5-2017

Search-driven string constraint solving for vulnerability detection

Julian THOME

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

Domenico BIANCULLI

Lionel BRIAND

Search-driven String Constraint Solving for Vulnerability Detection

Julian Thomé, Lwin Khin Shar, Domenico Bianculli, Lionel Briand
SnT Centre - University of Luxembourg
Email:julian.thome@uni.lu, lwinkhin.shar@uni.lu, domenico.bianculli@uni.lu, lionel.briand@uni.lu

***Abstract*—Constraint solving is an essential technique for detecting vulnerabilities in programs, since it can reason about input sanitization and validation operations performed on user inputs. However, real-world programs typically contain complex string operations that challenge vulnerability detection. State-of-the-art string constraint solvers support only a limited set of string operations and fail when they encounter an unsupported one; this leads to limited effectiveness in finding vulnerabilities.**

In this paper we propose a search-driven constraint solving technique that *complements* the support for complex string operations provided by any existing string constraint solver. Our technique uses a hybrid constraint solving procedure based on the Ant Colony Optimization meta-heuristic. The idea is to execute it as a fallback mechanism, only when a solver encounters a constraint containing an operation that it does not support.

We have implemented the proposed search-driven constraint solving technique in the *ACO-Solver* tool, which we have evaluated in the context of injection and XSS vulnerability detection for Java Web applications. We have assessed the benefits and costs of combining the proposed technique with two state-of-the-art constraint solvers (*Z3-str2* and *CVC4*). The experimental results, based on a benchmark with 104 constraints derived from nine realistic Web applications, show that our approach, when combined in a state-of-the-art solver, significantly improves the number of detected vulnerabilities (from 4.7% to 71.9% for *Z3-str2*, from 85.9% to 100.0% for *CVC4*), and solves several cases on which the solver fails when used stand-alone (46 more solved cases for *Z3-str2*, and 11 more for *CVC4*), while still keeping the execution time affordable in practice.

***Keywords*-vulnerability detection, string constraint solving, search-based software engineering**

I. Introduction

Malicious users can attack Web applications by providing in input properly-crafted strings that can exploit vulnerabilities in source code; a successful attack may lead to leaking sensitive user data or to a denial of service. According to the Open Web Application Security Project (OWASP [1]), two of the most critical types of vulnerability are injection (ranked #1) and XSS-cross-site scripting (ranked #3) vulnerabilities. Both types of vulnerability are caused by the improper use of user input strings in security-sensitive program statements: often these strings are not property validated or sanitized.

State-of-the-art approaches [2]–[5] for identifying these types of vulnerability are based on symbolic execution and constraint solving. Roughly speaking, these approaches consist of solving the constraints corresponding to the attack condition, obtained by conjoining the path conditions generated by the symbolic execution with attack specifications provided by security experts. If the solver yields SAT, showing the satisfiability of the attack condition, it means that the attack is feasible and that the analyzed path is vulnerable to the attack. 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 these approaches are challenged by the *degree of support for (complex) string operations provided by the constraint solver itself*. State-of-the-art solvers such as Kaluza [3], Stranger [6], CVC4 [7], S3 [8], and Z3-str2 [9] support only a limited number of strings operations, such as concatenation, assignment, and equality; more complex operations like string replacement or standard sanitization functions are not supported or only partially-supported. Existing solvers could be extended to provide native support for complex string operations, but the task is non-trivial and not scalable to the size of a complete string function library of a modern programming language, or of sanitization libraries like OWASP ESAPI [10] and Apache Commons Lang [11]; for example, the classes `String`, `StringBuffer`, `StringBuilder` from the Java Standard Library, and the classes `StringUtils`, `StringEscapeUtils` from the Apache Commons Lang library contain a total of 370 methods.

Alternatively, complex string operations could be transformed into a set of equivalent constraints with only operations natively-supported by the solver; however, such a solution would increase the complexity of the generated constraints, potentially leading to scalability issues [4]. In practice, existing solvers fail (i.e., they crash or return an error) when they encounter an unsupported operation; in the context of vulnerability detection, this behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

In the context of vulnerability detection, the goal of this paper is to address the challenge of supporting complex operations in string constraint solvers by proposing a search-driven constraint solving technique that *complements* them. We intentionally target a solution that does not rely on any assumption regarding the selected constraint solver and that can therefore be widely used in the future.

The idea is to solve the constraints (in an attack condition) through a two-stage process. In the first stage we take any constraint solver and use it to solve the constraints that contain

only operations supported by the solver itself. The remaining unsolved constraints, which contain operations not supported by the solver, are handled in the second stage, by means of a *hybrid* constraint solving procedure.

