the user's session cookies (possibly leading to hijacking of the user's session), or may alter the presentation of the content (possibly leading to phishing attacks).

As an example, consider the code snippet below:

```
1 String name = req.getParameter("user");
2 res.println("<div>Welcome_<"+name+"!</div>");
```

It dynamically generates an HTML element `div` based on the input received (through a Web request) and stored in the `name` variable. An attacker could perform an XSS attack by providing as input the string `<script language="JavaScript">alert('XSS');</script>`, which contains a snippet of JavaScript code. This code will be executed by the browser when it interprets the HTML code provided in the HTTP response. While in this case the injected code just displays a pop-up dialogue, in principle it could have much more harmful effects, like the ones mentioned above.

### 2.3.2 SQL injection (SQLi)

It is an attack technique that injects an SQL query in the input of a program, in order to read/write/admin a relational database by affecting the execution of predefined SQL statements. It can be accomplished by placing a meta-character into the input string, which acts as a modifier of the original SQL statement and allows the attacker to alter its behavior.

As an example, consider the code snippet below:

```
1 String userid = req.getParameter("userid");
2 String query = "SELECT_*_FROM_users_WHERE_user='"+
3   + userid + "'";
4 Statement st = conn.createStatement();
5 ResultSet rs = st.executeQuery(query);
```

It dynamically builds a query string based on the input received (through a Web request) on the first line, by concatenating a constant string with the user input string. If a malicious user provides as input the string `name' OR 1=1 --`, the resulting query string will be `SELECT * FROM users WHERE user='name' OR 1=1 --`. Notice that the malicious input string is built to correctly enclose (with a single quote character) the first condition of the WHERE clause and to add a second condition `OR 1=1`. The latter represents a tautology and causes the WHERE clause to always evaluate to true. The query becomes logically equivalent to `SELECT * FROM users`, allowing the attacker to access all the contents of the table `users` in the database.

### 2.3.3 XML injection (XMLi)

It is a technique that allows attackers to change the structure or the contents of an XML document before it is processed

by the program. It can be accomplished by placing reserved words or meta-characters into the input string. Such an attack may yield various consequences, such as invalidating the XML document, injecting malicious content in the document, or forcing the XML parser to access external entities.

Consider, for example, the following XML document named `students.xml`:

```
<students>
  <student>
    <sid>1</sid>
    <email>wd@svv.lu</email>
    <uid>wd003</uid>
    <pwd>300wd</pwd>
  </student>
  <student>
    <sid>2</sid>
    <email>abf@svv.lu</email>
    <uid>abf004</uid>
    <pwd>400abf</pwd>
  </student>
</students>
```

and the following Java snippet that updates the email address of a student:

```
1 File db = new File("students.xml");
2 Document doc = DocumentBuilderFactory.newInstance()
3   .newDocumentBuilder().parse(db);
4 String uid = req.getParameter("uid");
5 String pwd = req.getParameter("pwd");
6 String emailnew = req.getParameter("emailnew");
7 //code to find the right <student> element and
8 //its children
9 if (student-uid.equals(uid) &&
10   student-pwd.equals(pwd)) {
11   if ("email".equals(node.getNodeName())) {
12     node.setTextContent(emailnew);
13   }
14 }
```

A malicious user could invalidate the XML document by entering an email address that contains a meta-character, such as an angular parenthesis like `<`. For example, if the attacker enters the email `wd@svv.lu<`, the corresponding element updated by the snippet above will look like `<email>wd@svv.lu<</email>` and will invalidate the document, possibly leading to data integrity issues.

### 2.3.4 XPath injection (XPathi)

It is an attack technique that injects an XPath (XML Path Language) query in the input of a program, in order to query or navigate an XML document. This attack can be accomplished by placing a meta-character into the input string, which alters the behavior of the original query by modifying the query logic or bypassing authentication. XPathi can be exploited directly by an application to query an XML document as part of a larger operation, such as applying an XSLT transformation or an XQuery to an XML document.

As an example, consider the aforementioned document `students.xml` and the snippet of Java code below, which retrieves the student identification number with an XPath query:

```
1 File db = new File("students.xml");
2 Document doc = DocumentBuilderFactory.newInstance()
```

```

3         .newDocumentBuilder().parse(db);
4 XPath xpath = XPathFactory.newInstance().newXPath();
5 String query = "//students/student[uid/text()='\"
6             + req.getParameter(\"uid\")
7             + \"'_and_pwd/text()=\"_\"
8             + req.getParameter(\"pwd\")
9             + \"']/sid\"");
10 NodeList nl = (NodeList) xpath.evaluate(query, doc);

```

The XPath query is built dynamically using the inputs received through a Web request. An attacker could access all the student identification numbers by simply entering as user id the string 'foo' or '1=1' or 'a='a' and any random password. In this way, the XPath query will look like `//students/student[uid.text()='foo' or 1=1 or 'a='a' and pwd.text()='nopwd']/sid` and the conditions of the selection will always evaluate to true, returning all the nodes and thus possibly leaking confidential information.

### 2.3.5 LDAP injection (LDAPi)

It is an attack technique that targets programs that build LDAP statements based on user input. The attack can be accomplished by inserting meta-characters or crafted LDAP filters that alter the logic of the query. As a consequence, permissions can be granted for unauthorized queries or for modifying the LDAP tree.

As an example, consider the code snippet below, extracted from an LDAP-based authentication system:

```

1 DirContext ctx = new InitialDirContext(env);
2 String userid = req.getParameter("userid");
3 String pwd = req.getParameter("pwd");
4 String base = "OU=snt,DC=uni,DC=lu";
5 String filter = "(&(sn=" + userid + ")(password="
6 + pwd + "))";
7 SearchControls ctls = new SearchControls();
8 NamingEnumeration<SearchResult> results =
9 ctx.search(base, filter, ctls);

```

where `env` is a `HashTable` object containing the environment properties for the LDAP connection. The filter object is dynamically constructed using the user input strings (`userid` and `pwd`) and then used for querying the LDAP server. If an attacker knows a valid user id (e.g., "briandli"), he can make an attack by entering a user id of the form `briandli(&)`, and any value for the password (e.g., "nopwd"). This malicious string makes the filter string look like `(&(sn=briandli(&))(password=nopwd))`. Since an LDAP server processes only the first filter, the query will return true and will grant access to the attacker, even if he does not know the password of user `briandli`.

## 2.4 Threat Models

A *threat model* describes possible attacks that can be conducted through an input used in a sink. If an input can potentially contain values that match a threat model, the sink that uses such an input should be marked as vulnerable. According to our definition of a threat model and based on the types of vulnerabilities we focus on in this paper, an attacker is not required to know the source code of the application; we only assume that the attacker is capable of providing input to the potentially vulnerable Web application through an input source.

$$\begin{aligned}
 \text{TAUTCSTR}(input, ctxDel) &= \text{TAUTCSTRNUM}(input, ctxDel) \vee \text{TAUTCSTRSTR}(input, ctxDel) \\
 \text{TAUTCSTRNUM}(input, ctxDel) &= \bigvee_{nrel \in \{>, <, \leq, \geq, =, \neq\}} ( \\
 &\quad input.matches(".*".concat(ctxDel.concat(" +[0o][Rr] ")) \\
 &\quad .concat(N_1.toString()).concat(nrel.toString()) \\
 &\quad .concat(N_2.toString()).concat(".*") \wedge N_1 \text{ nrel } N_2) \\
 \text{TAUTCSTRSTR}(input, ctxDel) &= \bigvee_{cstr \in \{=, \neq\}} \bigvee_{d \in \{', \"\}} ( \\
 &\quad input.matches(".*".concat(ctxDel.concat(" +[0o][Rr] ")) \\
 &\quad .concat(d).concat(S_1).concat(d).concat(cstr.toString()) \\
 &\quad .concat(d).concat(S_2).concat(d).concat(".*") \wedge S_1 \text{ cstr } S_2)
 \end{aligned}$$

Figure 1. Tautology constraint template TAUTCSTR

In our approach we support the threat models listed in Table 1, which are based on various attack patterns defined as part of the OWASP security project [18]; for each model we indicate the corresponding constraint on the input. They are grouped by the type of sink (i.e., the type of vulnerability they exploit), and for each sink type we indicate also various *contexts*, which denote the possible ways in which inputs can be used in a sink. Each threat model is indicated for a specific context of a specific sink type, to reflect the specialization of an attack to exploit a certain vulnerability with a particular input. Notice that a precise characterization of the threat models is a fundamental step required to minimize the number of false positive and false negative results yielded by a vulnerability analysis technique. This list of threat models is not exhaustive but new attack patterns can be supported by modeling them with their corresponding constraint.

Threat models 1, 12, and 13 are applicable to the input used in element contents of HTML and XML documents. They reflect the attacks containing meta-characters such as `<` and `>`, which can be used to inject additional (malicious) elements into a document. For example, the constraint corresponding to the first threat model `input.matches(".*[<>/.]*")` matches attacks like `<script>alert();</script>`.

Threat models 2, 3, and 17 are applicable to the input used as value of event handlers, for source attributes in HTML documents, and for external entities in XML documents. No input should be allowed in these contexts, since an attack can be conducted by simply providing URLs pointing to malicious hosts, by injecting JavaScript code such as `javascript:alert()`, or by using the value `/etc` in an external entity of an XML document (to gain unauthorized access to local files). Moreover, input sanitization would not help in this case, since these attacks do not need to use meta-characters to be effective. Hence, these threat models are expressed with the constraint `input.matches(".*+")`, which enforces `input` to match any character except the empty string.

Threat models 4–7, 14–16, and 21 are applicable to the input used as the value of HTML, XML, and LDAP attributes. The difference among these models lies in the different type of quotation used for the attribute values. For example, if an input is enclosed with single quotes (as in `<div attr='input'`), an attack could be conducted by providing as input the single quote character `'` followed by an attack payload (e.g., a payload string like `'onmouseover=javascript:alert()`, which would inject an

Table 1  
Constraints corresponding to threat models

No.	Sink type	Context	Constraint
1	XSS	Element content: <code>&lt;tag&gt;input&lt;/tag&gt;</code>	<code>input.matches(".*[&lt;&gt;].*")</code>
2		Event handler value: <code>&lt;... onclick="input"&gt;</code>	<code>input.matches(".*+")</code>
3		Source value: <code>&lt;iframe src="input"&gt;</code>	<code>input.matches(".*+")</code>
4		Attribute value with single quotes: <code>&lt;div attr='input'&gt;</code>	<code>input.matches(".*'.*")</code>
5		Attribute value with double quotes: <code>&lt;div attr="input"&gt;</code>	<code>input.matches(".*\".*")</code>
6		Attribute value without quotes: <code>&lt;div attr=input&gt;</code>	<code>input.matches(".*[=&lt;&gt;/,;+-%\\*\\[\\]].*")</code>
7		URL parameter value: <code>&lt;a href="http://...?param=input"&gt;</code>	<code>input.matches(".*['\"=&lt;&gt;/,;+-%\\*\\[\\] ].*")</code>
8	SQLi	Attribute value with single quotes: <code>SELECT column From table WHERE row='input'</code>	<code>TAUTCSTR(input, "'")</code>
9		Attribute value with double quotes: <code>SELECT column From table WHERE row="input"</code>	<code>TAUTCSTR(input, "\"")</code>
10		Attribute value with date delimiters: <code>SELECT column From table WHERE row=#input#</code>	<code>TAUTCSTR(input, "#")</code>
11		Attribute value without quotes or delimiters: <code>SELECT column From table WHERE row=input</code>	<code>TAUTCSTR(input, "")</code>
12	XMLi	Element content: <code>&lt;node&gt;input&lt;/node&gt;</code>	<code>input.matches(".*[&lt;&gt;].*")</code>
13		CDATA content: <code>&lt;![CDATA[input]]&gt;</code>	<code>input.matches(".*\\[\\]&gt;.*")</code>
14		Attribute value with single quotes: <code>&lt;node attr='input' /&gt;</code>	<code>input.matches(".*'.*")</code>
15		Attribute value with double quotes: <code>&lt;node attr="input" /&gt;</code>	<code>input.matches(".*\".*")</code>
16		Attribute value without quotes: <code>&lt;node attr=input /&gt;</code>	<code>input.matches(".*['\"=&lt;&gt;].*")</code>
17	XPathi	External entity: <code>&lt;!ENTITY xxe SYSTEM "input"&gt;</code>	<code>input.matches(".*+")</code>
18		Attribute value with single quotes: <code>//table[column='input']</code>	<code>TAUTCSTR(input, "'")</code>
19		Attribute value with double quotes: <code>//table[column="input"]</code>	<code>TAUTCSTR(input, "\"")</code>
20		Attribute value without quotes or delimiters: <code>//table[column=input]</code>	<code>TAUTCSTR(input, "")</code>
21	LDAPi	LDAP search: <code>search="(attr=input)"</code>	<code>input.matches(".*[()\\*&amp;].*")</code>

additional JavaScript event). Such an attack is matched by the `.*'` regular expression. A similar threat model is defined for inputs with double quotes. If the input is not enclosed by any type of quote, various meta-characters (e.g., `=`, `*`, and `;`) may be used to conduct an attack like the one above. This type of attack is matched by threat models 6, 7, and 16, where the list of meta-characters is specific to the context in which they can be applied.

Threat models 8–11 and 18–20 are applicable to the input used as attribute value in SQL and XPath queries. These models reflect the various patterns of tautology attacks discussed in § 2.3, which cause the selection clause of an SQL or XPath query to always evaluate to true. We express them using the parameterized constraint template `TAUTCSTR`, whose definition is shown in Figure 1. This template has two parameters: `input` is a string variable representing the input to be matched against the tautology pattern; `ctxDel` represents the string delimiter used for enclosing the context of `input` (as shown in threat models 8–10 and 18–19). The template is defined as a disjunction of two constraints, each of them expressed through a sub-template: `TAUTCSTRNUM`, expressing numeric tautology attacks of the form `x' or N1 nrel N2`, where `N1` and `N2` are integer variables and `nrel`  $\in \{>, <, \leq, \geq, =, \neq\}$ ; `TAUTCSTRSTR`, expressing string tautology attacks of the form `x' or S1 cstr S2`, where `S1` and `S2` are string variables and `cstr`  $\in \{=, \neq\}$ . Template `TAUTCSTRNUM` is defined as a disjunction of constraints over `nrel`; each disjunct consists of two conjuncts:

1) The first conjunct generates a pattern against which the user input variable `input` has to be matched. The concatenation of string `.*'` with the context delimiter `ctxDel` encloses the context of `input`. Afterwards, the actual attack pattern is generated by concatenating the string `+ [0o][Rr]`

with the string representation of `N1`, together with the string representation of `nrel`, which is then concatenated with the string representation of `N2` and the string `.*'`.

2) The second conjunct `N1 nrel N2` enforces the numeric constraint defined by `nrel` on `N1` and `N2` to ensure that only satisfiable tautologies are accepted.

`TAUTCSTRSTR` is defined as a disjunction of constraints over `cstr`; each disjunct consists of two conjuncts, which are structurally similar to those used in the definition of `TAUTCSTRNUM`. The main difference is that string variables `S1` and `S2`, when used in a string concatenation, are always enclosed by a pair of delimited characters represented by the variable `d`, which ranges (through the inner disjunction) over the set `{', '"}`.

### 3 MOTIVATION

In this section we present a motivating example that highlights the challenges in adopting an approach based on symbolic execution and constraint solving in the context of vulnerability detection. Although we crafted this example for illustrative purposes, it can be considered realistic since it contains typical operations that are commonly found in modern Web applications. Moreover, it contains vulnerabilities that embody the patterns tracked in the CWE dictionary [19].

The program, shown in Figure 2, contains two sinks. The first sink is at line 29 and corresponds to an XSS vulnerability within an HTML output operation; the second one is at line 38 and corresponds to an XPATHi vulnerability within an XPath query.

The sink at line 29 is vulnerable to XSS because of the inadequate sanitization procedure applied to variable `sid`, which contains a user input. More specifically, it is sanitized



```

1 protected void doPost(HttpServletRequest req,
2   HttpServletResponse res) {
3   res.setContentType("text/html;charset=UTF-8");
4   PrintWriter out = res.getWriter();
5   Document doc = DocumentBuilderFactory
6     .newInstance()
7     .newDocumentBuilder().parse("./students.xml");
8   XPath xpath=XPathFactory.newInstance()
9     .newXPath();
10  out.println("<html><head><title>"
11    + "Student_Services" +
12    + "</title></head><body>...");
13  String op = req.getParameter("option");
14  String sid = req.getParameter("id");
15  int max = Integer.parseInt(req.
16    getParameter("max"));
17  String subj = req.getParameter("subjid");
18  if(subj.trim().toLowerCase()
19    .indexOf("code", 4) <= 10) {
20    subj="*";
21  }
22  if(max>20)
23    max=20;
24  if(op.trim().equalsIgnoreCase("GradeQuery")) {
25    if(sid.length()>max) {
26      sid=customSanit(sid); //remove chars <, >, /
27      out.println("<a href=\"foo.com?id=\""
28        + sid + "\">Invalid_ID:_"
29        + sid + "</a>"); //XSS sink
30      // ... other statements
31    } else {
32      sid=ESAPI.encoder().encodeForXPath(sid);
33      subj=ESAPI.encoder().encodeForXPath(subj);
34      String query = "//students/grade[sid=\""
35        + sid + "\"_and_subjid=\""
36        + subj + "\"]/mark";
37      NodeList nl=(NodeList)xpath.
38        evaluate(query, doc); //XPath sink
39      //... other statements
40    }
41  }
42 }

```

Figure 2. A Java servlet program vulnerable to XPathi and XSS

by applying a custom function `customSanit()` (line 26), which removes the meta-characters `<`, `>`, and `/` from the input string variable `sid`. However, in this operation variable `sid` is used in two different contexts: as an URL parameter value (`...com?id=sid`) and as the content of an HTML element `<a href>...sid</a>`. The sanitization procedure used is appropriate for the second context since it prevents the injection of additional HTML tags like `<script>`. However, it is not appropriate for the first one, which would have required URL encoding (also called percent-encoding).

The sink at line 38 is vulnerable to XPathi because the variable `sid`, containing a user input, is not sanitized properly before using it in the XPath query. Indeed, the standard sanitization procedure `ESAPI.encoder().encodeForXPath()` from OWASP [10] applied to variable `sid` only escapes meta-characters such as `'` and `''`. Assuming that the element `sid` is defined as a numeric data type in the schema of the document `students.xml` (the same presented in Section 2.3), one could still perform a successful attack without using those meta-characters, for example using the input 1 or 0<1. This example shows

that sanitization, even when achieved by applying widely used and well-tested sanitization libraries, does not always work. Indeed, sanitization libraries often provide operations that filter user input *only* based on a certain context (an XPath attribute in the example above), without necessarily considering all possible cases.

The two vulnerabilities in the example can be discovered by using symbolic execution and constraint solving, combined in a three-step procedure:

- 1) *Path conditions generation through symbolic execution.* For example, one of the path conditions generated by symbolically executing a path leading to the execution of the XSS sink at line 29 is:

$$\begin{aligned}
 PC \equiv & SUBJ.trim().toLowerCase() \\
 & \text{indexOf}("code", 4) > 10 \wedge \\
 & \text{Integer.parseInt}(MAX) > 20 \wedge \\
 & OP.trim().equalsIgnoreCase("GradeQuery") \wedge \\
 & SID.length() > 20
 \end{aligned}$$

where *OP*, *SID*, *MAX*, *SUBJ* are symbolic values for the variables initialized with the Web request inputs (lines 13–17).

- 2) *Definition of the attack specification.* In this step, usually performed by a security expert<sup>2</sup>, the attack specification is defined in a way that properly characterizes security threats. It can be done by writing attacks that match any of the attack patterns listed in § 2.4. For example, the security attack `&javascript:alert('XSS')` is a match for the regular expression of threat model #7 in Table 1. The attack specification can then be represented as follows:

$$\begin{aligned}
 ATTK \equiv & \text{customSanit}(SID).contains( \\
 & "&javascript:alert('XSS')")
 \end{aligned}$$

where `customSanit(SID)` is the symbolic expression over the symbolic value *SID* representing the values of variable *sid* at the XSS sink.