We treat the solver in the first stage as a black-box, and only assume that it terminates its execution either by failing (when it encounters a constraint containing an operation that it does not support) or by returning an answer (which can be either UNSAT or SAT and a solution). The hybrid solving procedure in the second stage is executed only when the solver in the first stage fails. In the second stage we solve the constraints containing unsupported operations by means of a *hybrid* search-driven procedure that leverages the *Ant Colony Optimization* meta-heuristic [12]. This procedure searches for a solution that satisfies the constraints involving unsupported operations; the search is driven by different fitness functions, depending on the type of the constraints. We call this procedure *hybrid* because we reduce its search space before running the search itself, to make the latter scalable. We perform the search space reduction by restricting the domains of the string variables involved in the constraints to solve. To do so, in our current strategy and given the state of the art, we rely on an automata-based string constraint solver (*Sushi* [4]).

We have implemented the proposed search-driven constraint solving technique in a tool called ACO-Solver [13]. We have evaluated the proposed technique in the context of injection and XSS vulnerability detection for Java Web applications. More specifically, we have assessed the benefits and costs of combining the proposed technique with two state-of-the-art string constraint solvers (*Z3-str2* and *CVC4*), on a benchmark with 104 constraints derived from nine realistic Web applications. The experimental results show that the proposed approach, when combined with a state-of-the-art solver, significantly improves the number of detected vulnerabilities (from 4.7% to 71.9% for *Z3-str2*, from 85.9% to 100.0% for *CVC4*), and solves several cases on which the solver fails when used stand-alone (46 more solved cases for *Z3-str2*, and 11 more for *CVC4*); both benefits can be obtained while still keeping the execution time reasonable, in the order of minutes. Furthermore, we have also assessed the role played by the automata-based solver in the search space reduction step that precedes the meta-heuristic search: the results confirm that it contributes to increasing the number of solved cases.

The overall results confirm that our search-driven approach for string constraint solving, when combined with existing solvers, adds significant benefits in terms of cases that are solved, while keeping the cost (in terms of computation time) affordable in practice. We remark that these results should be interpreted in the *specific context of vulnerability detection*, and cannot (and do not aim to) be extrapolated to the more general case of string constraint solving.

In summary, the main contributions of this paper are: 1) a search-driven technique for solving string constraints with complex string operations, in the context of vulnerability detection; 2) the empirical assessment of the benefits and costs of adding the proposed technique to two state-of-the-art string constraint solvers, evaluated in the context of injection and XSS vulnerability detection for Java Web applications.

The rest of this paper is structured as follows. Section II provides some background on the concepts and techniques used in the rest of the paper. Section III discusses the motivations for this work and illustrates a running example. Section IV illustrates our search-driven approach for string constraints solving. Section V presents the evaluation of our approach. Section VI discusses related work. Section VII concludes the paper and gives directions for future work.

II. Preliminaries

A. Automata-based String Constraint Solving

In this paper we focus on string constraints, i.e., constraints on string variables that can be involved in string-manipulating functions. An example of such a constraint is *ID*`.contains("ul")`, where *ID* is a string variable and `contains(...)` is a string-manipulating function from the Java class library. A *solution* for this constraint is a string value for variable *ID* that satisfies the constraint; for example, the string "`module`" is a solution for the above constraint.

One of the ways to solve string constraints is to consider their automata-based representation [4], [14]–[18]. More precisely, an automata-based string constraint solver uses finite-state machines (FSM) to encode the set of string values that a string variable can take. String operations are modeled through automata operations, such as concatenation and intersection. If the constraint is satisfiable, the solver returns SAT and yields a solution automaton for each string variable; this automaton accepts string values that satisfy the constraint in which the variable is involved.

For example, an automata-based constraint solver that supports, among others, the `contains` operation will solve the constraint above and will return for variable *ID* the FSM $M_{ID} =$ `.*ul.*`, i.e., the automaton that accepts string values containing the `ul` substring. Notice that we abuse the notation by defining an FSM by means of the regular expression characterizing the regular language it accepts.

B. Ant Colony Optimization

Ant Colony Optimization (ACO) [12] is one of the most 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 numerical 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.

III. Motivations and Running Example

In this section we present a motivating example that highlights the need to handle complex string constraints in the context of vulnerability detection based on constraint solving. We will also use it as a running example in the rest of the paper. 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.

The program, shown in Figure 1, contains a security-sensitive operation (i.e., a sink) at line 18, which corresponds to an XPath injection (XPathi) vulnerability within an XPath query. It 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 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`, one could still perform a successful attack without using those meta-characters, for example using the input `0 or 1`. This vulnerability can be discovered by using the following three-step vulnerability detection procedure:

1) *Path conditions generation through symbolic execution*. One of the path conditions generated by symbolically executing a path leading to the execution of the sink at line 18 is:

$$PC_{18} \equiv SUBJ\texttt{.trim().substring(0,2).equals("cd")} \wedge \texttt{Integer.parseInt(}MAX\texttt{)} \leq 20 \wedge OP\texttt{.trim().equalsIgnoreCase("GradeQuery")} \wedge SID\texttt{.toLowerCase().length()} \leq \texttt{Integer.parseInt(}MAX\texttt{)} \wedge SID\texttt{.contains("id")}$$

where $SUBJ, MAX, OP, SID$, are symbolic values for the variables initialized with the Web request inputs (lines 5–8).

2) *Definition of the attack specification*. In this step, the values at the sink are checked against an attack specification defined by a security expert (usually by means of threat catalogues and attack libraries). Attack specifications are defined in a way that properly characterizes security threats. For example, the following specification characterizes a security threat (in the form of a tautology attack) for variable *sid*:

$$ATTK \equiv \texttt{ESAPI.encoder().encodeForXPath(}SID\texttt{.toLowerCase()).matches("(0|[1-9][0-9]*) (o|O)(r|R) [1-9][0-9]*")}$$

```
1  protected void doPost(HttpServletRequest req,
2                        HttpServletResponse res) {
3    Document doc=getStudentDB("./students.xml");
4    XPath xpath=XPathFactory.newInstance().newXPath();
5    String op=req.getParameter("option");
6    String sid=req.getParameter("id");
7    int max=Integer.parseInt(req.getParameter("max"));
8    String subj=req.getParameter("subjid");
9    sid=sid.toLowerCase();
10   if(!subj.trim().substring(0,2).equals("cd")) {subj="*"; }
11   if(max>20) { max=20; }
12   if(op.trim().equalsIgnoreCase("GradeQuery")) {
13     if(sid.length()<=max & sid.contains("id")) {
14         sid=ESAPI.encoder().encodeForXPath(sid);
15         subj=subj.replaceAll("'|\"|\\", "");
16         String query="//students/grade[sid="
17                      +sid+"and subjid='"+subj+"']/mark";
18         NodeList nl=(NodeList)xpath.evaluate(query, doc);
19     } } }
```

Figure 1: A Java servlet program vulnerable to XPathi

where `ESAPI.encoderForXPath(`SID`.toLowerCase())` is the symbolic expression over the symbolic value *SID* representing the values of variable *sid* at the sink. The expression `matches("(0|[1-9][0-9]*) (o|O)(r|R) [1-9][0-9]*")` describes a tautology attack pattern.

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 attack condition $AC_1 \equiv PC_{18} \wedge ATTK$ is satisfiable, confirming the presence of the vulnerability.

This procedure assumes that the constraint solver is able to handle string operations like `trim`, `toLowerCase`, `parseInt`, `equalsIgnoreCase`, `length`, `replaceAll`, and `encodeForXPath`. However, state-of-the-art solvers such as Kaluza [3], CVC4 [7], S3 [8], and Z3-str2 [9] do not support at least one of these complex operations. From a more general standpoint, the major challenge faced when adopting a vulnerability detection procedure based on constraint solving is the *degree of support for (complex) string operations provided by the constraint solver itself*.

One way to face this challenge is to modify or enhance an existing solver in order to provide native support for complex string operations. 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 solver, 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 significant effort and expertise, and usually results in complex constraints that may still lead to scalability issues for constraint solvers [4]. For example, consider one of the constraints in the above path condition: OP`.trim().equalsIgnoreCase("GradeQuery");` as-

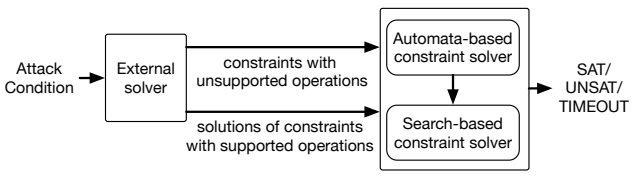

Figure 2: Two-stage approach for string constraint solving

suming the solver handles only `length`, `charAt`, `equals`, and `substring`, one could re-express this constraint as:

$\exists c_1, c_2, 0 \leq c_1 \leq c_2 \leq OP.\texttt{length()}$, such that
$(OP.\texttt{substring}(c_1,c_2).\texttt{equals("gradequery")}$
$\vee \cdots \vee \ldots\texttt{equals("gRadeQueRy")}\ldots \vee \cdots \vee$
$OP.\texttt{substring}(c_1,c_2).\texttt{equals("GRADEQUERY")})$
$\wedge \forall i, 0 \leq i < c_1, OP.\texttt{charAt}(i) = \texttt{' '}$
$\wedge \forall j, c_2 < j \leq OP.\texttt{length()}, OP.\texttt{charAt}(j) = \texttt{' '}$

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

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* and *Z3-str2*. However, in the context of vulnerability detection, such a behavior could yield false negatives (i.e., it misses some vulnerabilities) when the attack conditions are actually feasible.