- 3) *Constraint solving.* The third step requires to solve the *attack condition*, defined as the constraint obtained by conjoining the path condition with the attack specification; this step is performed using a constraint solver. If the solver yields SAT, showing the satisfiability of the constraint, it means that the attack is feasible and that the analyzed path is vulnerable to the attack. In the example, the constraint  $SECI \equiv PC \wedge ATTK$  is satisfiable, confirming the presence of the XSS vulnerability.

However, executing this procedure faces two main challenges:

**CH1: Integrated approach.** Performing the three steps illustrated above requires the integration of program analysis techniques (to identify input sources and sinks, to analyze paths between input sources and sinks, etc.), symbolic execution, definition of security threat models, and constraint solving. This integration is not trivial since it has to be realized keeping in mind that the output of each step has to become the input of the next step. This often requires to pre-process the output of each step, before feeding it to the next one. For example, to avoid the path explosion

2. This step needs to be done once for each type of vulnerability, and possibly refined over time if needed.



problem, the symbolic execution step should not explore all the paths of the program, but only those traversing security-sensitive locations (i.e., sources and sinks). Therefore, symbolic execution has to be complemented by techniques such as security slicing [6]. Similarly, the attack conditions could benefit from a simplification step, to speed-up the constraint solving phase.

Furthermore, the majority of existing approaches focus only on one of these steps. For instance, CVC4 and Z3 only focus on constraint solving and assume that constraints (including those corresponding to attack specifications) are already available; vulnerability detection approaches such as Andromeda [20], Taj [21], and SFlow [11] only perform static analysis, and do not apply symbolic execution and constraint solving.

**CH2: Support for complex string operations.** The execution of the first step above assumes that symbolic execution uses a constraint solver that is able to handle string operations like `trim`, `toLowerCase`, `indexOf`, `parseInt`, `equalsIgnoreCase`, and `length`. However, state-of-the-art solvers such as Hampi [22], Kaluza [2], CVC4 [13], Z3-str2 [23] and Z3 [14] do not support at least one of these non-basic, complex operations. When a string operation is not supported by the solver, symbolic execution typically has to analyze the implementation of the operation, to transform it into an equivalent set of basic constraints containing only primitive operations, i.e., operations supported by the solver. This approach may lead to the path explosion problem and, more generally, scalability issues [5], [24], especially when the implementations of unsupported operations contain many paths.

In principle, one could modify or enhance an existing solver in order to provide native support for complex operations, and avoid the path explosion problem in symbolic execution. However, this task is non-trivial and requires a deep understanding of string manipulating functions and constraint solving; moreover, it is not scalable to the size of a sanitization library like OWASP ESAPI or of a complete string function library of a modern programming language. Alternatively, instead of modifying the solvers, one could re-express complex operations with their equivalent set of basic constraints that can be solved by the solver. Although relatively easier, this alternative still requires non-trivial effort and usually results in complex constraints that may still lead to scalability issues for constraint solvers [3]. For example, consider one of the clauses in the above path condition: `OP.trim().equalsIgnoreCase("GradeQuery")`; assuming that the solver handles `length`, `charAt`, `equals`, and `substring`, one could re-express this clause as:

$$\begin{aligned} &\exists c_1, c_2, 0 \leq c_1 \leq c_2 \leq OP.length(), \text{ such that} \\ &OP.substring(c_1, c_2).equalsIgnoreCase("gradequery") \\ &\vee \dots \vee \dots equals("gRadeQueRy") \dots \vee \dots \vee \\ &OP.substring(c_1, c_2).equalsIgnoreCase("GRADEQUERY") \\ &\wedge \forall i, 0 \leq i < c_1, OP.charAt(i) = ' ' \\ &\wedge \forall j, c_2 < j \leq OP.length(), OP.charAt(j) = ' ' \end{aligned}$$

which uses equivalent constraints for `equalsIgnoreCase` and `trim`. Notice how the `equalsIgnoreCase` operation is expanded into a disjunction of constraints with the `equals`

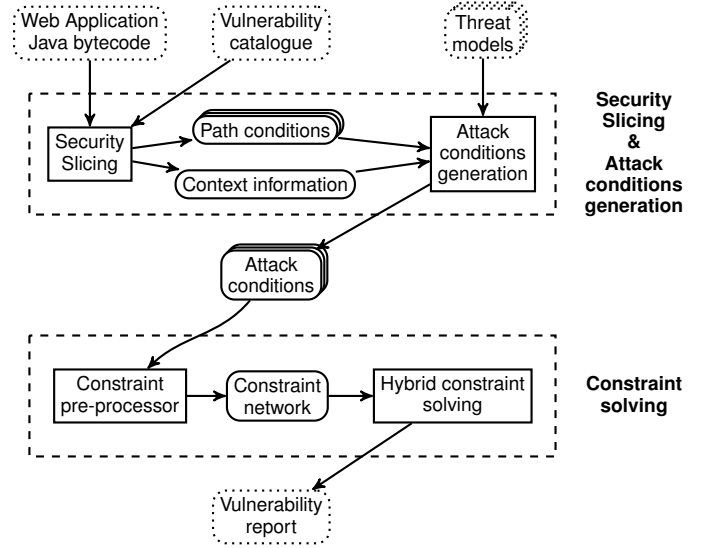


Figure 3. Overview of the approach

operation, which cover all the possible combinations of the characters denoting a case-insensitive representation of the string “GradeQuery”; also, modeling the `trim` operation requires to add several auxiliary variables and predicates. This simple example shows that transforming complex constraints into a set of equivalent constraints with only primitive operations increases the complexity of the generated constraints, potentially leading to scalability issues [5], [24]. Another approach [1] to deal with complex string operations relies on dynamic symbolic (concolic) execution [25], and treats complex operations by replacing symbolic values of the inputs involved in those operations with concrete values. However, this approach reduces precision since it can only reason about the paths that are exercised by the concrete values.

To work around this issue, the current solution in practice is to have the constraint solver fail (i.e., it crashes or returns an error) when it encounters an unsupported operation. Our experiments show that this is the case for state-of-the-art solvers like CVC4 [13] and Z3 [14]. However, in the context of vulnerability analysis, such a behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

To recap, the two challenges discussed above show that in order to use symbolic execution and constraint solving as effective and scalable techniques for vulnerability detection, there is a need for an end-to-end approach that seamlessly incorporates scalable program analysis techniques, modeling of security threats, and complex (string) constraints solving techniques.

## 4 OVERVIEW OF THE APPROACH

Our approach is outlined in Figure 3, where dotted rounded rectangles correspond to global inputs/outputs, solid rounded rectangles correspond to intermediate input-/outputs, solid rectangles correspond to operations, and dashed rectangles correspond to macro-steps.

The approach takes as input the bytecode of a Web application written in Java, a catalogue of vulnerabilities,

and a list of threat models; it yields a vulnerability report. The catalogue of vulnerabilities contains a characterization of the criteria for identifying *input sources* and *sinks* related to specific vulnerabilities; for more details, we refer the reader to our previous work [6]. The vulnerability report lists the vulnerabilities found in the Web application, and for each of them indicates the type and the corresponding sink. Our approach is composed of two macro-steps:

**Security slicing & Attack conditions generation.** This step first performs *security slicing* [6], i.e., given the bytecode of the Web application to analyze and the catalogue of vulnerabilities, it identifies sinks in the program and for each sink computes the path condition leading to it and the associated context information. The path condition and the context information of each sink are then used, together with the list of threat models, to generate *attack conditions*, i.e., conditions that could trigger a security attack over a security slice. This step is based on our previous work on security slicing [6]; we provide a brief summary of it in section 5.

**Constraint solving.** This step takes as input the attack conditions generated in the previous macro-step, in the form of a constraint. This constraint is first pre-processed to simplify it through the *constraint pre-processing* step (detailed in section 6). The resulting constraint, represented as a constraint network, is then given as input to a *hybrid constraint solver*, which orchestrates a constraint solving procedure for string and integer constraints with our search-based constraint solving procedure [7]; more details of this step are presented in section 7. The results yielded by the hybrid constraint solver are used to create the vulnerability report.

## 5 SECURITY SLICING AND ATTACK CONDITIONS GENERATION

In previous work we proposed security slicing [6], which is a technique that extracts all the program statements required for auditing the security of a given sink. It is similar to interprocedural program slicing [26], which extracts all statements across procedures that affect a given statement of interest. However, interprocedural program slices typically contain large amounts of information that is irrelevant to security, which lead to scalability issues in security audits. A security slice is more concise and contains the minimal sequence of program statements necessary to determine the vulnerability of a sink.

The main idea of security slicing is to prune: 1) library code that is known to be correct or irrelevant to security analysis; 2) secure program paths that apply proper, standard sanitization on the input variables that are used in a sink; 3) vulnerable program paths that can be fixed automatically because user input is directly used in the sink. Notice that for cases where the data-flow and control-flow between sources and sinks are more complex, it would not be possible to automatically fix the user input without affecting the integrity of the input data, possibly resulting in a malformed fix, changing the semantics of the original program, or even introducing a new security vulnerability.

To identify sources and sinks in the Web application under test, security slicing relies on a vulnerability catalogue, i.e., a predefined set of sink and source signatures,

```

1 protected void doPost(HttpServletRequest req,
2   HttpServletResponse res) {
3   res.setContentType("text/html;charset=UTF-8");
4   PrintWriter out = res.getWriter();
5   String op = req.getParameter("option");
6   String sid = req.getParameter("id");
7   int max = Integer.parseInt(req.
8     getParameter("max"));
9   if(max>20)
10    max=20;
11   if(op.trim().equalsIgnoreCase("GradeQuery")) {
12     if(sid.length()>max) {
13       sid=customSanit(sid); //remove chars <,>,/
14       out.println("<a_href=\"foo.com?id="
15         + sid + ">Invalid_ID:␣
16         + sid + "</a>"); //XSS sink
17     }
18   }
19 }

```

Figure 4. The security slice of the XSS sink at line 29 in Figure 2

which can be easily extended by adding new signatures. As part of our previous work [6], we preconfigured a default vulnerability catalogue that contains a rich set of commonly used API signatures.

Security slicing performs symbolic execution on each path in a security slice to extract the path condition(s) characterizing the path. 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. Examples of possible contexts are the content or an attribute of an HTML tag, as well as a quoted value of an SQL query. This information can be computed through *context analysis*, a lightweight technique for identifying the context (within a sink) in which the data of an input source is used. More specifically, context analysis<sup>3</sup> is a backward data flow analysis which traces the values of the variables used in the sink along the path, to reconstruct the query (e.g., SQL/XPath/LDAP query) or the document part (e.g., HTML/XML fragment) that is being generated at the sink. Context information is computed by matching the reconstructed query or document part against some predefined patterns (shown in the “Context” column of Table 1).

We give a bird’s eye view of security slicing through the example in Figure 4, which shows the security slice for the (XSS) sink at line 29 in Figure 2. The security slicing procedure filtered out library code from the Document, HttpServletRequest, HttpServletResponse, and PrintWriter classes, since they can be considered as irrelevant to security [6]. It also filtered out two of the four program paths leading to the XSS sink in Figure 2; the two pruned paths correspond to those including the predicate at line 19 in Figure 2, which does not affect the sink. As a result, the security slice shown in Figure 4 contains only two paths leading to the sink. The first path is characterized by path condition *PC1*, which corresponds to the path that follows the true branch of the

3. Context analysis attempts to detect the appropriate threat model, and is thus different from the path exploration method presented in [27], which aims at improving the scalability of symbolic execution by grouping program paths that are equivalent with respect to the produced output.

selection statement at line 9 in Figure 4 and leads to the execution of the sink; this path condition is:

$$PC1 \equiv \text{Integer.parseInt}(MAX) > 20 \wedge \\ OP.\text{trim}().\text{equalsIgnoreCase}("GradeQuery") \\ \wedge SID.\text{length}() > 20$$

The second path is characterized by path condition  $PC2$ , which corresponds to the path that follows the false branch of the selection statement at line 9 in Figure 4 and leads to the execution of the sink; this path condition is:

$$PC2 \equiv \neg(\text{Integer.parseInt}(MAX) > 20) \wedge \\ OP.\text{trim}().\text{equalsIgnoreCase}("GradeQuery") \\ \wedge SID.\text{length}() > \text{Integer.parseInt}(MAX)$$

For both paths, our context analysis procedure [6] identifies the following two contexts:  $CTX1$ , in which the symbolic expression  $\text{customSanit}(SID)$  is used as a URL parameter value in an XSS sink;  $CTX2$ , in which the symbolic expression  $\text{customSanit}(SID)$  is used as an element content in an XSS sink. Note that the symbolic expression  $\text{customSanit}(SID)$  represents the values of variable  $sid$  used at the sink.

The output of security slicing—path conditions and context information—is used to generate attack conditions, represented as constraints. A new constraint is generated for each context identified for each path, based on the threat model characterizing the security threat in that specific context. The attack condition generation (ACG) process follows three steps:

**ACG1)** Since different contexts require different threat models, the procedure determines the appropriate threat model for a given context by looking up the list of threat models provided as input. The identification of the threat model is a fully automated procedure that matches the context returned by security slicing with one of the entries in the threat models list. The predefined version of this list has been presented in § 2.4; though not showed in Table 1, the predefined list also contains catch-all entries<sup>4</sup> for each type of vulnerability, which are used as fallback mechanism when there is no context matching pattern. Furthermore, the structure of the list guarantees that there is always only one applicable threat model for a given context. For example, the threat model for context  $CTX1$  is #7 in Table 1; likewise, the threat model for  $CTX2$  is #1.

**ACG2)** In the constraint corresponding to each identified threat model, the symbol *input* is replaced with the actual symbolic expression of the input. This results in a constraint that checks if an input used at the sink contains a security attack. For example, the constraint  $\text{input.matches}(".*['\\"=< > / ; + - \& * \backslash [ \ ] ] . *")$  — corresponding to threat model #7 (see Table 1)— results in the following constraint  $ATTK1$ , related to context  $CTX1$ :

$$ATTK1 \equiv \text{customSanit}(SID) \\ .\text{matches}(".*['\\"=< > / ; + - \& * \backslash [ \ ] ] . *")$$

Likewise, the constraint  $ATTK2$  related to  $CTX2$  is:

$$ATTK2 \equiv \text{customSanit}(SID).\text{matches}(".*[< > / ] . *")$$

4. The catch-all entries for threat models are more generic and might lead to false positives in terms of vulnerability detection.

**ACG3)** For each constraint generated in the previous step and a given path condition, the attack conditions are generated by simply conjoining the path condition with the constraint. For example, the attack condition  $SEC1$  is the constraint conjoining  $PC1$  and  $ATTK1$ :

$$SEC1 \equiv \text{Integer.parseInt}(MAX) > 20 \wedge \\ OP.\text{trim}().\text{equalsIgnoreCase}("GradeQuery") \wedge \\ SID.\text{length}() > 20 \wedge \\ \text{customSanit}(SID) \\ .\text{matches}(".*['\\"=< > / ; + - \& * \backslash [ \ ] ] . *")$$

Likewise, attack condition  $SEC4$  conjoins  $PC2$  and  $ATTK2$ :

$$SEC4 \equiv \neg(\text{Integer.parseInt}(MAX) > 20) \wedge \\ OP.\text{trim}().\text{equalsIgnoreCase}("GradeQuery") \wedge \\ SID.\text{length}() > \text{Integer.parseInt}(MAX) \wedge \\ \text{customSanit}(SID).\text{matches}(".*[< > / ] . *")$$

Similar attack conditions (omitted for space reasons) are computed by conjoining  $PC1$  and  $ATTK2$ , as well as  $PC2$  and  $ATTK1$ .

More details on security slicing and context analysis are available in our previous work [6].

## 6 CONSTRAINT PREPROCESSING

Constraints corresponding to attack conditions generated from the previous steps are represented as constraint networks, which are a common representation of instances of a constraint satisfaction problem [28].

We build the hypergraph of the constraint network representing an attack condition similarly to how string graphs are constructed in [29]. The nodes of the graphs are either constant values or string/integer variables appearing in the constraints of the network. Each operation (e.g., method calls like  $\text{trim}()$  or comparison operators like  $>$ ) in an attack condition corresponds to exactly one hyperedge in the constraint network graph. Notice that boolean operators are represented as hyperedges labeled with the operator itself. Transformational operations (which return a value of type different from boolean) require to add a node to the graph, representing an auxiliary variable that corresponds to the result of the operation; the transformational operation is then added as a hyperedge that connects the nodes of the initial constants/variables and the new auxiliary variable. The latter is denoted with a name ending with a prime (') symbol, except when the transformational operation is  $\text{length}$ , in which case we use a name of the form  $l_{VAR}$ , where  $VAR$  is the original variable.

For example, the constraint network corresponding to attack condition  $SEC1$  is shown in Figure 5. Rounded nodes represent variables, squared nodes represent constant values, hyperedges are denoted by lines that meet at a black dot. A hyperedge is labeled with the name of the corresponding operation and with numbers that indicate the role of its component nodes in the corresponding operation (i.e., order of function arguments, return variable). For instance, the node labeled  $MAX'$  corresponds to the auxiliary variable resulting from the application of the transformational operation  $\text{parseInt}()$  to variable  $MAX$ ; the node labeled

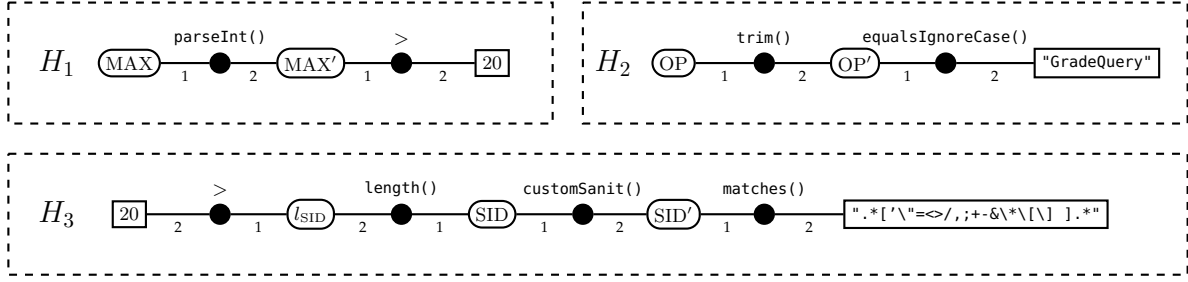


Figure 5. Constraint network equivalent to the attack condition *SEC1*

$l_{SID}$  is the auxiliary variable that represents the length of variable *SID*. Notice that, to keep figures readable, we omit the representation of hyperedges labeled with a boolean AND.

Once a constraint network is constructed, we preprocess it to apply some rules that simplify the solving procedure; these rules are captured by two preprocessing procedures: 1) *derived constraint generation* and 2) *constraint refinement*, executed in this order and applied through a work-list algorithm. Both procedures have been already described in [29]; here we propose new derived constraints and new rules to deal with additional operations. As described in § 6.3, both procedures preserve the equisatisfiability between the original and modified versions of the constraint network.

Note that when some of the variables or the constraints of a constraint network are independent, the underlying hypergraph is disconnected; in such a case, we apply the preprocessing procedures to each of the maximal connected components (i.e., hypersubgraphs) of the hypergraph. For example, in Figure 5, there are three hypersubgraphs, enclosed in dashed rectangles and denoted as  $H_1$ ,  $H_2$ , and  $H_3$ .

## 6.1 Derived Constraint Generation

For some operations, additional constraints are derived to reduce the input domain or to solve the constraints more efficiently. For example, given the constraint  $X.contains(Y)$ , one can generate the derived constraint  $l_X \geq l_Y$  on the length of  $X$  and  $Y$ . The addition of these derived constraints reduces the size of the variable domains and may lead to unsatisfiability results faster, since some of the new derived constraints may be easier to solve (e.g., because they use a smaller number of variables).

Table 2 shows the derived constraints corresponding to string/mixed operations; the derived constraints marked with a star are introduced for the first time in this paper, while the others have been taken from [29], [30]. Notice that some operations in Table 2 may return a specific value to indicate an error: for example, `indexOf` returns a negative value when the search string is not found; we model this semantics using logical implications.

The derived constraints are then included in the constraint network accordingly. For example, Figure 6 shows the constraint network obtained after generating the derived constraints for the network in Figure 5; the new constraints are derived from the `trim`, `equalsIgnoreCase`, and `length` operations.

## 6.2 Constraint Refinement

In this step, some rules are used to simplify the constraint network and to detect trivially inconsistent constraints, to avoid unnecessary and expensive constraint solving. For example, if there is a constraint of the form  $X.equals(Y)$ , the hyperedge corresponding to the equality constraint and one of the nodes (either  $X$  or  $Y$ ) can be removed, and its connected hyperedges can be redirected to the remaining node.

Table 4 shows the constraint refinement rules for specific types of hyperedges (or pair of hyperedges), as well as a pictorial representation of them; all the rules have been already described in the literature [29], [31], [32], [33].

Rule 1 corresponds to the evaluation of a transformational operation involving only constants. The operation is actually executed to determine its result; the node corresponding to the auxiliary variable is replaced by a single constant node corresponding to the computed result. The hyperedge labeled with the transformational operation is removed from the hypergraph, and also its associated constant nodes, if they are not connected to any other hyperedge.

Rule 2 is similar to the previous; it corresponds to the case in which a hyperedge is labeled with a relational or boolean operation involving only constants. The operation is evaluated; either it evaluates to true, and thus the hyperedge is removed from the hypergraph, or to false, and thus it determines the unsatisfiability of entire constraint network.

Rule 3 corresponds to the case in which there is a pair of hyperedges representing integer constraints, which matches one of the patterns shown in the left column of Table 3. This pair is replaced by a hyperedge representing the equivalent constraint, as indicated in the right side of Table 3. This table is based on well-known equivalences between numeric constraints. For example, the pair of hyperedges equivalent to a constraint of the form  $X \geq Y \wedge X = Y$ , is replaced by one hyperedge corresponding to the constraint  $X = Y$ .

Rules 4 and 5 are applicable to a hyperedge that corresponds to an equality constraint between two variables and, respectively, to an equality constraint between a variable and a constant. In both cases, the hyperedge corresponding to the equality constraint is removed, as well as one of the component nodes (the node corresponding to the variable in rule 5); the hyperedges that were connected to the removed node are reconnected to the other.

Rule 6 captures pairs of hyperedges corresponding to inconsistent constraints (imposed on the same variables) of the form  $RelOp(v_1, \dots, v_K) \wedge \neg RelOp(v_1, \dots, v_K)$ ; this

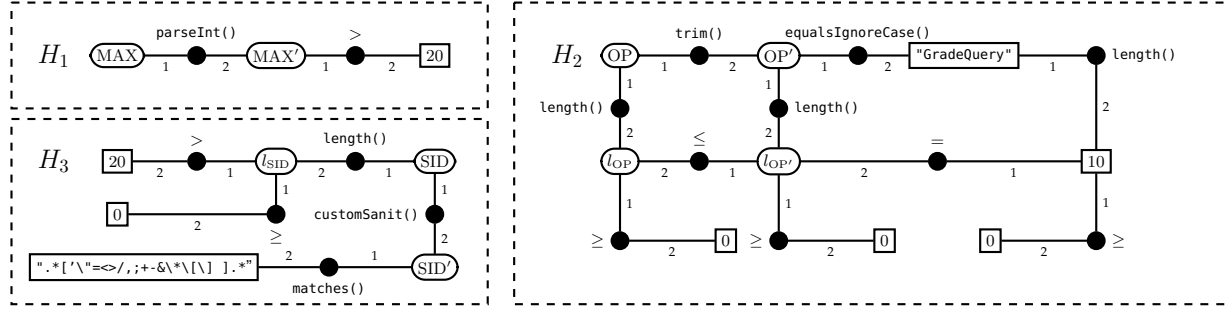


Figure 6. The constraint network in Figure 5 augmented with the derived constraints

Table 2

String/mixed operations and their corresponding derived constraints ( $X$  and  $Y$  are string variables,  $X'$  is an auxiliary string variable,  $i$  and  $j$  are integer variables,  $l_T$  represents the length of string  $T$ )

String/mixed operation	Derived constraint(s)	Source
$X.contains(Y)$	$l_X \geq l_Y$	[29]
$X.startsWith(Y)$	$l_X \geq l_Y$	[29]
$X.endsWith(Y)$	$l_X \geq l_Y$	[29]
$X.isEmpty()$	$l_X = 0$	*
$X.concat(Y)$ $X + Y$ $X.append(Y)$	$l_{X'} = l_X + l_Y$	[29], [30] [29], [30] *
$X.equals(Y)$ $X.equalsIgnoreCase(Y)$ $X.contentEquals(Y)$	$l_X = l_Y$	[29], [30] * *
$String.valueOf(X)$ $valueOf(X)$ $X.toString()$	$l_X = l_{X'}$	* * *
$X.trim()$ $X.length()$ $X.indexOf(Y)$	$l_X \geq l_{X'}$ $l_X \geq 0$ $(l' \geq 0 \rightarrow \neg(X.substring(0, l').contains(Y)) \wedge X.substring(l', l_X).startsWith(Y) \wedge l_X \geq l_Y) \wedge$ $(l' < 0 \rightarrow \neg(X.contains(Y)))$	[29], [30] [29], [30] *
$X.lastIndexOf(Y)$	$(l' \geq 0 \rightarrow \neg(X.substring(l' + 1, l_X).contains(Y)) \wedge X.substring(l', l_X).startsWith(Y) \wedge l_X \geq l_Y) \wedge$ $(l' < 0 \rightarrow \neg(X.contains(Y)))$	*
$X.charAt(i)$	$l_{X'} = 1 \wedge X.contains(X')$	*
$X.toLowerCase()$	$l_X = l_{X'}$	*
$X.toUpperCase()$	$l_X = l_{X'}$	*
$X.isEmpty()$	$l_X = 0$	*
$X.substring(i)$	$0 \leq i < l_X \wedge l_{X'} = l_X - i \wedge X.contains(X')$	*
$X.substring(i, j)$	$0 \leq i < l_X \wedge i \leq j \leq l_X \wedge l_{X'} = j - i \wedge X.contains(X')$	*

Table 3

Patterns for the refinement of integer constraints used in rule 3 of Table 4

pattern	equivalent constraint
$X \geq Y \wedge X > Y$	$X > Y$
$X \geq Y \wedge X = Y$	$X = Y$
$X \geq Y \wedge X \neq Y$	$X > Y$
$X \leq Y \wedge X < Y$	$X < Y$
$X \leq Y \wedge X = Y$	$X = Y$
$X \leq Y \wedge X \neq Y$	$X < Y$
$X \leq Y \wedge X \geq Y$	$X = Y$

rule determines the unsatisfiability of the entire constraint network.

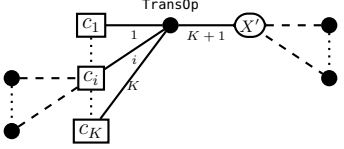
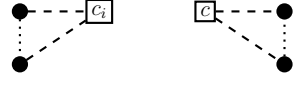
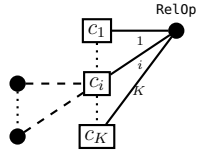
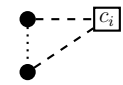
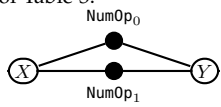
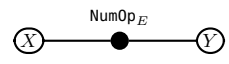
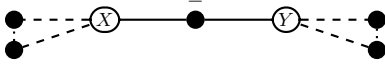
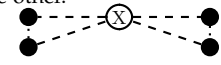
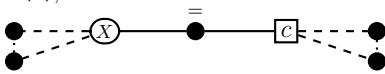
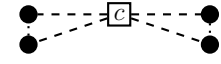
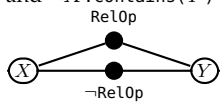
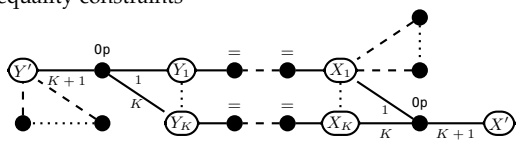
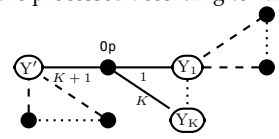
Rule 7 is applicable to pairs of hyperedges that are labeled with the same operation and whose parameters are connected through equality constraints; this means that the two hyperedges are semantically equivalent. They are

merged into a single hyperedge and the component nodes are processed according to rules 4 and 5.

Rules 4–7 use the theory of Equality of Uninterpreted Functions (EUF) [33], a widely used theory in constraint solving, to identify and merge semantically equivalent nodes and hyperedges.

As an example, the constraint network in Figure 6 is refined into the constraint network shown in Figure 7. More specifically, since the constraint  $10 \geq 0$  in  $H_2$  trivially evaluates to true, according to rule 2 the hyperedge labeled with  $\geq$  and the constant node 0 are removed from the network; also, according to rule 5, in  $H_2$  the hyperedge corresponding to constraint  $l_{OP'} = 10$  as well as the variable node for  $l_{OP'}$  are removed and the hyperedges that were connected to the latter are now connected to the node for constant 10; since one of the resulting hyperedges corresponds to the constraint  $10 \geq 0$ , rule 2 can be applied again as above, to remove the hyperedge.

Table 4  
Hyperedges and their corresponding refinement rules. ( $X, Y, X_1, X_K, Y_1, Y_K$  are variables,  $X'$  and  $Y'$  are auxiliary variables,  $c_1, \dots, c_K, 1 \leq i \leq K$  and  $c$  are constants)

ID	Hyperedge(s)	Refinement Rule	Source
1	<p><math>\text{TransOp}(c_1, \dots, c_K)</math> where <math>\text{TransOp}</math> is any transformational operation that involves only constants</p> 	<p>The operation <math>\text{TransOp}</math> is executed to determine its result <math>c</math>. The node corresponding to the auxiliary variable is replaced with the constant <math>c</math>. The hyperedge and the component nodes <math>c_i, 1 \leq i \leq K</math>, not connected to any other hyperedge, are removed from the hypergraph.</p> 	[31]
2	<p><math>\text{RelOp}(c_1, \dots, c_K)</math>, where <math>\text{RelOp}</math> is any relational or boolean operation that involves only constants</p> 	<p>If the operation <math>\text{RelOp}</math> returns true, the hyperedge and the component nodes <math>c_i, 1 \leq i \leq K</math>, not connected to any other hyperedge, are removed from the hypergraph. Otherwise, constraint unsatisfiability is detected. (The figure below corresponds to the case in which <math>\text{RelOp}</math> returns true)</p> 	[31]
3	<p>A pair of hyperedges corresponding to a pair of integer constraints that matches one of the patterns shown in the left column of Table 3.</p> 	<p>The two hyperedges are replaced by the hyperedge representing the equivalent constraint <math>\text{NumOp}_E</math> indicated in the right column of Table 3.</p> 	[32]
4	<p><math>X.\text{equals}(Y), X = Y</math></p> 	<p>The hyperedge and one of the component nodes (<math>X</math> or <math>Y</math>) are removed. The hyperedges that were connected to the removed node are reconnected to the other.</p> 	[29]
5	<p><math>X.\text{equals}(c), X = c</math></p> 	<p>Same as above, except that the variable <math>X</math>, not the constant <math>c</math>, is removed.</p> 	[29]
6	<p>A pair of hyperedges corresponding to inconsistent constraints between the same nodes, e.g., <math>X.\text{contains}(Y)</math> and <math>\neg X.\text{contains}(Y)</math></p> 	<p>Constraint unsatisfiability is detected.</p>	[29]
7	<p>A pair of hyperedges that are labeled with the same operation and whose parameters are connected through equality constraints</p> 	<p>The two hyperedges are merged into a single hyperedge and the component nodes are processed according to rules 4 and 5.</p> 	[33]

### 6.3 Constraint Preprocessing and Equisatisfiability

Both derived constraint generation (§ 6.1) and constraint refinement (§ 6.2) procedures apply changes to the constraint network. A change (addition, removal, or replacement of a hyperedge) applied to the *original* constraint network  $\mathcal{R}_0 = (X_0, D_0, C_0)$  results in a new network  $\mathcal{R}_1 = (X_1, D_1, C_1)$ , followed by the application of  $k - 1$  further changes by the worklist algorithm and resulting in the final network  $\mathcal{R}_k = (X_k, D_k, C_k)$ .

The sole purpose of constraint preprocessing is to simplify the solving procedure and, hence, the equisatisfiability between  $\mathcal{R}_0$  and  $\mathcal{R}_k$  has to be preserved. In the rest of this section we show how the application of any of the proposed constraint preprocessing rules to a constraint network  $\mathcal{R}_t$ , resulting in a new network  $\mathcal{R}_{t+1}$ , preserves the equisatisfiability between  $\mathcal{R}_t$  and  $\mathcal{R}_{t+1}$ , for  $0 \leq t < k$ . Since the equisatisfiability relation is transitive, we can conclude, by induction, that  $\mathcal{R}_0 \equiv_{\text{sat}} \mathcal{R}_k$ .

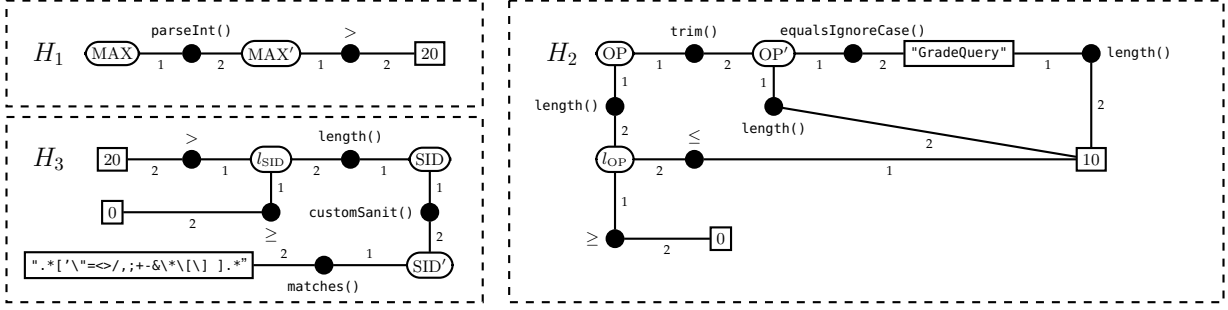


Figure 7. Constraint network resulting from the application of the constraint refinement rules to the network in Figure 6.

### Derived constraint generation

This procedure adds constraint(s) to a constraint network  $\mathcal{R}_t$ , yielding a new network  $\mathcal{R}_{t+1}$  with  $\mathcal{R}_t \subseteq \mathcal{R}_{t+1}$ . Note that during this procedure, besides new constraints, also new auxiliary variables may be added to the network.

For a generic rule in Table 2, we call  $\mathcal{R}_O = (X_O, D_O, C_O)$  the constraint network representing only the string/mixed operation in the first column of the table, and  $\mathcal{R}_D = (X_D, D_D, C_D)$  the constraint network representing the derived constraint added by the rule, as indicated in the second column of the table. To apply a preprocessing rule in Table 2 to a constraint network  $\mathcal{R}_t$ ,  $\mathcal{R}_O$  has to be a subgraph of  $\mathcal{R}_t$ . After the application of the rule, the graph union of  $\mathcal{R}_O$  and  $\mathcal{R}_D$ , denoted by  $\mathcal{R}_{OD} = \mathcal{R}_O \cup \mathcal{R}_D$ , has to be a subgraph of  $\mathcal{R}_{t+1}$ , because the derived constraint is always added to the hyperedge corresponding to the string/mixed operation. One can see that, for each rule in Table 2,  $\mathcal{R}_O \models_{X_O} \mathcal{R}_D$ ; we conclude that  $\mathcal{R}_{OD} \equiv_{X_O} \mathcal{R}_O$  and, hence, they have to be equisatisfiable. The same reasoning can be applied when considering  $\mathcal{R}_t$  and  $\mathcal{R}_{t+1}$ , because  $\mathcal{R}_O \subseteq \mathcal{R}_t$ ,  $\mathcal{R}_{OD} \subseteq \mathcal{R}_{t+1}$  and  $\mathcal{R}_{t+1} = \mathcal{R}_t \cup \mathcal{R}_D$ .

Let us consider the first rule in Table 2 as an example.  $\mathcal{R}_O$  is the network representing the constraint  $c_O \equiv A.\text{contains}(B)$ ;  $\mathcal{R}_D$  is the network corresponding to the derived constraint  $c_D \equiv l_A \geq l_B$ . In this case, we have that  $\mathcal{R}_O \models_I \mathcal{R}_D$ , with  $I = \{A, B\}$ . In other words, for a string  $B$  to be contained in  $A$ , it is necessary but not sufficient that  $l_A \geq l_B$ ; hence, all solutions for  $c_O$  also satisfy  $c_D$  with respect to the variables in  $I$ . Therefore, the network  $\mathcal{R}_{OD}$  that corresponds to the conjunction  $c_O \wedge c_D$  has to be equivalent to  $\mathcal{R}_O$  with respect to the variables in  $I$ ; thus,  $\mathcal{R}_{OD}$  has to be equisatisfiable to  $\mathcal{R}_O$ .

### Constraint refinement

This procedure replaces or removes hyperedges/nodes in a constraint network  $\mathcal{R}_t$ , yielding a new network  $\mathcal{R}_{t+1}$ . All the rules in Table 4 apply change that preserves the relation  $\mathcal{R}_t \equiv_I \mathcal{R}_{t+1}$ , with  $I = X_t \cap X_{t+1}$  being the set of non-redundant variables, i.e., the remaining set of variables not affected by the application of a refinement rule.

In the following, we show how each rule preserves the equisatisfiability between  $\mathcal{R}_t$  and  $\mathcal{R}_{t+1}$ .

Rule 1 and 2 in Table 4 replace hyperedges that involve only constant parameters, with a constant node representing the output of the operation that corresponds to the replaced hyperedge. Since the parameters of these operations are

constant, the output of the transformational/relational operation is also constant; hence its replacement does not affect the equisatisfiability between  $\mathcal{R}_t$  and  $\mathcal{R}_{t+1}$ .

Rule 3 performs only hyperedge replacement without affecting the variables. The application of this rule to  $\mathcal{R}_t$  results in an equivalent network  $\mathcal{R}_{t+1}$  because no variable is added or removed, and because both the replaced hyperedges and their replacement hyperedge are logically equivalent. For example, let  $\mathcal{R}_t$  represent the constraint  $c_t \equiv X \geq Y \wedge X > Y$ , and  $\mathcal{R}_{t+1}$  represent the refined constraint  $c_{t+1} \equiv X > Y$ ;  $\mathcal{R}_t$  is equivalent (and thus equisatisfiable) to  $\mathcal{R}_{t+1}$  because  $c_t$  and  $c_{t+1}$  are logically equivalent and defined over the same set of variables.

Rules 4, 5 and 7 are only applied to equality constraints in  $\mathcal{R}_t$  in order to collapse equivalent nodes and hyperedges. Because equality is an equivalence relation, the application of these rules preserves equisatisfiability between  $\mathcal{R}_t$  and  $\mathcal{R}_{t+1}$ .

Rule 6 is a simple inconsistency check for detecting trivially inconsistent constraints and does not perform any change on the constraint network  $\mathcal{R}_t$ .

## 7 HYBRID CONSTRAINT SOLVING

The constraint network resulting from the previous preprocessing step is then solved by our *hybrid* constraint solver. Our solver is hybrid because it orchestrates a constraint solving procedure for string/mixed and integer constraints with our search-based constraint solving procedure [7].

The idea behind this hybrid solving strategy is to solve a constraint network through a two-stage process: in the first stage, our solving procedure solves all the constraints with supported operations, providing a unified treatment for string and integer constraints. In the second stage, we use our search-driven solving procedure based on the Ant Colony Optimization meta-heuristic to solve the remaining constraints which contain unsupported operations. The solver in the first stage is used to reduce (possibly in a significant way) the search space, i.e., the domains of the string and integer variables, for the search-driven solving procedure; hence, it makes the search in the second stage more scalable and effective.

The pseudocode of our string constraint solving algorithm CSTRSOLVE is shown in Figure 8. It takes as input a hypergraph  $H$  corresponding to the constraint network to solve and returns whether it is satisfiable, unsatisfiable,



or whether it timed out; when it returns satisfiable, it also returns the set of solutions found, which are used to build the vulnerability report.

First, it computes (line 4) the set  $HS$  of connected hyper-subgraphs of  $H$  by means of function `GETMCHYPERSUBGRAPHS`. Then, it iterates through all the elements  $H_i \in HS$  to perform the following steps (lines 5–13).