Hence, the challenge discussed above shows that, in the context of vulnerability detection, there is the need for scalable and precise techniques for constraint solving that can handle constraints with complex string operations.

IV. Search-driven String Constraint Solving

We address the challenge of supporting complex operations in string constraint solvers — in the context of vulnerability detection — by proposing a search-driven constraint solving technique that *complements* their support for complex string operations.

The idea, illustrated in Figure 2, is to solve the constraints corresponding to an attack condition *AC* through a two-stage process. In the first stage we take any existing constraint solver and use it to solve the conjuncts in *AC* that contain only operations supported by the solver itself. The remaining conjuncts in *AC*, which contain unsupported operations, are solved in the second stage, by means of a *hybrid* constraint solving procedure that combines an automata-based solver with a search-driven solving procedure based on the Ant Colony Optimization meta-heuristic.

```
1: function CstrSolve(AttackCondition AC)
2:     Set of Solution Sol ← ∅
3:     Boolean externalSolved ← false
4:     Set of Set of Constraint H ← getDependentSetsOfCstrs(AC)
5:     for all H_i ∈ H do
6:         ⟨externalSolved, Sol⟩ ← externalSolve(H_i)
7:         if ¬externalSolved then
8:             ⟨H'_i, Sol⟩ ← automataSolve(H_i)
9:             if H'_i ≠ ∅ then
10:                Sol ← searchSolve(H'_i, Sol)
11:                if Sol = ∅ then
12:                    return TIMEOUT
13:                end if
14:            end if
15:        end if
16:    end for
17:    return ⟨SAT, Sol⟩
18: end function
```

Figure 3: Search-driven string constraint solving algorithm

The meta-heuristic search in the second stage tries to find solutions for the variables involved in conjuncts of *AC* that contain operations that neither the solver in the first stage nor the automata-based solver in the first step of the second stage supports. Nevertheless, we invoke the automata-based solver *before* the meta-heuristic search in order to *reduce* the search space of the latter. We specifically use an automata-based (vs. bit-vector-based or word-based) constraint solver because it returns, when successful, a solution automaton for each variable occurring in the constraints it could solve, based on the operations it supports. This automaton accepts the language corresponding to the set of values (for the variable) that satisfy the constraints involving the operations that the solver supports. In this way, we are able to reduce (possibly in a significant way) the size of the domains of the variables involved in the constraint; this is expected to make the search more scalable and effective.

A. Overview

The pseudocode of our string constraint solving algorithm is shown in Figure 3. It takes in input an attack condition *AC* (expressed as a conjunction of constraints) and returns whether it is satisfiable, unsatisfiable, or whether it timed out; when it returns satisfiable, it also returns the set of solutions found.

First, it decomposes (line 4) the attack condition *AC* into the set H of sets of dependent constraints. More specifically, function getDependentSetsOfCstrs identifies the connected sub-hypergraphs (i.e., the maximal connected components) of the hypergraph equivalent to the constraint network [17], [19] representing the attack condition; each set $H_i \in H$ corresponds to a sub-hypergraph. For the attack condition AC_1 in our running example, we have $H = \{H_1, H_2, H_3\}$, with $H_1 = \{cstr_1\}$, $H_2 = \{cstr_2\}$, $H_3 = \{cstr_3, cstr_4, cstr_5, cstr_6\}$, where:

$cstr_1 \equiv SUBJ.\texttt{trim().substring(0,2).equals("cd")}$
$cstr_2 \equiv OP.\texttt{trim().equalsIgnoreCase("GradeQuery")}$
$cstr_3 \equiv SID.\texttt{toLowerCase().length()} \leq \texttt{Integer.parseInt(}MAX\texttt{)}$
$cstr_4 \equiv \texttt{Integer.parseInt(}MAX\texttt{)} \leq 20$
$cstr_5 \equiv SID.\texttt{contains("id")}$
$cstr_6 \equiv \texttt{ESAPI.encoder().encodeForXPath(}SID\texttt{.toLowerCase())}$
`.matches("(0|[1-9][0-9]*) (o|O)(r|R) [1-9][0-9]*")`

Next, the algorithm iterates through the sets of constraints H_i in H, performing the following steps (lines 5–16).