Function `SOLVESUPPORTEDOPS` (line 6) solves the constraints in  $H_i$  containing supported operations and returns the set  $Sol$ , which contains the solutions for all the string and integer variables in  $H_i$ ; the details of `SOLVESUPPORTEDOPS` are presented in § 7.1.

Subsequently, if  $H_i$  contains any constraints containing unsupported operations, the algorithm invokes the `SEARCHSOLVE` function, which implements a meta-heuristic search algorithm (detailed in § 7.2). `SEARCHSOLVE` tries to find a solution for the constraints which could not be solved by `SOLVESUPPORTEDOPS` due to the presence of unsupported string or integer operations and, thus, provides a general mechanism for solving them. `SEARCHSOLVE` returns a flag *solved* and an updated set of solutions  $Sol$  (line 8). If the flag *solved* is true, it means that the constraints in  $H_i$  have been solved and the algorithm can proceed to process the hypersubgraph  $H_{i+1}$ ; otherwise, it means that the `SEARCHSOLVE` function timed out and, thus, the algorithm returns `TIMEOUT`, terminating the entire constraint solving procedure. A time-out can indicate either that a solution exists for the constraint but the solver could not find it, or that the constraint is actually unsatisfiable.

The algorithm returns SAT and the set of solutions  $Sol$  (line 14) only when the loop over  $H$  has been completely executed, meaning that the attack condition corresponding to the constraint network given as input is satisfiable.

*Representation of the solutions.* The search-based constraint solving step requires every variable domain to be represented in the form of a solution automaton. A solution automaton is an FSM accepting the language determined by the constraints imposed on the variable; in our case, a solution automaton for a variable accepts only the language corresponding to the set of values (for the variable) that satisfy the constraints the solver has solved so far. Hence, the set  $Sol$  computed at lines 6 and 8 contains solution automata. Notice that we provide a unified treatment of integer and string constraints by converting integer ranges into their automaton representation. For example, the solution automaton for a string variable  $s$  involved in a constraint  $s.matches("abc*")$  would be `"abc*"`; the solution automaton for an integer variable  $i$  involved in the constraint  $i > 2$  is `[3-9] | [1-9][0-9]+`, corresponding to the range  $[3, \infty)$ . For a variable involved only in constraints with unsupported operations, its corresponding solution automaton is the default one, accepting any value (i.e., the automaton accepting the regular language `0 | (-?[1-9][0-9]*)` for an integer variable and `.*` for a string variable).

## 7.1 Solving Supported Operations

Our solving procedure leverages automata-based solving for solving string/mixed constraints, and linear interval

```

1: function CSTRSOLVE(Hypergraph  $H$ )
2:   Boolean solved  $\leftarrow$  false
3:   Set of Solution  $Sol \leftarrow \emptyset$ 
4:   Set of Hypergraph  $HS \leftarrow$  GETMCHYPERSUBGRAPHS( $H$ )
5:   for all  $H_i \in HS$  do
6:      $Sol \leftarrow$  SOLVESUPPORTEDOPS( $H_i, Sol$ )
7:     if CONTAINSUNSUPPORTEDOPS( $H_i$ ) then
8:       (solved,  $Sol$ )  $\leftarrow$  SEARCHSOLVE( $H_i, Sol$ )
9:       if  $\neg$ solved then
10:         return TIMEOUT
11:       end if
12:     end if
13:   end for
14:   return (SAT,  $Sol$ )
15: end function

16: function SOLVESUPPORTEDOPS(Hypergraph  $H$ , Set of Solution  $Sol$ )
17:    $Sol \leftarrow$  INITIALIZE( $Sol$ )
18:   Set of Hyperedge  $Worg \leftarrow W \leftarrow$  GETSUPPORTEDEDGES( $H$ )
19:   repeat
20:     Hyperedge  $e \leftarrow$  SELECTEDGE( $W$ )
21:     Set of Solution  $newSol \leftarrow$  APPLYRECIPE( $e, Sol$ )
22:      $Sol \leftarrow$  UPDATE( $newSol, Sol$ )
23:      $W \leftarrow$  UPDATEREWORKLIST( $newSol, Worg, W$ )
24:      $W \leftarrow W \setminus \{e\}$ 
25:   until ISEMPTY( $W$ )
26:   return  $Sol$ 
27: end function

```

Figure 8. Hybrid Constraint solving algorithm

arithmetic for solving integer constraints<sup>5</sup>. In both cases, constraint solving rules are expressed using *recipes* that model the semantics of the operations.

We specifically use automata-based solving and linear interval arithmetic (vs. bit-vector-based or word-based methods) because both methods return, when successful, a solution range for each variable occurring in the constraints they could solve, based on the operations they support.

The pseudocode of our `SOLVESUPPORTEDOPS` algorithm for solving constraints with supported operations is shown in Figure 8; it takes as input a hypergraph  $H$  and set of solutions  $Sol$ . First, the algorithm initializes each variable in set  $Sol$  with a default solution automaton, which is `.*` for string variables and `0 | (-?[1-9][0-9]*)` for integer variables (line 17), and extracts from  $H$  the set of hyperedges  $W$  (and its copy  $Worg$ ) labeled with supported operations (line 18). Next, the constraints are solved using a worklist procedure (lines 19–25). First, a hyperedge  $e$  is selected from  $W$  (line 20). Then the `APPLYRECIPE` procedure (line 21) processes the domains of the variables in the component nodes of  $e$  according to the recipe that models the semantics of the operation labeling  $e$ . We provide recipes (see Table 5 and Table 6) for both string and integer variables: string/mixed constraints are solved by means of automaton-operations (such as union, intersection, concatenation), and integer constraints are solved through linear interval arithmetic (where integer domains are represented by intervals).

As a result, `SOLVESUPPORTEDOPS` returns the set  $newSol$  (line 21), in which the solution automata of the variables in the component nodes of  $e$  accept the languages corresponding to the sets of values that satisfy the string/mixed or integer constraints represented by  $e$ . The set  $Sol$  is then updated with the solutions in  $newSol$  (line 22). If the solution automaton of a variable  $v$  is affected by this update, it needs

5. Integer constraints that cannot be solved with linear interval arithmetic are treated as constraints with unsupported operations.

to be checked for satisfiability with other constraints that involve  $v$ ; in this case, the function `UPDATEWORKLIST` puts back into  $W$  all the previously-removed hyperedges (determined from the original list *Worg*) which are connected to the node for variable  $v$  (line 23). Finally, the algorithm removes the hyperedge  $e$  from the set  $W$ . This worklist procedure is repeated until the set  $W$  is empty, i.e., until all the constraints with supported operations have been successfully solved. Notice that when a recipe results in an automaton that accepts the empty language, meaning that there is no solution to satisfy a constraint, the entire constraint solving procedure terminates internally and returns UNSAT.

The above worklist procedure implements the fixpoint iteration based on an arc-consistency algorithm [34]. A hyperedge is added back to the worklist (through the `UPDATEWORKLIST` operation) only if the domains of its variables could be further restricted through the application of a recipe. Since all the recipes in Table 5 and Table 6 are of the form  $A := A \cap B$ , i.e., they only restrict the domains of the involved variables, the worklist procedure is guaranteed to converge and to terminate with two possible results: (1) one variable domain becomes empty or (2) the constraint network stabilizes because none of the variable domains can be further restricted.

In the following subsections, we discuss the constraints that are supported by our automata-based+linear interval arithmetic solver and illustrate the worklist procedure with an application to the running example.

### 7.1.1 Supported String and Mixed Constraints

Our automata-based solver supports many operations from the `java.lang.String` class, except methods like `format` and `hashCode` that cannot be expressed in terms of FSMs. The solver also supports the `matches`, `parseInt`, `parseLong`, `toString`, `valueOf`, `append`, and `length` operations from other Java classes that are commonly found in Java applications [35]. In addition, the solver supports 16 input sanitization operations from the Apache Commons Lang 3 [9] and OWASP [10] standard security libraries; we remark that in the context of security analysis it is important to support such operations for achieving precise and efficient analysis.

Table 5 shows a subset of operations supported by our automata-based solver and the corresponding recipes<sup>6</sup>; the operations marked with a star are proposed for the first time in this paper, while the others have been taken from [3], [29]. In the table we adopt the following notation:  $M_X$  denotes the automaton accepting the language  $L_X = \{X\}$ , containing only the string  $X$ ;  $M_\varepsilon$  denotes the automaton that accepts the empty string;  $\cdot^i$  denotes the automaton that accepts  $i$  number of any characters;  $*$  denotes the Kleene-star operator;  $\neg$  denotes the complement operation;  $\oplus$  denotes concatenation;  $M_X \cap := M_Y$  is a short-hand for  $M_X := M_X \cap M_Y$ ;  $[c_0 c_1 \dots c_n]$  is a short-hand for  $c_0 \cup c_1 \cup \dots \cup c_n$  where  $c$  stands for a character or a character range. In addition, we use a number of auxiliary operations that work on automata: *Prefix*( $M$ ) and *Suffix*( $M$ ) return the prefixes and suffixes of the words accepted by  $M$ , respectively; *Substring*( $M$ ) returns an automaton that accepts

the substrings of the words accepted by  $M$ ; *Substring*( $M, i$ ) returns an automaton that accepts the substrings starting from the index  $i$  of the words accepted by  $M$ ; *Trim*( $M$ ) returns an automaton that accepts all words of  $M$  without leading and trailing blanks; *CharAt*( $M, i$ ) returns an automaton that accepts the characters at the index  $i$  of the words accepted by  $M$ ; *LowerCase*( $M$ ) and *UpperCase*( $M$ ) return an automaton that accepts the lowercase (respectively, uppercase) words of the words accepted by  $M$ ; *Replace*( $M_X, c_1, c_2, M_{X'}$ ), *ReplaceFirst*( $M_X, c_1, c_2, M_{X'}$ ), and *ReplaceAll*( $M_X, c_1, c_2, M_{X'}$ ) are functions defined in [3], [36] modeling the homonymous replacement string operations, which return a tuple of solution automata  $M_X$  and  $M_{X'}$ , where  $M_{X'}$  accepts the words resulting from the semantics of the corresponding replacement operation; *ApacheSanitizeHTML3*( $M_X, M_{X'}$ ) returns a tuple of solution automata  $M_X$  and  $M_{X'}$ , where  $M_{X'}$  accepts the words resulting after replacing all occurrences of HTML meta-characters in the words accepted by  $M_X$  with their corresponding escape characters. The latter models the semantics of a standard sanitization function from Apache [9], through a series of *Replace* functions for (meta-character, escape character) pairs; we model the remaining 15 standard sanitization functions we support in a similar way.

### 7.1.2 Supported Integer Constraints

Integer constraints are solved by means of linear interval arithmetic. When successful, this method returns a range of solutions for each integer variable involved in the constraints.

Our solver supports basic operations of the form  $I \text{ op } K$ , where  $I, K$  are integer variables and  $\text{op} \in \{=, >, <, \geq, \leq, +, -\}$ . From our experience with the test subjects in our experiments<sup>7</sup> this set of supported operations is enough, since most of the constraints encountered when analyzing the injection vulnerabilities of Web applications are linear integer constraints; more details are provided in section 9.

Table 6 shows the recipes for various integer operators *op*; in this case, a recipe is a sequence of operations that are executed to compute a range of solutions for each integer variable according to the semantics of *op*. These recipes have been taken from the sources ([37], [38], [39]) indicated in the last column of Table 6. In the table, we adopt the following notation:  $I_V$  denotes the interval of variable  $V$  with  $v_{\min}$  and  $v_{\max}$  being its lower and upper bounds, respectively;  $I_K \cap := I_N$  is a short-hand for  $I_K := I_K \cap I_N$ , i.e., the intersection of the two intervals  $I_K$  and  $I_N$ .

As discussed above, we convert integer ranges into their automaton representation. More specifically, for each integer variable  $V$ , we generate an automaton representing its interval  $I_V$  with  $I_V := [v_{\min}, v_{\max}]$ , i.e., the automaton that accepts only the string representation of numbers within  $I_V$ . We convert intervals by intersecting the automata for  $[v_{\min}, \infty)$  and  $(-\infty, v_{\max}]$ . For example, the integer interval  $I_V = [3, 45]$  can be represented as  $[3-9] \mid [1-9] [0-9]^+ \cap 0 \mid [1-9] [0-9]^* \mid 4 [0-5] \mid [1-3] [0-9] \mid [0-9]$ .

7. All of the numeric constraints that appeared in our test subjects could be solved by means of the recipes listed in Table 6.

6. The full list of supported operation is available online [15].

Table 5

Automata operations (recipes) corresponding to string/mixed operations. ( $X$  and  $Y$  are string variables,  $X'$  is an auxiliary variable,  $c$ ,  $c_1$  and  $c_2$  are string constants,  $i$ ,  $i_1$  and  $i_2$  are integer constants)

Operation	Recipe	Source
$X.\text{charAt}(i)$	$M_X \cap := .^i \oplus M_{X'} \oplus .^*$ $M_{X'} \cap := \text{CharAt}(M_X, i)$	[29]
$X.\text{concat}(Y)$	$M_X \cap := \text{Prefix}(M_{X'})$	[29]
$X.\text{append}(Y)$	$M_Y \cap := \text{Suffix}(M_{X'})$	[29]
$X+Y$	$M_{X'} \cap := M_X \oplus M_Y$	*
$X.\text{contains}(Y)$	$M_X \cap := .^* \oplus M_Y \oplus .^*$ $M_Y \cap := \text{Substring}(M_X)$	[29]
$\text{copyValueOf}(X)$ $\text{valueOf}(X)$ $X.\text{toString}()$	$M_X \cap := M_{X'}$ $M_{X'} \cap := M_X$	*
$X.\text{endsWith}(Y)$	$M_X \cap := .^* \oplus M_Y$ $M_Y \cap := \text{Suffix}(M_X)$	[29]
$X.\text{equals}(Y)$ $X.\text{contentEquals}(Y)$	$M_X \cap := M_Y$ $M_Y \cap := M_X$	[29]
$X.\text{equalsIgnoreCase}(Y)$	$M_X \cap := \text{LowerCase}(M_Y) \cup \text{UpperCase}(M_Y)$ $M_Y \cap := \text{LowerCase}(M_X) \cup \text{UpperCase}(M_X)$	*
$\text{escapeHtml3}(X)$	$\langle M_X, M_{X'} \rangle := \text{ApacheSanitizeHTML3}(M_X, M_{X'})$	*
$X.\text{indexOf}(Y)$	$M_X \cap := \{.^i \cap \neg M_Y\} \oplus M_Y \oplus .^*$ $M_Y \cap := \text{Substring}(M_X, i)$	[29]
$X.\text{isEmpty}()$	$M_X \cap := M_\epsilon$	*
$X.\text{lastIndexOf}(Y)$	$M_X \cap := (.^* \oplus M_Y \oplus .^*) \cap \{.^{i+1} \oplus (.^* \cap \neg M_Y)\}$ $M_Y \cap := \text{Substring}(M_X, i)$	[29]
$X.\text{length}()$	$M_X \cap := .\{i_1, i_2\}$ where $i_1$ and $i_2$ are equal to the lower limit and the upper limit, respectively, of the length of $X$ . If the upper limit is $\infty$ , $i_2$ is removed from the expression, i.e., $.\{i_1, \}$ . If $X$ has a fixed length, the expression is $.\{i_1\}$ .	*
$X.\text{matches}(c)$	$M_X \cap := M_c$	[3]
$\text{parseInt}(X)$ $\text{parseLong}(X)$	$M_X \cap := M_{X'} \cap (\emptyset \cup (-\{0, 1\}[1-9][0-9]^*))$	*
$X.\text{replace}(c_1, c_2)$	$\langle M_X, M_{X'} \rangle := \text{Replace}(M_X, c_1, c_2, M_{X'})$	[3]
$X.\text{replaceAll}(c_1, c_2)$	$\langle M_X, M_{X'} \rangle := \text{ReplaceAll}(M_X, c_1, c_2, M_{X'})$	[3]
$X.\text{replaceFirst}(c_1, c_2)$	$\langle M_X, M_{X'} \rangle := \text{ReplaceFirst}(M_X, c_1, c_2, M_{X'})$	[3]
$X.\text{startsWith}(Y)$	$M_X \cap := M_Y \oplus .^*$ $M_Y \cap := \text{Prefix}(M_X)$	[29]
$X.\text{substring}(i)$	$M_X \cap := .^{i-1} \oplus M_{X'}$ $M_{X'} \cap := \text{Substring}(M_X, i)$	[3], [29]
$X.\text{substring}(i_1, i_2)$	$M_X \cap := .^{i_1} \oplus M_{X'} \oplus .^*$ $M_{X'} \cap := \text{Substring}(M_X, i_1, i_2)$	[3], [29]
$X.\text{toLowerCase}()$	$M_{X'} \cap := \text{LowerCase}(M_X)$ $M_X \cap := M_{X'} \cup \text{UpperCase}(M_{X'})$	*
$X.\text{toUpperCase}()$	$M_{X'} \cap := \text{UpperCase}(M_X)$ $M_X \cap := M_{X'} \cup \text{LowerCase}(M_{X'})$	*
$X.\text{trim}()$	$M_X \cap := []^* \oplus M_{X'} \oplus []^*$ $M_{X'} \cap := \text{Trim}(M_X)$	[29]

### 7.1.3 Application to the Running Example

We now show an example of the execution of SOLVESUPPORTEDOPS when solving the hypersubgraph  $H_3$  in Figure 7, which contains string/mixed, and integer constraints. The solution automata for the two string variables  $SID$  and  $SID'$  are initialized with  $.*$ ; the solution automaton for the integer variable  $l_{SID}$  is initialized with  $\emptyset | (-? [1-9] [0-9]^*)$ . First, the procedure GETSUPPORTEDEDGES returns  $W$  (fur-

ther assigned also to  $W_{org}$ ), which contains four hyperedges, labeled with  $>$ ,  $\geq$ ,  $\text{length}()$  and  $\text{matches}()$ .

Let us assume that in the first iteration of the worklist loop, function SELECTEDGE selects the hyperedge labeled with  $\geq$ . This hyperedge has two component nodes: the integer variable node  $l_{SID}$  and the constant node 0. The solution for  $l_{SID}$  is solved by applying the recipe given in row 4 of Table 6, which yields  $l_{SID} := [0, \infty)$ . The

Table 6

Interval operations (recipes) corresponding to integer constraints ( $L$ ,  $N$  and  $K$  are integer variables,  $I_V$  stands for the interval of variable  $V$  with  $I_V := [v_{\min}, v_{\max}]$  where  $v_{\min}$  and  $v_{\max}$  denote lower and upper bounds, respectively).

Operation	Recipe	Source
$K = N$	$I_K \cap := I_N$ $I_N \cap := I_K$	[37], [38]
$K > N$	$I_{>N} := [n_{\min} + 1, \infty)$ is the interval that captures all numbers greater than $n_{\min}$ $I_{<K} := (-\infty, k_{\max} - 1]$ is the interval that captures all numbers smaller than $k_{\max}$ $I_N \cap := I_{<K}$ $I_K \cap := I_{>N}$	[38]
$K \geq N$	$I_{\geq N} := [n_{\min}, \infty)$ is the interval that captures all numbers greater than or equals to $n_{\min}$ $I_{\leq K} := (-\infty, k_{\max}]$ is the interval that captures all numbers smaller than or equals to $k_{\max}$ $I_N \cap := I_{\leq K}$ $I_K \cap := I_{\geq N}$	[38]
$L = K + N$	$I_L \cap := [k_{\min} + n_{\min}, k_{\max} + n_{\max}]$ $I_K \cap := [l_{\min} - n_{\max}, l_{\max} - n_{\min}]$ $I_N \cap := [l_{\min} - k_{\max}, l_{\max} - k_{\min}]$	[37], [38], [39]
$L = K - N$	$I_L \cap := [k_{\min} - n_{\max}, k_{\max} - n_{\min}]$ $I_K \cap := [l_{\min} + n_{\min}, l_{\max} + n_{\max}]$ $I_N \cap := [k_{\min} - l_{\max}, k_{\max} - l_{\min}]$	[37], [38], [39]

hyperedge  $\geq$  is then removed from the worklist  $W$  and the worklist loop will continue to the next iteration; let us assume that the hyperedge labeled with  $>$  is selected in the latter. Similarly to the previous iteration, the *new* solution for  $l_{SID}$  is obtained by applying the recipe given in row 3 of Table 6, which yields  $l_{SID} := [21, \infty)$ . Since this results in a change to the existing solution automaton of  $l_{SID}$  in *Sol*, the previously-removed edge  $\geq$ , which is connected to the variable node  $l_{SID}$ , is put back into the worklist  $W$ ; the hyperedge  $>$  is also removed from  $W$ . Let us assume that the hyperedge labeled with  $\geq$  is selected in the next iteration; by applying the recipe given in row 4 of Table 6, the solution for  $l_{SID}$  is  $l_{SID} := [21, \infty)$  which is converted to the automaton  $2 [1-9] [3-9] [0-9]^+ [1-9] [0-9] [0-9]^+$ . Since this results in no change to any existing solution automaton in *Sol*, no previously-removed edge is put back into  $W$ ; the hyperedge  $\geq$  is then removed from  $W$ . Subsequently, let us assume that the hyperedge labeled with *matches()* is selected. This hyperedge has two component nodes: the string variable node  $SID'$  and the constant node `.*['\n'=<>/,;+-&*[]].*`, which is equivalent to the regular expression `.*['\n'=<>/,;+-&*[]].*`. The solution automaton  $M_{SID'}$  for the variable  $SID'$  is computed according to the recipe for the operation *matches()* in Table 5:

$$\begin{aligned}
 M_{SID'} &:= M_{SID'} \cap .*['\n'=<>/,;+-&*[]].* \leftrightarrow \\
 M_{SID'} &:= .* \cap .*['\n'=<>/,;+-&*[]].* \leftrightarrow \\
 M_{SID'} &:= .*['\n'=<>/,;+-&*[]].*
 \end{aligned}$$

This means that the language accepted by  $M_{SID'}$  is now restricted to the values matching the regular expression `.*['\n'=<>/,;+-&*[]].*`. This results in a change to the existing solution automaton of  $SID'$  in *Sol* (recall that it was initialized with `.*`). The procedure is supposed to put back any previously-removed hyperedge that is connected to node  $SID'$ ; however, in this case there is no such edge. The hyperedge *matches()* is then removed from  $W$  and the worklist loop continues to the next iteration, in which

the remaining hyperedge *length()* is selected. The solution automaton  $M_{SID}$  is then updated according to the recipe for operation *length()* in Table 5, with  $i_1 = 21$ ,  $i_2 = \infty$ :

$$\begin{aligned}
 M_{SID} &:= M_{SID} \cap .\{21, \infty\} \leftrightarrow M_{SID} := .* \cap .\{21, \infty\} \leftrightarrow \\
 M_{SID} &:= .\{21, \infty\}
 \end{aligned}$$

This means that the language accepted by  $M_{SID}$  is now restricted to the strings with a length greater than or equal to 21. This results in a change to the existing solution automaton of  $M_{SID}$  in *Sol*. Again, there is no previously-removed hyperedge connected to  $M_{SID}$  that has to be put back in  $W$ . After this iteration, the worklist is empty and the algorithm returns the set  $Sol = \{l_{SID}, M_{SID}, M_{SID'}\}$ , where  $l_{SID} := 2 [1-9] [3-9] [0-9]^+ [1-9] [0-9] [0-9]^+$ ,  $M_{SID} := .\{21, \infty\}$ , and  $M_{SID'} := .*['\n'=<>/,;+-&*[]].*$

Notice that the hyperedge labeled with *customSanit()* is not in the worklist since it reflects an unsupported operation.

## 7.2 Solving Unsupported Operations

We use the Ant Colony Optimization (ACO) meta-heuristic search for solving constraints that involve unsupported operations. We chose ACO over other well-known meta-heuristic search techniques (such as hill climbing, simulated annealing, and genetic algorithms [40]) because:

- It is typically used for finding good solutions (i.e., paths that return good fitness values) in graphs [41]. Hence, it can be easily adapted to our problem where the search space is defined in a graph form, i.e., an automaton.
- Differently from other search algorithms, in ACO, defining a new candidate solution is straightforward, since it only requires having an ant exploring a new path in the solution automaton.
- It has inherent parallelism in which multiple candidate solutions can be searched in parallel for efficiency.
- It is stochastic in nature, which prevents the search from getting stuck in local optima.

Several algorithms that implement the ACO meta-heuristic have been proposed in the literature. In this paper we will use *MAX-MIN* Ant System [42] with 2-opt local search [43], in which the pheromone values are bounded by maximum and minimum values, which are dynamically computed after every search iteration. We use this algorithm because the bounding of the pheromone values prevents their relative difference from becoming too extreme during the run of the algorithm and, therefore, mitigates the search stagnation problem in which ants traverse the same trails and construct the same solutions over and over again.

Below, we first present the fitness functions used within the algorithm and then the algorithm itself.

### 7.2.1 Fitness Functions

Any search-based procedure requires defining one or more fitness functions to assess the quality of the potential solutions, i.e., their distance from the best solution. A low(er) value for the fitness of a solution implies a high(er) quality for the solution itself. Since in the context of this work we deal with both integer and string constraints, we use fitness functions specific to these domains.

For integer constraints we use the Korel function [44], which is a standard fitness function for this domain. We consider constraints of the form  $C \equiv E_1 \bowtie E_2$ , where  $\bowtie \in \{=, <, \leq, >, \geq\}$  and  $E_1, E_2$  are integer expressions that can be integer variables, integer constants, or any other expression whose evaluation results in an integer value (e.g., the length operation for strings); notice that we treat boolean expressions also as integer expressions. Let  $s = [s_1, \dots, s_n]$  be the vector of candidate solutions for the integer variables  $x_1, \dots, x_n$  in  $C$ , and  $a(s), b(s)$  be the integer values resulting from the evaluations of  $E_1$  and  $E_2$  respectively, after replacing the variables in them with the corresponding solutions in  $s$ . The fitness of  $s$  is defined as:

$$f(s) = \begin{cases} 0, & a(s) \bowtie b(s) \text{ is true;} \\ |a(s) - b(s)| + k, & a(s) \bowtie b(s) \text{ is false;} \end{cases} \quad (7.1)$$

where  $k = 0$  when  $\bowtie \in \{=, \leq, \geq\}$  and  $k = 1$  otherwise.

For string constraints we use two different functions, depending on the operations in which string variables are involved: the Levenshtein (edit) distance function [45] and the equality cost function for regular expression matching [46]; both functions have been shown to be useful for search-based generation of string values [46]. The Levenshtein distance between two strings  $a$  and  $b$  is defined as the minimum number of insert, delete, and substitute operations (of characters) needed to convert  $a$  into  $b$ . The regular expression matching function between a string  $a$  and a regular expression  $b$  is defined as the minimum Levenshtein distance among  $a$  and the strings belonging to the regular language defined by  $b$ . We consider string constraints of the form  $C \equiv E_1 \bowtie E_2$ , where  $\bowtie$  is a string operation returning a boolean result, and  $E_1, E_2$  are string expressions that can be string variables, string literals, or any other expression whose evaluation results in a string value (e.g., the concat operation for two strings). Let  $s = [s_1, \dots, s_n]$  be the vector of candidate solutions for the string variables  $x_1, \dots, x_n$  in  $C$ , and  $a(s), b(s)$  be the string values resulting from the evaluations of  $E_1$  and  $E_2$  respectively, after replacing the

variables in them with the corresponding solutions in  $s$ . The fitness of  $s$  is defined as:

$$f(s) = \begin{cases} 0, & a(s) \bowtie b(s) \text{ is true;} \\ \psi(a(s), b(s)), & a(s) \bowtie b(s) \text{ is false;} \end{cases} \quad (7.2)$$

where  $\psi$  is:

- the equality cost function for regular expression matching, when  $\bowtie$  is a regular expression-based string matching operation (e.g., the matches operation for strings in Java);
- the Levenshtein distance, in all other cases for  $\bowtie$ .

We assume to have a list of operations classified as regular expression-based string matching operations; if there is an unknown regular expression-based matching operation, it will be treated as a generic case, using the Levenshtein distance function. For both types of constraints, the fitness of a candidate solution is set to an arbitrarily selected large value (such as 1000) when the solution leads to an exception during the evaluation of the expressions in which it is used.

We now show the application of these fitness functions in the context of solving the constraints used in the attack condition *SEC1* from our running example (see section 5). *SEC1* contains two integer constraints, `Integer.parseInt(MAX) > 20` and `SID.length() > 20`, one string constraint with a regular expression-based string matching operation, `customSanit(SID).matches(".*['\"=<>/,;+&*\[\] ].*")`, and one constraint with a generic string operation, `OP.trim().equalsIgnoreCase("GradeQuery")`.

Let us consider the case in which the search algorithm has returned the following candidate solutions for the variables involved in *SEC1*:  $MAX := 10$ ,  $OP := \text{Grade}$ , and  $SID := \text{aRXxQ1zCVmaetowbnZv0t}$ ; the fitness function for these solutions is computed as follows. For constraint `Integer.parseInt(MAX) > 20`, we apply Eq. 7.1; since the evaluation of the constraint after replacing the variable with the candidate solution is false, we get  $f(MAX := 10) = |10 - 20| + 1 = 11$ . For constraint `SID.length() > 20`, we get  $f(SID := \text{aRXxQ1zCVmaetowbnZv0t}) = 0$  because the solution for *SID* satisfies the constraint. For constraint `customSanit(SID).matches(".*['\"=<>/,;+&*\[\] ].*")`, we apply Eq. 7.2; since the evaluation of the constraint after replacing the variable with the candidate solution is false (because the string `aRXxQ1zCVmaetowbnZv0t` does not match the regular expression `.*['\"=<>/,;+&*\[\] ].*`), we get  $f(SID := \text{aRXxQ1zCVmaetowbnZv0t}) = \psi(\text{aRXxQ1zCVmaetowbnZv0t}, \text{.*['\"=<>/,;+&*\[\] ].*}) = 1$ . In this case,  $\psi$  is the equality cost function for regular expression matching; a fitness value equal to 1 means that at least one character operation is needed to convert the candidate solution to a string belonging to the regular language defined by the given regular expression. Following a similar process, the fitness of the candidate solution for *OP* in the constraint `OP.trim().equalsIgnoreCase("GradeQuery")` is computed as  $f(OP := \text{Grade}) = \psi(\text{Grade}, \text{GradeQuery}) = 5$ , where  $\psi$  is the Levenshtein distance function.

```

1: function SEARCHSOLVE(Hypergraph  $H$ , Set of Solution  $Sol$ )
2:   Set of Solution-automaton  $K \leftarrow \text{GETSOLAUTOMATA}(Sol)$ 
3:   Tuning-parameters  $\langle \alpha, \beta, \rho, \xi_{max}, \xi_{min} \rangle \leftarrow \text{SETTUNINGPARAMS}()$ 
4:   Population-size  $A \leftarrow \text{SETNUMBERANTS}()$ 
5:   Set of Desirability-value  $\Delta \leftarrow \text{SETDESIRABILITYVAL}(K)$ 
6:   Set of Pheromone  $\Xi \leftarrow \text{SETPHEROMONES}(K)$ 
7:   Set of Solution-component  $T_{Best} \leftarrow \emptyset$ 
8:   Fitness  $F_{Best} \leftarrow 1$ ; Fitness  $F_{pBest} \leftarrow 1$ 
9:   Array of Fitness  $tempF \leftarrow \emptyset$ 
10:  Array of Set of Solution-component  $tempT \leftarrow \emptyset$ 
11:  repeat
12:    loop  $A$  times
13:      Set of Solution-component  $T \leftarrow \text{CONSTRUCTSOLUTIONS}(K, \Delta, \Xi)$ 
14:      Fitness  $F \leftarrow \text{COMPUTEFITNESS}(T, H)$ 
15:       $tempF \leftarrow \text{APPEND}(tempF, F)$ ;  $tempT \leftarrow \text{APPEND}(tempT, T)$ 
16:    end loop
17:     $\langle F_{Best}, T_{Best} \rangle \leftarrow \text{BESTSOLUTION}(tempF, tempT)$ 
18:    if  $F_{Best} < F_{pBest}$  then
19:       $\langle F_{Best}, T_{Best} \rangle \leftarrow \text{2OPTLOCALSEARCH}(K, T_{Best})$ 
20:    end if
21:     $\text{UPDATEPHEROMONES}(K, \Xi, F_{Best}, T_{Best})$ 
22:     $F_{pBest} \leftarrow F_{Best}$ 
23:    until  $F_{Best} = 0$  or timeout
24:    if timeout then
25:      return  $\emptyset$ 
26:    end if
27:     $Sol \leftarrow \text{UPDATESOL}(T_{Best})$ 
28:    return  $Sol$ 
29: end function

30: function CONSTRUCTSOLUTIONS(Set of Solution-automaton  $K$ ,
                               Set of Desirability-value  $\Delta$ ,
                               Set of Pheromone  $\Xi$ )
31:   Set of Solution-component  $S \leftarrow \emptyset$ 
32:   repeat
33:     Automaton  $k \leftarrow \text{RANDOMSELECT}(K)$ 
34:     FSMState  $v \leftarrow \text{GETSTARTSTATE}(k)$ 
35:     repeat
36:       Set of FSMTransition  $E \leftarrow \text{GETOUTTRANSITIONS}(v)$ 
37:       FSMTransition  $e \leftarrow \text{SELECTTRANSITION}(E, \Delta, \Xi)$ 
38:        $S \leftarrow S \cup \{e\}$ 
39:        $v \leftarrow \text{GETNEXTSTATE}(e)$ 
40:     until  $\text{ISACCEPTSTATE}(v)$ 
41:      $\text{MARKASVISITED}(k, K)$ 
42:   until all the automata in  $K$  have been traversed
43:   return  $S$ 
44: end function

45: function COMPUTEFITNESS(Hypergraph  $H$ ,
                             Set of Solution-component  $T$ )
46:   Set of Constraint  $\Theta \leftarrow \text{GETCONSTRAINTS}(H)$ 
47:    $i \leftarrow 0$ 
48:   for all Constraint  $\theta$  in  $\Theta$  do
49:      $i \leftarrow i + 1$ 
50:     Fitness  $f_i \leftarrow \text{EVALUATE}(\theta, T)$ 
51:     Fitness  $\hat{f}_i \leftarrow \text{NORMALIZE}(f_i)$ 
52:   end for
53:   return Fitness  $F \leftarrow \text{AVERAGE}(\hat{f}_1, \hat{f}_2, \dots, \hat{f}_i)$ 
54: end function

```

Figure 9. Ant colony search for string constraint solving

### 7.2.2 Search Algorithm

The ACO meta-heuristic for solving constraints containing unsupported operations is implemented in function `SEARCHSOLVE`, whose pseudocode is shown in Figure 9. The function takes as input a hypergraph  $H$  and a set of solutions  $Sol$  (as determined by the call to function `SOLVESUPPORTEDOP`).

First, the function retrieves from  $Sol$  the set of solution automata  $K$  for the string input variables (line 2); notice that auxiliary string variables are excluded because the search procedure needs to find solutions only for the input variables. The next steps of function `SEARCHSOLVE` (lines 3–6) initialize the tuning and search parameters as follows (the initialization value is indicated next to each parameter):

- *Tuning parameters*:  $\alpha = 1$  and  $\beta = 1$  determine the relative importance of the pheromone trail and the heuristic-

based desirability information;  $\rho = 0.01$  is the evaporation rate used to prevent the pheromone values from piling up;  $\xi_{max} = 5$  and  $\xi_{min} = 0$  determine the bounds of pheromone values.

- *Search parameters*: the number of ants  $A = 20$ ; the set  $\Delta$  of desirability values  $\delta_e = 1$  for each transition  $e$  of each automaton in  $K$ ; the set  $\Xi$  of pheromone values  $\xi_e = \xi_{max}$  for each transition  $e$  of each automaton in  $K$ .

In ACO, these parameters have to be defined specifically for the target problem; we chose them based on the guidelines provided in [42] and on our own preliminary experiments. Notice that for each transition  $e$ , the parameter  $\xi_e$  is initialized to the value  $\xi_{max}$ ; as discussed in [42], this allows for diverse explorations of the solutions during the first iterations of the algorithm, because of the small, relative differences between the pheromone values of the explored transitions and of the ones not-yet explored.

The algorithm then loops through the three main steps (construction of candidate solutions, application of local search, update of pheromone values) until the termination conditions are met (lines 11–23). We illustrate these steps through the example shown in Figure 10. In this example, the variable  $V$  is involved in three constraints (shown in Figure 10(a)):  $cstr_1 \equiv V.\text{matches}("ab*|ca")$  contains one supported operation;  $cstr_2 \equiv V.\text{length}() \leq 3$  contains two supported operations `length()` and `<=`;  $cstr_3 \equiv \text{custom}(V).\text{equals}("bba")$  contains one supported operation `equals()` and one unsupported operation `custom()`.

**Construction of candidate solutions.** This step (lines 12–17) consists of three sub-steps:

1) *Building the set of solution components.* This step is represented by the call to function `CONSTRUCTSOLUTIONS`, which takes as input the set of solution automata  $K$ , the set of desirability values  $\Delta$ , and the set of pheromones values  $\Xi$ . It outputs a set of solution components; a solution component is a sequence of transitions in a solution automaton as traversed by the procedure.

This function goes through (lines 32–42) the set of automata  $K$ , and at each iteration it randomly selects an automaton  $k \in K$ . Starting from the start state of  $k$ , it traverses the outgoing transitions of the states in  $k^8$ .

Upon reaching a state where there are multiple outgoing transitions, it selects (line 37) one of them (say transition  $e$ ) based on the probability  $P_e = \frac{\xi_e^\alpha \delta_e^\beta}{\sum_{t \in E} \xi_t^\alpha \delta_t^\beta}$ , computed using the pheromone value  $\xi_e$  and the desirability  $\delta_e$  of the transition.

The selected transition is added to the local set of solution components  $S$  (line 38) and its reaching state is retrieved (line 39). The traversal/selection of the transitions of an automaton is repeated until the final state is found<sup>9</sup>, which means that a solution for the variable associated with the current automaton  $k$  has been found. In this case the outer loop moves to explore the next automaton in  $K$ , and continues until all automata in  $K$  have been traversed. At the end,

8. In our automaton representation, a transition reflects a Unicode character; each transition is updated with a new pheromone value during the search iterations to reflect the fitness of the solution that contains the corresponding character.

9. Internally we represent solution automata as generalized non-deterministic finite automata, which have only one final state.

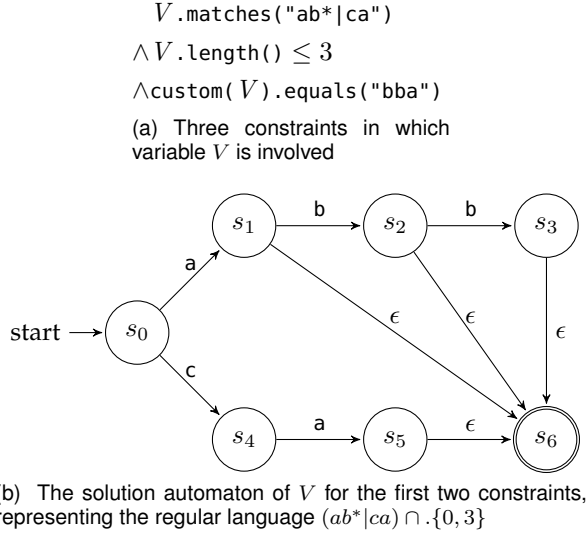


Figure 10. Example to illustrate the search algorithm

the function returns a set of solution components, with one candidate solution for each string variable.

Let us execute this step through the example shown in Figure 10, by assuming that the automaton  $k$  extracted at line 33 is the one shown in Figure 10(b). This is the solution automaton of the input variable  $V$ , as determined after solving the first two constraints with the procedure described in § 7.1; each transition is associated with a character or the empty string  $\epsilon$ . We assume that the transition  $(s_0, s_1)$  has the pheromone value  $\xi_{(s_0, s_1)} = 2$ , while the other transitions have the pheromone value  $\xi = 1$ ; also recall that the desirability value for all the transitions is set to a fixed value of  $\delta = 1$ .

To construct a solution, the procedure starts from the initial state  $s_0$  and then selects one of its outgoing transitions based on the probability  $P_e$ . In this case, the probability of selecting transition  $(s_0, s_4)$  is  $P_{(s_0, s_4)} = (1 * 1)/(1 + 2) = 0.33$ ; the probability of selecting transition  $(s_0, s_1)$  is  $P_{(s_0, s_1)} = (2 * 1)/(1 + 2) = 0.67$ . Let us assume that the transition  $(s_0, s_1)$  is selected and traversed: the procedure reaches state  $s_1$ . It then traverses one of the two outgoing transitions of  $s_1$ ; the probability of selecting transition  $(s_1, s_6)$  is 0.5 and the probability of selecting transition  $(s_1, s_2)$  is 0.5. Assuming that the procedure selects  $(s_1, s_2)$ , it reaches state  $s_2$ . Afterwards, assuming that the procedure selects transition  $(s_2, s_6)$ , it reaches the final state  $s_6$ . This generates the sequence of transitions  $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$ , which represents the candidate solution  $V = "ab"$ .

2) *Determining the fitness of solution components.* This step computes the fitness for the solution components identified in the previous step. Function COMPUTEFITNESS (line 14) takes as input the hypersubgraph  $H$  and the set of solution components  $T$ , which contains a candidate solution for each input variable in  $H$ . It first obtains the set of constraints  $\Theta$  represented by  $H$ . Then, for each constraint  $\theta$  in  $\Theta$  it

invokes<sup>10</sup> function EVALUATE (line 50), which evaluates the expressions in  $\theta$  with the solutions given in  $T$ . More specifically, each unsupported operation in  $\theta$  is invoked with parameters that are retrieved from  $T$ ; the return value is then used to compute the fitness  $f$ , using one of the aforementioned fitness functions (Eq. 7.1 or Eq. 7.2) depending on the type of constraint, as explained in § 7.2.1. To ensure that the search process is not biased towards solving the constraints with larger-scale fitness values, each fitness value  $f$  is normalized (line 51) using the normalization function proposed in [47], resulting in a normalized fitness value  $\hat{f} = f/(f + 1)$ . We use this normalization function since it has been proven to be useful in the similar domain of search-based test input generation of string data types [48]. After computing the fitness for all the constraints in  $H$ , the overall fitness  $F$  of  $T$  is computed by taking the average of individual, normalized fitness values  $\hat{f}$  (line 53).

The execution of this step through the example in Figure 10 works as follows. Recall that the solution identified in the previous step is  $V = "ab"$ . For each constraint, the corresponding expression is evaluated and the fitness of the solution is computed accordingly. For  $cstr_1$ , we apply Eq. 7.2 with the equality cost function for regular expression matching; since the evaluation of the constraint after replacing the variable with the candidate solution is true, we get  $f_{cstr_1} = 0$ . For  $cstr_2$ , we apply the Korel function (Eq. 7.1); since the evaluation of the constraint after replacing the variable with the candidate solution is true, we get  $f_{cstr_2} = 0$ . For constraint  $cstr_3$ , let us assume that the resulting value after executing the operation `custom()` with the input  $V = "ab"$  is "ba"; the fitness, computed by applying the Levenshtein (edit) distance function, is  $f_{cstr_3} = 1$ <sup>11</sup>. By applying the normalization we get:

$$\begin{aligned}\hat{f}_{cstr_1} &= f_{cstr_1}/(f_{cstr_1} + 1) = 0 \\ \hat{f}_{cstr_2} &= f_{cstr_2}/(f_{cstr_2} + 1) = 0 \\ \hat{f}_{cstr_3} &= f_{cstr_3}/(f_{cstr_3} + 1) = 1/(1 + 1) = 0.5\end{aligned}$$

Finally, the overall fitness  $F$  of the solution  $V = "ab"$  is computed as  $F = avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}, \hat{f}_{cstr_3}) = 0.16$ .

3) *Selecting the best solution components.* The two steps above are repeated  $A$  times, with the values computed at each iteration stored as elements of the auxiliary variables  $tempT$ , an array containing sets of solution components, and  $tempF$ , an array containing the fitness values for the corresponding elements in  $tempT$ . Function BESTSOLUTION determines among them the solution components that have the minimum (i.e., best) fitness, and assign them to variable  $T_{Best}$ , representing the best solution of the current iteration of the outer loop.

**Application of local search.** This step (lines 18–20) is used to refine the set of candidate solutions built in the step above, to locally optimize them. More precisely, if the best solution of the current iteration ( $T_{Best}$ ) is better

10. To invoke unsupported operations, we assume that the corresponding bytecode is accessible through the classpath; we use the Java reflection methods to load and execute the code of the unsupported operation. Notice that this execution is subject to the timeout defined in function CSTRSOLVE.

11. This means that the insertion, modification, or deletion of one character is required in order to satisfy this constraint (see Section 7.2.1).



than (i.e., its fitness is lower than the fitness of) the best solution of the previous iteration, we perform a local search procedure to see whether further improvements can be made with other solutions that are in the neighborhood of  $T_{Best}$ . The local search is performed using the 2-opt local search algorithm [49], which finds in each automaton in  $K$  other paths (or sequence of transitions) that reach the final state. This algorithm replaces at most two transitions of the current path with one or more transitions; if it finds a set of solution components with a better fitness value, this set becomes the new  $T_{Best}$ .

To illustrate this step through the example in Figure 10, let us assume that the current best solution  $T_{Best}$  is the sequence of transitions  $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$ , which represents the solution  $V = "ab"$ . The application of 2-opt local search algorithm might result in replacing the second transition  $(s_1, s_2)$  with a different transition  $(s_1, s_6)$ . This produces a new sequence of transitions  $\{(s_0, s_1), (s_1, s_6)\}$ , which represents a new candidate solution  $V = "a"$ . The fitness of this new solution is lower than the one of the current best solution and, hence, the current  $T_{Best}$  is not changed. This example shows that a local search procedure may not always find a better solution; nevertheless it is useful when there is a better solution in the neighborhood of the current search space.

**Update of pheromone values.** This step (line 21) updates the pheromone values  $\xi_e \in \Xi$ , for each transition  $e$  of each automaton in  $K$ . It first computes  $\xi_{max} = \frac{1}{1-\rho} \frac{1}{F_{Best}}$  and  $\xi_{min} = \frac{\xi_{max}}{2n}$ , where  $n$  denotes the cumulative total number of states of all the automata in  $K$ ; then, it sets  $\xi_e = (1 - \rho)\xi_e + \Delta\xi_e$ , where  $\Delta\xi_e = \frac{1}{F_{Best}}$  if the transition  $e$  is part of the solution components in  $T_{Best}$ , 0 otherwise. If  $\xi_e > \xi_{max}$ , then it sets  $\xi_e = \xi_{max}$ ; dually, if  $\xi_e < \xi_{min}$ , then it sets  $\xi_e = \xi_{min}$ .

In the case of the example in Figure 10, recall that the current best solution  $T_{Best}$  is the sequence of transitions  $\{(s_0, s_1), (s_1, s_2), (s_2, s_6)\}$ , with  $F_{Best} = 0.16$ . The pheromone values of the transitions of the solution automaton are updated as follows:

$$\begin{aligned}\xi_{max} &= \frac{1}{1-\rho} \frac{1}{F_{Best}} = \frac{1}{1-0.01} \frac{1}{0.16} = 6.31 \\ \xi_{min} &= \frac{\xi_{max}}{2n} = 6.31 / (2 * 7) = 0.45 \\ \xi_{(s_0, s_1)} &= (1 - \rho)\xi_{(s_0, s_1)} + \Delta\xi_{(s_0, s_1)} \\ &= (1 - 0.01) * 2 + 1/0.16 = 8.23 \rightarrow 6.31 \\ \xi_{(s_1, s_2)} &= (1 - 0.01) * 1 + 1/0.16 = 7.24 \rightarrow 6.31 \\ \xi_{(s_2, s_6)} &= (1 - 0.01) * 1 + 1/0.16 = 7.24 \rightarrow 6.31 \\ \xi_{(s_0, s_4)} &= (1 - 0.01) * 1 + 0 = 0.99 \\ \xi_{(s_4, s_5)} &= (1 - 0.01) * 1 + 0 = 0.99 \\ \xi_{(s_5, s_6)} &= (1 - 0.01) * 1 + 0 = 0.99 \\ \xi_{(s_1, s_6)} &= (1 - 0.01) * 1 + 0 = 0.99 \\ \xi_{(s_2, s_3)} &= (1 - 0.01) * 1 + 0 = 0.99 \\ \xi_{(s_3, s_6)} &= (1 - 0.01) * 1 + 0 = 0.99\end{aligned}$$

Regarding the transitions  $(s_0, s_1)$ ,  $(s_1, s_2)$ , and  $(s_2, s_6)$ , their pheromones values are set to  $\xi_{max}$  since their originally computed values are larger than  $\xi_{max}$ . Note that, for the transitions

$$\begin{aligned}T_{Best} &= \{SID := aRXxQ1zCVmaetowbnZv0t\} \\ f_{cstr_1} &= 0 \\ f_{cstr_2} &= \psi(aRXxQ1zCVmaetowbnZv0t, .*['<=>/,;+-&*[]].*) = 1 \\ \hat{f}_{cstr_1} &= 0; \hat{f}_{cstr_2} = 0.5 \\ F_{Best} &= avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}) = 0.25 \\ &\text{(a) Iteration \#100} \\ T_{Best} &= \{SID := \$Qaa.&@erp!t'TmoopEn=\} \\ f_{cstr_1} &= 0 \\ f_{cstr_2} &= \psi(\$Qaa.&@erp!t'TmoopEn=, .*['<=>/,;+-&*[]].*) = 0 \\ \hat{f}_{cstr_1} &= 0; \hat{f}_{cstr_2} = 0 \\ F_{Best} &= avg(\hat{f}_{cstr_1}, \hat{f}_{cstr_2}) = 0 \\ &\text{(b) Iteration \#1000}\end{aligned}$$

Figure 11. Results after 100 and 1000 iterations of the SEARCHSOLVE procedure

$(s_0, s_4), (s_4, s_5), (s_5, s_6), (s_1, s_6), (s_2, s_3), (s_3, s_6)$ ,  $\Delta\xi = 0$  since these transitions are not part of  $T_{Best}$ .

The termination conditions of the loop at line 23 in Figure 9 correspond either to a time-out or to the finding of a solution that satisfies all the constraints in  $H$ , for which the fitness  $F_{Best}$  is zero. If there is a timeout, the function returns an empty set of solutions. Otherwise, it updates  $Sol$  with  $T_{Best}$  (line 27); i.e., the solution automata in  $Sol$  are replaced with the solutions represented by  $T_{Best}$  for the corresponding variables, and it returns  $Sol$ .

### 7.2.3 Application to the Running Example

We now show the application of SEARCHSOLVE to our running example. We recall that hypersubgraph  $H_3$  in Figure 7 was only partially solved through the application of function SOLVESUPPORTEDOPS in § 7.1.3 since it contains an unsupported operation customSanit.

Procedure SEARCHSOLVE will attempt to find a value from the solution automata  $M_{SID} := \{21\}$  (determined by the automata-based solver) for the string variable  $SID$  that satisfies the two constraints  $cstr_1 \equiv SID.length() > 20$ ;  $cstr_2 \equiv customSanit(SID).matches(".*['<=>/,;+-&*[]].*")$  in  $H_3$ . Figure 11 shows the best results ( $T_{Best}$ ,  $f$ ,  $\hat{f}$ , and  $F_{Best}$ ) obtained after 100 and 1000 iterations;  $f_{cstr_1}$  is computed by evaluating the constraint  $cstr_1$  with the value in  $T_{Best}$  and by using the Korel function;  $f_{cstr_2}$  is computed by evaluating the constraint  $cstr_2$  with the value in  $T_{Best}$  and by using the equality cost function for regular expression matching. At iteration 1000, the search converges towards the desired solution for  $SID$ , which satisfies the constraints in  $H_3$ .

## 8 IMPLEMENTATION

We have implemented our approach in a tool called JOACO (available online [15]). It consists of two major components, the *security slicer* and the *constraint solver*; both are implemented in Java, comprising approximately 34 kLOC excluding library code, spaces and comments.

The *security slicer* is derived from our previous work [6] and is built on top of Wala [50] and Joana [51], which

provide interprocedural program slicing and static analysis of paths in the slice, respectively. Given the bytecode of a Web program, the security slicer first extracts a security slice for each sink. It then explores the paths in the slice that lead to the sink in a depth-first manner, extracting the path conditions and the context information. The latter is used to generate the attack condition, by conjoining the path condition with the appropriate threat model. For scalability reasons, when encountering loops and recursive function calls, the slices iterates through them only once.

The *constraint solver* comprises three modules: constraint preprocessor, an automata-based and interval constraint solver and a search-based constraint solver. The constraint preprocessor makes use of the JGraphT library [52], a Java class library that provides mathematical graph-theory objects and algorithms, in order to generate a constraint network from the attack condition, as explained in section 6. The constraint network is then passed to the constraint solver to prove the presence/absence of a vulnerability.

Our automata-based and interval constraint solver handles string and integer constraints with supported operations, as described in § 7.1. It is built on top of JSA [36] and Sushi [3]. JSA models a set of Java string/mixed operations using finite state automata; Sushi adds supports for string replacement and regular expression replacement operations using finite state automaton and transducer operations. In this component, we also defined the recipes for additional string operations (see Table 5), such as the security APIs provided by two popular security libraries (OWASP [10] and Apache [9]). The search-driven constraint solver is invoked when a constraint contains unsupported operations, as described in § 7.2.

## 9 EVALUATION

In this section we report on the evaluation of *JOACO*, in terms of 1) vulnerability detection capability when analyzing the source code of a Web application; 2) capability of solving string constraints derived from potential vulnerabilities in realistic systems. The first task corresponds to the normal usage scenario of *JOACO* for detecting vulnerabilities in Web applications. We also included the second usage scenario because many research efforts (see related work in section 11) focus (only) on string constraint solving, as a means to enable vulnerability detection; indeed, in such a context, *JOACO* can be also used as a stand-alone string constraint solver (we call it *JOACO-CS* when used in this mode).

We assess the effectiveness of *JOACO* in performing these two tasks by answering the following research questions:

RQ1: *What is the effectiveness of JOACO in detecting XSS, SQLi, XMLi, XPathi, and LDAPi vulnerabilities and how does it compare with state-of-the-art vulnerability detection tools, including our previous work [6], [7]? (§ 9.2.1)*

RQ2: *What is the effectiveness of JOACO-CS in solving string constraints characterizing potential vulnerabilities in representative and widely used systems and benchmarks and how does it compare with state-of-the-art, general-purpose string constraint solvers, including our previous work [7]? (§ 9.2.2)*

RQ3: *How does the constraint preprocessing described in section 6 affect the execution time of JOACO? (§ 9.2.3)*

### 9.1 Benchmarks and Evaluation Settings

We use six different benchmarks, obtained from different sources, to evaluate *JOACO*: *JOACO-Suite*, *StrangerJ-Suite*, *Pisa-Suite*, *AppScan-Suite*, *Kausler-Suite*, and *Cashew-Suite*.

*JOACO-Suite* is our homegrown benchmark, composed of 11 open-source Java Web applications/services and security benchmark applications that have been used in the literature, with known XSS, XMLi, XPathi, LDAPi, and SQLi vulnerabilities. It is an extended version of the benchmark used in our previous work [6], [7], enriched with two new applications: *Bodgeit* and *OMRS-LUI*.

*WebGoat* [53] and *Bodgeit* [54] are deliberately insecure Web applications developed for the purpose of teaching security vulnerabilities in Web applications. *Roller* [55] and *Pebble* [56] are blogging applications that also expose Web service APIs. *WebGoat*, *Roller* and *Pebble* have been already used as benchmarks in the vulnerability detection literature [20], [21], [57], [58], [59], [60]. *openmrs-module-legacyui* (shortened as *OMRS-LUI*) [61] is the user interface package of *OpenMRS* [62], a widely used, open-source medical record system that manages highly sensitive medical data. *Regain* [63] is a search engine, known to be used in a production-grade system by one of the biggest drugstore chains in Europe. The *pubsubhubbub-java* (shortened as *PSH*) tool [64] is the most popular Java project related to the *PubSubHubbub* protocol in the Google Code archive. The *rest-auth-proxy* (shortened as *RAP*) microservice [65] is one of the most popular LDAP-based Web service Java projects returned by a query on Github.com with the search string `ldap rest`. *TPC-APP*, *TPC-C*, and *TPC-W* are the standard benchmarks provided by [66] for evaluating vulnerability detection tools for Web services; the set of Web services they provide has been accepted as representative of real environments by the Transactions processing Performance Council (<http://www.tpc.org>). As shown in the top part of Table 7, this benchmark contains in total 129 paths to sinks (and as many constraints): 86 paths vulnerable to XMLi, XPathi, XSS, LDAPi, or SQLi, and 43 non-vulnerable ones. Note that a vulnerable path corresponds to a single vulnerability.

*StrangerJ-Suite* is a security benchmark distilled from five real-world PHP web applications (*MyEasyMarket*, *proManager*, *PBLguestbook*, *aphpkb*, and *BloggIT*). It has been used for assessing the effectiveness of the *Stranger* tool [67] in the context of automatically detecting and sanitizing security vulnerabilities in PHP Web applications. We have manually translated every program of this benchmark from PHP to Java so that we could use it in our evaluation. As shown in the bottom part of Table 7, this benchmark contains in total 9 paths which are all vulnerable to XSS.

*Pisa-Suite* contains 12 constraints generated from sanitizers detected by PISA [68]; these constraints have been used in the experimental evaluation reported in [23].

*AppScan-Suite* contains 8 constraints derived from the security warnings emitted by IBM Security AppScan [69], a commercial vulnerability scanner tool, when executing on a set of popular websites. The generated warnings contain

traces of program statements that reflect potentially vulnerable information flows. Also these constraints have been used in the experimental evaluation reported in [23].

*Kausler-Suite* contains 120 constraints obtained from eight Java programs via dynamic symbolic execution. A superset of this benchmark (with 175 constraints) has been used for evaluating four string constraint solvers in the context of symbolic execution [70]; the subset used in this paper contains the constraints that were successfully translated into the SMT-LIB format by the tool provided in the replication package of [70].

*Cashew-Suite* contains 394 distinct constraints obtained through the normalization of the constraints of the SMC/*Kaluza* benchmark by means of the *Cashew* tool [71]. The SMC/*Kaluza* benchmark [72] contains the 18896 satisfiable<sup>12</sup> constraints of the *Kaluza* benchmark, converted to the input format of the SMC solver [72]; the *Kaluza* benchmark contains constraints corresponding to path conditions generated from a set of JavaScript programs by a symbolic execution engine [2]. Although the *Kaluza* benchmark has been widely used for evaluating string constraint solvers in the past, its high degree of redundancy (high number of constraints equivalent in terms of satisfiability, as highlighted by the recent work on constraint normalization [71]) led us to rely on *Cashew-Suite* instead. This was meant to prevent biasing the overall results with an extremely large and redundant benchmark, without any loss of information.

Since *JOACO-Suite* and *StrangerJ-Suite* are the only benchmarks containing the source code of Java Web applications, they were the only ones used for answering RQ1. All six benchmarks were however used for answering RQ2 and RQ3. Notice that *JOACO-Suite*, *StrangerJ-Suite*, *Pisa-Suite*, and *AppScan-Suite* contain constraints derived from potentially vulnerable Web applications, whereas *Kausler-Suite* and *Cashew-Suite* have been used to evaluate string constraint solvers from a general standpoint (not necessarily related to security analysis).

We established the ground truth (i.e., whether a path is vulnerable or not) of the constraints in *JOACO-Suite* in the following way. *WebGoat* and *Bodgeit* are deliberately in-secured applications for teaching security vulnerabilities and, hence, they already provided the ground truth. However, as explained in § 9.2.1, *JOACO* was able to detect four previously unknown vulnerabilities in *Bodgeit*. Since *TPC-APP*, *TPC-C* and *TPC-W* are standard benchmarks for testing vulnerability detection tools for Web services, their ground truth was available. Although *Pebble* and *Roller* have been already used as benchmarks for vulnerability detection, no ground truth was available; therefore, we consulted the US National Vulnerability Database (NVD) [73] and confirmed the reported vulnerabilities, by exploiting them in the deployed applications and by locating their corresponding paths in the source code. *RAP*, *PSH*, *Regain* and *OMRS-LUI* did not have any recent entries in NVD; therefore, we established the ground truth by manually inspecting their source code and verified potential vulnerabilities by exploiting them in the deployed applications.

12. Notice that the SMC paper [72] reports a total of 18901 satisfiable constraints, but the evaluation artifacts include only 18896 constraints.

Table 7  
Vulnerable and non-vulnerable paths in the applications contained in the *JOACO-Suite* and *StrangerJ-Suite* benchmarks. A vulnerable path corresponds to a single vulnerability.

Application	LOC	# Paths	Vulnerable					Non-Vulnerable				
			XML	XPATH	XSS	LDAP	SQL	XML	XPATH	XSS	LDAP	SQL
JOACO-Suite												
WebGoat	24,608	15	1	2	0	0	8	0	1	0	0	3
Roller	52,433	13	0	0	3	0	0	7	0	3	0	0
Pebble	36,592	13	2	0	4	0	0	1	0	6	0	0
Regain	23,182	6	0	0	3	0	0	0	0	3	0	0
PSH	1,964	4	1	0	0	0	0	1	0	2	0	0
TPC-APP	2,082	12	0	0	0	0	6	0	0	0	0	6
TPC-C	9,184	34	0	0	0	0	30	0	0	0	0	4
TPC-W	2,470	6	0	0	0	0	3	0	0	0	0	3
RAP	442	1	0	0	0	1	0	0	0	0	0	0
Bodgeit	3,376	21	0	0	9	0	9	0	0	2	0	1
OMRS-LUI	34,074	4	0	0	4	0	0	0	0	0	0	0
			4	2	23	1	56	9	1	16	0	17
Subtotal	190,407	129	86					43				
StrangerJ-Suite												
MyEasyMarket	14	1	0	0	1	0	0	0	0	0	0	0
proManager	68	3	0	0	3	0	0	0	0	0	0	0
PBLguestbook	64	3	0	0	3	0	0	0	0	0	0	0
aphpkb	19	1	0	0	1	0	0	0	0	0	0	0
BloggIT	55	1	0	0	1	0	0	0	0	0	0	0
			0	0	9	0	0	0	0	0	0	0
Subtotal	220	9	9					0				
Total	190,627	138	95					43				

The ground truth for *StrangerJ-Suite* was available on the *Stranger* tool website [74].

The ground truth of *Pisa-Suite* and *AppScan-Suite* was established in [23]. As for *Kausler-Suite*, its constraints are all satisfiable since they were generated by dynamic symbolic execution. The ground truth of *Cashew-Suite* was established by running *Cashew* [71], which counts the number of models for every constraint: a model count greater zero indicates satisfiability.

We ran the experiments on a machine equipped with an Intel i7 2.4 GHz processor, 8 GB memory, running Apple Mac OS X 10.13.

The applications and the constraints included in the benchmark used for the evaluation are available on the tool web site [15].

## 9.2 Experimental Results

### 9.2.1 Effectiveness of Vulnerability Detection

To answer RQ1, we executed *JOACO* on the *JOACO-Suite* and *StrangerJ-Suite* benchmarks and compared it with two state-of-the-art vulnerability detection tools for Java Web applications: *LAPSE+* [12] and *SFlow* [11]<sup>13</sup>. Similarly to *JOACO*, *LAPSE+* and *SFlow* provide an end-to-end solution to detect vulnerabilities, since they take as input the source code of an application and produce a vulnerability report. Both tools are based on taint analysis and thus require to specify sources, sinks, and sanitization procedures. More

13. *SFlow* has been shown [11] to perform better than *Andromeda* [20], a commercial product from IBM.



SQL injection because the threat model 11 of Table 1 does not consider as vulnerable a user input sanitized through the `parseInt()` method.

In terms of precision, *LAPSE+* reported 5 false positives and failed to analyze 40 cases, resulting in a precision of 84%; *SFlow* reported 14 false positives and failed in 34 cases, resulting in a precision of 63%. In both cases, the false positives were mainly due to the lack of constraint solving capabilities in these tools. *JoanAudit+CVC4+ACO-Solver* achieved a precision of 100%, with no false positives and failing cases; however, it timed-out in 26 cases, out of which 21 were non-vulnerable and 5 were vulnerable.

*JOACO* achieved 100% precision, with no failing cases and only 7 time-out cases, which are all UNSAT cases and hence could not be solved by the search-based solver. Compared to *JoanAudit+CVC4+ACO-Solver*, *JOACO* could handle 18 more cases without running into time-outs.

We remark that *JOACO-Suite* is an extended version of the benchmark used in our previous work [6], [7]. For the two new applications added to the benchmark (*Bodgeit* and *OMRS-LUI*), *JOACO* was able to detect 4 previously unknown vulnerabilities (1 XSS and 3 SQLi vulnerabilities) for *Bodgeit* and 4 previously unknown XSS vulnerabilities for *OMRS-LUI*. *LAPSE+* detected only 3 of the new vulnerabilities for *Bodgeit* and could not detect any of the new vulnerabilities for *OMRS-LUI*; as mentioned above, *SFlow* could not analyze any program included in *Bodgeit*, *OMRS-LUI*, and *Stranger*. The new vulnerabilities found in *Bodgeit* have been reported on the project web site<sup>15</sup>; the new vulnerabilities for *OMRS-LUI* have been reported and confirmed by the *OpenMRS* developers.

We also compared the four tools in terms of execution time; the detailed results are shown in columns  $t(s)$  of Table 8. *LAPSE+* took 68s; *SFlow* took 84s; *JoanAudit+CVC4+ACO-Solver* took 3773.4s; *JOACO*, with constraint preprocessing enabled (+*Opt*), took 2981.4s. The execution time of *JOACO* is much larger than that of *LAPSE+* and *SFlow*, since it includes several steps such as security slicing and constraint solving; nevertheless, such a large time is not practically relevant for the purpose of vulnerability detection since such analysis is performed when new code is committed and is not required to provide immediate feedback. Furthermore, *JOACO* is about 21% faster than *JoanAudit+CVC4+ACO-Solver*.

The answer to RQ1 is that the proposed approach implemented in *JOACO* is highly effective (achieving 98% recall, 100% precision) in detecting injection vulnerabilities; it performs much better than state-of-the-art vulnerability detection tools, yielding a higher recall (between +70pp and +73pp, with pp=percentage points) and precision (between +16pp and +27pp), with no failing cases. This high effectiveness in vulnerability detection comes at the cost of a higher execution time, which is however practically acceptable. Compared with our previous work *JoanAudit+CVC4+ACO-Solver*, *JOACO* detected more vulnerabilities, had much less time-out cases, and was faster.

## 9.2.2 Effectiveness of String Constraint Solving

To answer RQ2, we compared the constraint solving capabilities of *JOACO* when used in the stand-alone solver mode (dubbed *JOACO-CS*) with three state-of-the-art constraint solvers: *CVC4* (version 1.4) [13], the latest stable release of *Z3* (version 4.6.0) [14], and our previous work *CVC4+ACO-Solver* [7].

For the comparison, we used the constraints contained in all six benchmark suites, for a total of 672 constraints. We set the time-out for solving each constraint to 600s.

Table 9 shows the evaluation results. For each solver, we indicate the number of correct (column  $\checkmark$ ) and incorrect (column  $\times$ ) answers returned by the tool, grouped by the cases “SAT” and “UNSAT”, as well as the total; column  $?$  indicates the number of cases for which the solver returned “UNKNOWN”; column  $\dagger$  indicates the number of cases for which the solver execution failed, due to a run-time error or crash; column  $\odot$  indicates the number of cases in which the solver timed out; column  $S$  indicates the number of constraints for which the search-based solver had to be invoked.

We answer RQ2 by examining the number of correct and incorrect results, and the number of unknown/failing/time-out cases in Table 9.

For *JOACO-Suite*, *JOACO-CS* was the most effective solver, with 122 correct results (out of 129) and no unknown/failing cases. The 7 time-out cases are the ones discussed above for the same benchmark in the answer to RQ1: they are UNSAT cases and therefore, the search-based solver could not find satisfying solutions for them.

*Z3* yielded 34 correct with 52 unknown cases, 35 failing cases, and 8 time-outs; *CVC4* yielded 92 correct results and 2 incorrect ones, with 35 failing cases; *CVC4+ACO-Solver* yielded 103 correct results and 2 incorrect ones, with 24 time-out cases. The failing cases of *CVC4* and *Z3* were due to unsupported operations.

For *StrangerJ-Suite*, *JOACO-CS* was the only solver that could solve all 9 cases correctly; in particular, five of them contained unsupported operations that could be solved only thanks to the search-based solver of *JOACO-CS*. These unsupported operations are incomplete sanitization operations that only partially filter attack patterns from user inputs.

Both *Z3* and *CVC4* had 5 failing cases because of unsupported operations and 1 timeout. *CVC4+ACO-Solver* yielded 7 correct results and two timeouts.

For *Pisa-Suite*, *Z3* and *JOACO-CS* were the most effective solvers, solving all constraints correctly; by contrast, both *CVC4* and *CVC4+ACO-Solver* had six timeouts.

*AppScan-Suite* was correctly solved by both *JOACO-CS* and *Z3*, whereas both *CVC4* and *CVC4+ACO-Solver* had 5 time-outs.

For *Kausler-Suite*, *JOACO-CS* was the most effective solver as well, solving all the 120 constraints correctly. On the other hand, *CVC4* and *CVC4+ACO-Solver* yielded 117 correct results and 3 time-out cases; *Z3* yielded 118 correct results, one unknown and one failing case.

For *Cashew-Suite*, *CVC4+ACO-Solver* and *JOACO-CS* solved all the constraints correctly. *CVC4* reported one unknown case; *Z3* had one failing case.

15. Issues #17–#20 on <https://github.com/psiinon/bodgeit/>.

Table 9

Comparison of the effectiveness in constraint solving among *JOACO-CS* (with constraint preprocessing switched on and off), *Z3*, *CVC4* and *CVC4+ACO-Solver* (✓: correct answers, ✗: incorrect answers, ?: unknown cases, ⚡: failing cases, ⌚: time-outs, S: search-based solver invocation)

Suite	Z3									CVC4									CVC4+ACO-Solver									JOACO-CS												
	SAT			UNSAT			TOTAL			?	f	⊘	SAT			UNSAT			TOTAL			?	f	⊘	S	SAT			UNSAT			TOTAL			?	f	⊘	S		
	X	✓		X	✓		X	✓					X	✓		X	✓		X	✓						X	✓		X	✓		X	✓						X	✓
JOACO	0	14		0	20		0	34	52	35	8	1	72	1	20	2	92	0	35	0	1	81	1	22	2	103	0	0	24	33	0	85	0	37	0	122	0	0	7	7
Stranger]	0	3		0	0		0	3	0	5	1	0	3	0	0	0	3	0	5	1	0	7	0	0	0	7	0	0	2	5	0	9	0	0	0	9	0	0	0	5
Pisa	0	8		0	4		0	12	0	0	0	0	6	0	0	0	6	0	0	6	0	6	0	0	0	6	0	0	6	0	0	8	0	4	0	12	0	0	0	0
AppScan	0	8		0	0		0	8	0	0	0	0	3	0	0	0	3	0	0	5	0	3	0	0	0	3	0	0	5	0	0	8	0	0	0	8	0	0	0	0
Kausler	0	118		0	0		0	118	1	1	0	0	117	0	0	0	117	0	0	3	0	117	0	0	0	117	0	0	3	0	0	120	0	0	0	120	0	0	0	0
Cashew	0	381		0	12		0	393	0	1	0	0	381	0	12	0	393	1	0	0	0	382	0	12	0	394	0	0	0	0	0	382	0	12	0	394	0	0	0	0
Total	0	532		0	36		0	568	53	42	9	1	582	1	32	2	614	1	40	15	1	596	1	34	2	630	0	0	40	38	0	612	0	53	0	665	0	0	7	12

To sum up, even if we disregard the *JOACO-Suite* and *Stranger]*-Suite benchmarks (which are the only ones containing unsupported operations) and consider only the remaining four benchmarks (which contain only supported operations), *CVC4* correctly solved 519 constraints and had 14 time-outs (and also one unknown case); *Z3* correctly solved 531 constraints and had two failing cases (and also one unknown case). Our previous work *CVC4+ACO-Solver* correctly solved 520 constraints and had 14 time-outs. By contrast, *JOACO-CS* correctly solved all the 534 constraints with no time-outs, no unknown cases, and no failing cases.

We also compared the constraint solving time of the four tools; the results are shown in Table 10, together with the number of constraints in each benchmark. In total, *Z3* took 6840.6 s ( $\approx 2$ h) and correctly solved 568 cases; *CVC4* took 9735.9 s ( $\approx 3$ h) and correctly solved 614 cases; *CVC4+ACO-Solver* took 32061.7 s ( $\approx 9$ h) and correctly solved 630 cases; *JOACO-CS* took 16879.4 s ( $\approx 5$ h) with constraint preprocessing enabled (+*Opt*) and correctly solved 665 cases. The average execution time for solving one constraint (computed as  $\frac{\text{Total time}}{\text{\# constraints}}$ ) is 10.2 s for *Z3*, 14.5 s for *CVC4*, 47.7 s for *CVC4+ACO-Solver*, 25.1 s for *JOACO-CS* (+*Opt*). On average, our approach is  $1.7\times$  slower than the most effective, state-of-the-art solver (*CVC4*), and about  $2.4\times$  slower than *Z3*. Note that *Z3* is faster but less effective than *CVC4* since it solved only 568 cases correctly, whereas *CVC4* was able to solve 614 cases. *JOACO-CS* is about  $1.9\times$  faster than our previous work *CVC4+ACO-Solver*; this is due to the larger number of string operations supported by *JOACO-CS*, which reduces the number of constraints (from 33 to 7) for which it is necessary to invoke the search-based solver. Nevertheless, *JOACO-CS* could solve the highest number of constraints in our benchmarks.

The answer to RQ2 is that the proposed constraint solving approach implemented by *JOACO-CS* is highly effective in string constraint solving and performs similarly to or better (7%–14% more correctly solved cases) than state-of-the-art string constraint solvers, including our previous work, depending on the benchmark considered. In terms of execution time, *JOACO-CS* is  $1.7\times$  slower than the most effective, state-of-the-art constraint solver (*CVC4*). However, since *JOACO-CS* can solve more cases and constraint solving is typically an offline activity, with no stringent time requirements, we consider this slowdown as practically acceptable.

Table 10

Execution time (in seconds) for *Z3*, *CVC4*, *CVC4+ACO-Solver* and *JOACO-CS* with constraint preprocessing switched off (−*Opt*) and constraint preprocessing switched on (+*Opt*)

Suite	#Constraints	Z3	CVC4	CVC4+ACO-Solver	JOACO-CS (− <i>Opt</i> )	JOACO-CS (+ <i>Opt</i> )
<i>JOACO</i>	129	5156.8	69.1	14505.4	5999.4	5707.9
<i>Stranger]</i>	9	671.7	610.8	1264.3	174.4	162.9
<i>Pisa</i>	12	4.5	3964.8	4006.8	212.9	178.3
<i>AppScan</i>	8	10.5	3002.2	3026.4	168.2	144.0
<i>Kausler</i>	120	945.8	2043.5	8804.3	6079.7	5031.3
<i>Cashew</i>	394	51.4	45.6	454.5	6827.9	5654.9
Total	672	6840.6	9735.9	32061.7	19462.6	16879.4
Avg. Time		10.2	14.5	47.7	29.0	25.1

### 9.2.3 The Role of Constraint Preprocessing

To answer RQ3, we re-ran all the experiments conducted for answering RQ1 and RQ2 by using *JOACO* and *JOACO-CS* with preprocessing disabled (denoted by −*Opt*); we then compared the resulting values for the execution time with the ones obtained with the preprocessing enabled (denoted by +*Opt*).

For the use case of vulnerability detection, columns −*Opt* and +*Opt* in Table 8 show that enabling the constraint preprocessing led to reduction of about 200 s in execution time (from 3194.9 s down to 2981.4 s) corresponding to a relative reduction of about 7%. For the use case of string constraint solving, columns *JOACO-CS* (−*Opt*) and *JOACO-CS* (+*Opt*) in Table 10 show that *JOACO-CS* with constraint preprocessing disabled took 19 460.2 s, whereas it took only 16 879.4 s with constraint preprocessing enabled, corresponding to an execution time reduction of about 15%.

Since constraint preprocessing only impacts the efficiency of constraint solving, it has a higher impact on the execution time of *JOACO-CS* for string constraint solving (see Table 10) than for the case of vulnerability detection with *JOACO*, which also includes the security slicing step (see Table 8).

The answer to RQ3 is that constraint preprocessing has a positive impact on the execution time of our approach, with reductions ranging between 7% and 15% depending on the use case.

## 10 LIMITATIONS

The results presented above have shown the effectiveness of our approach in detecting injection vulnerabilities and solving string constraints. However, it suffers from the following main limitations.

Since our approach is mainly based on static analysis (for determining attack conditions), it inherits one of the intrinsic weaknesses of the latter: it cannot deal with calls to system and library functions for which the source code is not available. To solve constraints containing calls to these functions, we consider the latter as unsupported operations and rely on our search-driven technique, effectively invoking the functions in isolation. This strategy may generate false positives when the functions have side-effects (e.g., they modify the global heap memory) that affect some constraints, because the path conditions previously collected may not reflect the actual execution. Furthermore, the invocation of some library functions may not always be possible, e.g., for functions that require complex data structures as arguments. In such a situation, our approach heuristically determines that the constraints involving those functions are satisfiable, which may also produce false positives.

Another limitation is that the threat models we defined may not be complete. Our approach can miss vulnerabilities (i.e., yielding false negatives) due to unknown or new type of attack patterns. To mitigate this, *JOACO* supports the inclusion of additional threat models that reflect new attack patterns through a configuration file.

Search-based solving has limitations in the presence of non-deterministic operations: for solving random operations (e.g., random hash functions), search-based solving is as (in-)effective as random search. However, since these cases are relatively rare, we consider the practical consequences to be negligible.

Our approach may also miss vulnerabilities when the solver times out, even though the attack condition is satisfiable. We expect such time-out cases to be rare since our hybrid constraint solving technique has experimentally shown to be very effective; indeed, in our experiments all the time-out cases of *JOACO* were due only to unsatisfiable constraints.

Finally, *JOACO* is designed to work only on the (Java) source-code of a Web application/service; hence it cannot be used to analyze Web applications exposing business processes implemented using workflow languages such as BPMN [75] or WS-BPEL [76]. Furthermore, the current version of *JOACO* does not support the analysis of applications written using Java Web frameworks (e.g., Spring [77]); adding this support is planned for the next major version (see also the discussion of future work in section 12).

## 11 RELATED WORK

Our approach is related to work done in the areas of code-based security analysis, automated prevention of injection vulnerabilities, penetration testing, (string) constraint solving, constraint solving through heuristic search, and search-based test input generation for string data types.

*Code-based security analysis.* This category includes two types of approaches: taint analysis and symbolic execution. Approaches based on taint analysis (such as [11], [20],

[21], [57], [78], [79]) check whether application inputs are used in sinks without passing through known sanitization functions. However, these approaches tend to generate many false alarms [66], [80] since they cannot reason about the implementation of sanitization functions. Some approaches [81], [82] incorporate string analysis into taint analysis, improving the precision in the analysis of SQLi and XSS vulnerabilities. Other approaches [80], [83], [84] reason about the adequacy of input sanitization code by combining taint analysis and string constraint solving using finite state automata operations. However, these approaches usually support only a limited set of string operations, are targeted towards XSS and SQLi, and lack support for complex constraints that involve string/mixed and numeric operations.

Approaches based on symbolic execution [1], [2], [4] perform (dynamic) symbolic execution on programs and generate path conditions. They then use a constraint solver to check these conditions and determine whether inputs used in sinks may contain security attack values. These approaches, which rely on (string) constraint solving, exhibit the same limitations (e.g., limited support for complex string operations) as constraint solvers, which are discussed further below. In addition, these approaches switch to *dynamic* symbolic execution for scalability when encountering the path explosion problem; however, such a strategy may lead to omitting the analysis of certain program parts and thus, missing vulnerabilities. By contrast, our approach applies security slicing to extract only program parts relevant to security; this greatly improves scalability without sacrificing vulnerability detection effectiveness.

*Automated prevention of injection vulnerabilities.* There are approaches that automatically prevent code injection vulnerabilities by sanitizing potentially malicious user inputs [85], [86] or by inserting run-time mechanisms that check against security policies [87], [88]. CSAS [86] automatically inserts sanitization routines into the code generated by Web templating frameworks. ScriptGuard [85] learns which sanitizers to use for certain program paths during a training phase, infers incorrect sanitizations, and fixes them by applying the correct sequences of sanitizers. Synode [88] statically computes templates of the values passed to Node.js APIs and synthesizes a security policy from these templates, which is then used to detect potential attacks at run time. XSS-Guard [87] retrofits a Web application with the capability of learning (through the analysis of HTTP responses) the scripts it intends to create; at run time, the instrumented application is then able to remove any unintended, potentially malicious script.

*JOACO* also applies a lightweight automatic sanitization during the security slicing step. However, by contrast, our approach does not require the use of dynamic analysis or run time checks; it is only based on static analysis. Furthermore, the aforementioned approaches focus only on XSS vulnerabilities whereas *JOACO* can deal with several injection vulnerability types (XSS, SQLi, XMLi, XPathi, and LDAPi).

*Penetration testing.* Penetration testing tools like Acunetix [89], BurpSuite [90], and AppScan [69] are useful for detecting the presence of vulnerabilities in Web programs. Antunes and Vieira [66] evaluated these tools and



observed that penetration testing approaches miss vulnerabilities and are in general less accurate than taint analysis approaches. More specifically, penetration testing cannot detect vulnerabilities that require to craft the corresponding attack values in order to exploit the weaknesses of input sanitization functions. On the other hand, our approach, being based on constraint solving, returns a concrete attack value for the input (i.e., a solution) when it identifies a vulnerability.

*(String) constraint solving.* There are many constraint solvers that provide, to a certain degree, support for strings: bit-vector-based solvers like Hampi [22], Kaluza [2], and Utopia [91]; automata-based solvers like Violist [92], *Stranger* [67], [93], ABC [94], StrSolve [95], Pass [96], StringGraph [29], and JST [30]; word-based solvers like Norn [97], S3 [98], and the aforementioned Sushi, CVC4, Z3-str2 and Z3. Among them, *Stranger*, JST, StringGraph, S3, CVC4, Z3-str2 and Z3 support the highest number of string operations (e.g., `startsWith`, `endsWith`, `replace`, `replaceAll`, `length`, and `matches`) that are essential in the context of vulnerability detection; they also support numeric constraints. Although Hampi and Kaluza have been widely used as benchmarks for evaluating other solvers (see [13], [23], [97], [98]), they actually support only a smaller set of string operations than the solvers listed above; also, Hampi does not support numeric constraints. Support for regular expressions (which are usually used in attack specifications) is only provided — often in a limited form — by Sushi, *Stranger*, ABC, Kaluza, S3, Z3-str2, Z3 and CVC4. Nevertheless, none of them provides full support for a complete string function library of a modern programming language or for sanitization libraries like OWASP ESAPI and Apache Commons Lang. This means that they fail when they encounter an unsupported operation in an input constraint; in turn, this may lead to missing vulnerabilities. By contrast, in our approach we use a search-based meta-heuristic algorithm to handle unsupported operations.

*Constraint solving through heuristic search.* Heuristic search has been already proposed [99] for solving non-linear arithmetic constraints with operations from unsupported numeric libraries; the heuristics is optimized to explore an  $n$ -dimensional space over real numbers. In our approach we focus on solving constraints with string/mixed and integer operations; the search heuristics is optimized, in terms of search strategy and fitness functions, for these kind of constraints. Further, the approach in [99] is evaluated in terms of coverage of test generators, while we evaluated our approach in the context of vulnerability detection.

*Search-based test input generation for string data types.* There are a few proposals [46], [48], [100] that apply a search-based approach (typically genetic algorithms) for generating test cases in the form of string input values, in the context of satisfaction of branch coverage criteria. Their goal is to improve coverage by driving the search for string values, either with useful seed values [46], [48] or by hybridizing global search and local search [100]. In our case, attack conditions (which include full path conditions and attack specifications) are much more complex than branch conditions and thus we need to reduce the search space. Since we rely on automata-based solvers for search space reduction, our search algorithm works on automata and, as a result, we had to devise

a search strategy that is effective on graph representations. This was the main reason to select Ant Colony Optimization, which resulted in a significantly different search strategy than the ones proposed in the above-mentioned approaches.

## 12 CONCLUSION

This work addresses the challenge of analyzing the source code of a Java Web application for detecting injection vulnerabilities in a scalable and effective way. We have proposed an integrated approach that seamlessly combines static analysis-based security slicing with hybrid constraint solving, that is constraint solving based on a combination of automata-based solving and meta-heuristic search (Ant Colony Optimization). We use static analysis to extract minimal program slices from Web programs relevant to security and to generate the attack conditions, i.e., conditions necessary for the slices to be vulnerable. We then apply a hybrid constraint solving procedure to determine the satisfiability of attack conditions and thus detect vulnerabilities.

The experimental results, using a benchmark comprising a set of diverse and representative Web applications/services as well as security benchmark applications, show that our approach (implemented in the JOACO tool) is significantly more effective at detecting injection vulnerabilities than state-of-the-art approaches, achieving 98% recall, without producing any false alarm. We also compared the constraint solving module of our approach with state-of-the-art constraint solvers, using six different benchmarks; our approach correctly solved the highest number of constraints (665 out of 672), without producing any incorrect result, and was the one with the least number of time-out/failing cases. In both scenarios, the execution time was practically acceptable, given the offline nature of vulnerability detection.

As part of future work, we plan to extend our integrated vulnerability detection approach with support for widely used Java Web frameworks such as Spring [77]. We also plan to incorporate dynamic symbolic execution to further enhance our approach.

## ACKNOWLEDGMENTS

We would like to thank Dr. Xiang Fu from Hofstra University, Hempstead for sharing the Sushi tool. This work is supported by the National Research Fund, Luxembourg FN-R/P10/03, INTER/DFG/14/11092585, and the AFR grant FNR9132112.

## REFERENCES

- [1] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” in *Proceedings of ICSE’09*. IEEE, 2009, pp. 199–209.
- [2] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for JavaScript,” in *Proceedings of S&P’10*. IEEE, 2010, pp. 513–528.
- [3] X. Fu, M. Powell, M. Bantegui, and C.-C. Li, “Simple linear string constraints,” *Form. Asp. Comput.*, vol. 25, no. 6, pp. 847–891, 2013.
- [4] Y. Zheng and X. Zhang, “Path sensitive static analysis of Web applications for remote code execution vulnerability detection,” in *In Proceedings of ICSE’13*. IEEE, 2013, pp. 652–661.
- [5] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

- [6] J. Thomé, L. K. Shar, D. Bianculli, and L. Briand, "Security slicing for auditing common injection vulnerabilities," *J. Syst. Softw.*, 2017, (in press) <https://doi.org/10.1016/j.jss.2017.02.040>.
- [7] J. Thomé, L. Shar, D. Bianculli, and L. Briand, "Search-driven string constraint solving for vulnerability detection," in *Proceedings of ICSE'17*. IEEE, 2017, pp. 198–208.
- [8] M. Dorigo and K. Socha, "An introduction to ant colony optimization," IRIDIA, Tech. Rep. TR/IRIDIA/2006-010, 2006.
- [9] Apache, "StringEscapeUtils," <https://commons.apache.org/proper/commons-lang/javadocs/api-3.1/org/apache/commons/lang3/StringEscapeUtils.html>, 2017.
- [10] OWASP, "OWASP ESAPI," [https://www.owasp.org/index.php/Category:OWASP\\_Enterprise\\_Security\\_API](https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API), 2017.
- [11] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for Java Web applications," in *Proceedings of FASE'14*. Springer, 2014, pp. 140–154.
- [12] P. M. Pérez, J. Filipiak, and J. M. Sierra, "LAPSE+ static analysis security software: Vulnerabilities detection in Java EE applications," in *Proceedings of FutureTech*. Springer, 2011, pp. 148–156.
- [13] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A DPLL(T) theory solver for a theory of strings and regular expressions," in *Proceedings of CAV'14*. Springer, 2014, pp. 646–662.
- [14] M. Berzish, Y. Zheng, and V. Ganesh, "Z3str3: A string solver with theory-aware branching," *CoRR*, vol. abs/1704.07935, 2017.
- [15] J. Thomé, "JOACO: Vulnerability analysis through security slicing and hybrid constraint solving," <https://sites.google.com/site/joacosite/home>, 2017.
- [16] R. Dechter, *Constraint Processing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.
- [17] É. Grégoire, J. Lagniez, and B. Mazure, "Preserving partial solutions while relaxing constraint networks," in *Proceedings of IJCAI'13*, Beijing, China, 2013, pp. 552–558.
- [18] OWASP, "OWASP Top 10," [https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project), 2017.
- [19] CWE, "Common weakness enumeration," <http://cwe.mitre.org/>, 2017.
- [20] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "ANDROMEDA: Accurate and scalable security analysis of Web applications," in *Proceedings of FASE'13*. Springer, 2013, pp. 210–225.
- [21] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of Web applications," in *Proceedings of PLDI'09*. ACM, 2009, pp. 87–97.
- [22] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A solver for string constraints," in *Proceedings of ISSTA'09*. ACM, 2009, pp. 105–116.
- [23] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," in *Proceedings of CAV'15*. Springer, 2015, pp. 235–254.
- [24] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *Proceedings of TAICPART-MUTATION'07*. IEEE, 2007, pp. 13–22.
- [25] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *Proceedings of ESEC/FSE'05*. ACM, 2005, pp. 263–272.
- [26] S. Horwitz, T. W. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.
- [27] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 32:1–32:41, 2013.
- [28] C. Lecoutre, *Constraint Networks: Targeting Simplicity for Techniques and Algorithms*. Wiley, 2009.
- [29] G. Redelinghuys, W. Visser, and J. Geldenhuys, "Symbolic execution of programs with strings," in *Proceedings of SAICSIT'12*. ACM, 2012, pp. 139–148.
- [30] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang, "JST: An automatic test generation tool for industrial Java applications with strings," in *Proceedings of ICSE'13*. IEEE, 2013, pp. 992–1001.
- [31] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
- [32] H. J. Greenberg, "Consistency, redundancy, and implied equalities in linear systems," *Ann. of Math. and Artif. Intell.*, vol. 17, no. 1, pp. 37–83, 1996.
- [33] K. L. McMillan, "An interpolating theorem prover," *Theor. Comput. Sci.*, vol. 345, no. 1, pp. 101–121, 2005.
- [34] P. D. L. and M. A. K., *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010.
- [35] G. Redelinghuys, "Symbolic string execution," Ph.D. dissertation, Stellenbosch: Stellenbosch University, 2012.
- [36] A. Christensen, A. Møller, and M. Schwartzbach, "Precise analysis of string expressions," in *Proceedings of SAS'03*. Springer, 2003, pp. 1–18.
- [37] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.
- [38] Y. Zhang and R. H. C. Yap, "Arc consistency on n-ary monotonic and linear constraints," in *Proceedings of CP'02*. Springer, 2000, pp. 470–483.
- [39] T. Hickey, Q. Ju, and M. H. Van Emden, "Interval arithmetic: From principles to implementation," *J. ACM*, vol. 48, no. 5, pp. 1038–1068, 2001.
- [40] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, 2010.
- [41] M. Dorigo and M. Birattari, "Ant colony optimization," in *Encycl. of Mach. Learn.* Springer, 2010, pp. 36–39.
- [42] T. Stützle and H. H. Hoos, "Max-min ant system," *Future Gener. Comput. Syst.*, vol. 16, no. 9, pp. 889–914, 2000.
- [43] G. A. Croes, "A method for solving traveling-salesman problems," *Oper. Res.*, vol. 6, no. 6, pp. 791–812, 1958.
- [44] B. Korel, "Automated software test data generation," *Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.
- [45] G. Navarro, "A guided tour to approximate string matching," *Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.
- [46] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Softw. Test. Verif. Reliab.*, vol. 16, no. 3, pp. 175–203, 2006.
- [47] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *Proceedings of ICST'10*. IEEE, 2010, pp. 205–214.
- [48] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of Web queries," in *Proceedings of ICST'12*. IEEE, 2012, pp. 141–150.
- [49] S. Lin, "Computer solutions of the traveling salesman problem," *Alcatel-Lucent Bell Syst. Tech. J.*, vol. 44, no. 10, pp. 2245–2269, 1965.
- [50] IBM, "T. J. Watson Libraries for Analysis (WALA)," <http://wala.sourceforge.net/>, 2017.
- [51] C. Hammer, "Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs," Ph.D. dissertation, Karlsruhe Institute of Technology, 2009.
- [52] B. Naveh, "JGraphT," <https://github.com/jgrapht/jgrapht>, 2017.
- [53] OWASP, "OWASP WebGoat project," [https://www.owasp.org/index.php/Category:OWASP\\_WebGoat\\_Project](https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project), 2017.
- [54] S. Bennetts, "The bodgeit store," <https://github.com/psiinon/bodgeit>, 2017.
- [55] Apache, "Apache Roller blogging application," <http://roller.apache.org/>, 2017.
- [56] Pebble, "A lightweight, open source, Java EE blogging tool," <http://pebble.sourceforge.net/>, 2017.
- [57] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proceedings of USENIX Security*. USENIX Association, 2005.
- [58] Y. Liu and A. Milanova, "Practical static analysis for inference of security-related program properties," in *Proceedings of ICPC'09*. IEEE, 2009, pp. 50–59.
- [59] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "ASIDE: IDE support for Web application security," in *Proceedings of ACSAC'11*. ACM, 2011, pp. 267–276.
- [60] A. Møller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," *Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 29:1–29:30, 2014.
- [61] O. community, "OpenMRS legacy user interface module," <https://github.com/openmrs/openmrs-module-legacyui>, 2017.
- [62] —, "OpenMRS project," <http://openmrs.org/>, 2017.
- [63] Regain, "Regain search engine," <http://regain.sourceforge.net/>, 2017.
- [64] PubSubHubbub, "A simple, open, webhooks based pubsub protocol & open source reference implementation," <https://code.google.com/p/pubsubhubbub/>, 2017.

- [65] K. Safar, “rest-auth-proxy,” <https://github.com/kamranzafar/rest-auth-proxy>, 2017.
- [66] N. Antunes and M. Vieira, “Assessing and comparing vulnerability detection tools for Web services: Benchmarking approach and examples,” *Trans. Serv. Comput.*, vol. 8, no. 2, pp. 269–283, 2015.
- [67] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for PHP,” in *Proceedings of TACAS’10*. Springer, 2010, pp. 154–157.
- [68] T. Tateishi, M. Pistoia, and O. Tripp, “Path- and index-sensitive string analysis based on monadic second-order logic,” *Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 33:1–33:33, 2013.
- [69] IBM, “IBM Security Scanner: AppScan,” <http://www-03.ibm.com/software/products/en/appscan-source>, 2017.
- [70] S. Kausler and E. Sherman, “Evaluation of string constraint solvers in the context of symbolic execution,” in *Proceedings of ASE’14*. ACM, 2014, pp. 259–270.
- [71] T. Brennan, N. Tsiskaridze, N. Rosner, A. Aydin, and T. Bultan, “Constraint normalization and parameterized caching for quantitative program analysis,” in *Proceedings of ESEC/FSE’17*. ACM, 2017, pp. 535–546.
- [72] L. Luu, S. Shinde, P. Saxena, and B. Demsky, “A model counter for constraints over unbounded strings,” in *Proceedings of PLDI’14*. New York, NY, USA: ACM, 2014, pp. 565–576.
- [73] NIST, “NIST: National vulnerability database,” <https://nvd.nist.gov/>, 2017.
- [74] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger - An Automata-Based PHP String Analysis Tool,” <https://vlab.cs.ucsb.edu/stranger/>, 2018.
- [75] OMG, “BPMN 2.0 specification,” <http://www.bpmn.org>, January 2011.
- [76] OASIS, “Web Services Business Process Execution Language Version 2.0,” 2007.
- [77] SpringSource, “The Spring Framework,” <https://spring.io/>, 2017.
- [78] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: a static analysis tool for detecting Web application vulnerabilities,” in *Proceedings of S&P’06*. IEEE, 2006, pp. 6 pp.–263.
- [79] W. Halfond, A. Orso, and P. Manolios, “WASP: Protecting Web applications using positive tainting and syntax-aware evaluation,” *Trans. Softw. Eng.*, vol. 34, no. 1, pp. 65–81, 2008.
- [80] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in Web applications,” in *Proceedings of S&P’08*. IEEE, 2008, pp. 387–401.
- [81] G. Wassermann and Z. Su, “Sound and precise analysis of Web applications for injection vulnerabilities,” in *Proceedings of PLDI’07*. ACM, 2007, pp. 32–41.
- [82] —, “Static detection of cross-site scripting vulnerabilities,” in *Proceedings of ICSE’08*. ACM, 2008, pp. 171–180.
- [83] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, “Optimal sanitization synthesis for Web application vulnerability repair,” in *Proceedings of ISSA’16*. ACM, 2016, pp. 189–200.
- [84] F. Yu, M. Alkhalaf, and T. Bultan, “Patching vulnerabilities with sanitization synthesis,” in *Proceedings of ICSE’11*. ACM, 2011, pp. 251–260.
- [85] P. Saxena, D. Molnar, and B. Livshits, “SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications,” in *Proceedings of CCS’11*. New York, NY, USA: ACM, 2011, pp. 601–614.
- [86] M. Samuel, P. Saxena, and D. Song, “Context-sensitive auto-sanitization in web templating languages using type qualifiers,” in *Proceedings of CCS’11*. New York, NY, USA: ACM, 2011, pp. 587–600.
- [87] P. Bisht and V. N. Venkatakrishnan, “XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks,” in *Proceedings of DIMVA’08*. Berlin, Heidelberg: Springer, 2008, pp. 23–43.
- [88] C.-A. Staicu, M. Pradel, and B. Livshits, “Synode: Understanding and automatically preventing injection attacks on Node.js,” in *Proceedings of NDSS 2018*, 2018.
- [89] Acunetix, “Acunetix: Web vulnerability scanner,” <https://www.acunetix.com/>, 2017.
- [90] PortSwigger Ltd., “Burp Suite: toolkit for Web application security testing,” <https://portswigger.net/burp/>, 2017.
- [91] A. Aquino, G. Denaro, and M. Pezzè, “Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions,” in *Proceedings of ICSE’17*. IEEE, 2017, pp. 427–437.
- [92] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, “String analysis for Java and Android applications,” in *Proceedings of ESEC/FSE’15*. ACM, 2015, pp. 661–672.
- [93] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, “Automata-based symbolic string analysis for vulnerability detection,” *Form. Methods Syst. Des.*, vol. 44, no. 1, pp. 44–70, 2014.
- [94] A. Aydin, L. Bang, and T. Bultan, “Automata-based model counting for string constraints,” in *Proceedings of CAV’15*. Springer, 2015, pp. 255–272.
- [95] P. Hooimeijer and W. Weimer, “StrSolve: solving string constraints lazily,” *Autom. Softw. Eng.*, vol. 19, no. 4, pp. 531–559, 2012.
- [96] G. Li and I. Ghosh, “PASS: String solving with parameterized array and interval automaton,” in *Proceedings of HVC’13*. Springer, 2013, pp. 15–31.
- [97] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, “Norn: An SMT solver for string constraints,” in *Proceedings of CAV’15*. Springer, 2015, pp. 462–469.
- [98] M.-T. Trinh, D.-H. Chu, and J. Jaffar, “S3: A symbolic string solver for vulnerability detection in Web applications,” in *Proceedings of CCS’14*. ACM, 2014, pp. 1232–1243.
- [99] P. Dinges and G. Agha, “Solving complex path conditions through heuristic search on induced polytopes,” in *Proceedings of FSE’14*. ACM, 2014, pp. 425–436.
- [100] G. Fraser, A. Arcuri, and P. McMinn, “A “memetic” algorithm for whole test suite generation,” *J. Syst. Softw.*, vol. 103, pp. 311–327, 2015.