First, it calls function EXTERNALSOLVE (line 6), which invokes the external solver and returns a tuple $\langle externalSolved, Sol \rangle$. If the external solver supports all the operations used in the constraints contained in H_i, it will return a true value for the flag *externalSolved* and the set *Sol* will contain a solution for each variable involved in the constraints in H_i; the algorithm can then proceed to the next iteration of the loop, to process the set H_{i+i}. Otherwise, in case the external solver does not support an operation used in a constraint in H_i, it will fail and EXTERNALSOLVE will set the flag *externalSolved* to false and *Sol* to the empty set.

When the flag *externalSolved* is false, the algorithm enters the second stage of our approach. It calls function AUTOMATASOLVE (line 8), which internally invokes the automata-based string constraint solver *Sushi* [4] to solve the constraints in H_i that use operations it supports (i.e., `concat`, `contains`, `equals`, `trim`, `substring`, `replace`, `replaceAll`, and `matches`). If a constraint is satisfiable, *Sushi* yields a solution automaton for each string variable involved in the constraint. Function AUTOMATASOLVE returns a tuple $\langle H_i', Sol \rangle$. The set $H_i' \subseteq H_i$ contains the constraints in H_i that could not be solved by *Sushi* because they use unsupported operations. The set *Sol* contains the solution automata for the variables involved in the constraints in H_i: if a variable was involved only in constraints with unsupported operations, its corresponding solution automaton is the default one, accepting any string (i.e., the automaton accepting the regular language $.*$); otherwise, the solution automaton is the one determined by *Sushi*. Notice that both EXTERNALSOLVE and AUTOMATASOLVE internally terminate the entire constraint solving procedure and return UNSAT, without proceeding to the following steps, when they detect unsatisfiable constraints.

Subsequently, if the set H_i' is not empty (meaning that there are unsolved constraints in H_i using unsupported operations), the algorithm invokes the SEARCHSOLVE function, which implements a meta-heuristic search algorithm (detailed in the next subsection) to solve the constraints in H_i' and returns an updated set of solutions *Sol* (line 10). If the set *Sol* is not empty, it means that the set of constraints H_i has been solved and the algorithm can proceed to process the set H_{i+i}; 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 security analyst then has to decide (possibly based on empirical studies) how to treat it.

The algorithm returns SAT and the set of solutions *Sol* (line 17) only when the loop over H has been completely executed, meaning that all the constraints in the sets in H are satisfiable, which is equivalent to say that the attack condition *AC* in input is satisfiable.

For our running example, the call to EXTERNALSOLVE with input H_1 will return $\langle true, \{SUBJ = \text{“}\texttt{cd}\text{”}\} \rangle$, meaning that the external solver was able to solve the constraint, determining the solution “`cd`” for variable *SUBJ*. However, the call to EXTERNALSOLVE with input H_2 and H_3 will return $\langle false, \emptyset \rangle$ because the external solver cannot handle some of the operations used in the constraints in H_2 and H_3 (e.g., the operations `equalsIgnoreCase`, `encodeForXPath`). This means that the constraints in H_2 and H_3 will be processed in the second stage. In particular, the calls to AUTOMATASOLVE will behave as follows. AUTOMATASOLVE$(H_2) = \langle \{cstr_2\}, \{M_{OP}\} \rangle$, with $M_{OP} = .*$, meaning that $cstr_2$ could not be solved by *Sushi* (because it contains an unsupported operation). AUTOMATASOLVE$(H_3) = \langle \{cstr_3, cstr_4, cstr_6\}, \{M_{SID}, M_{MAX}\} \rangle$, where $M_{SID} = .*\texttt{id}.*$ and $M_{MAX} = .*$, meaning that *Sushi* could only solve $cstr_5$ and determine a solution automaton for variable *SID*.

B. Solving Constraints using Meta-heuristic Search

In this subsection we illustrate how function SEARCHSOLVE works, by explaining how we use meta-heuristic search for solving string constraints that involve unsupported operations[1].

Our search procedure is based on the $\mathcal{MAX}$–$\mathcal{MIN}$Ant System proposed in [20]. In this algorithm the pheromone values are bounded by maximum and minimum values, which are dynamically computed after every search iteration. This avoids the relative differences between the pheromone values 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.

We chose ACO over other well-known meta-heuristic search techniques (such as hill climbing, simulated annealing, and genetic algorithms [21]) because:

- It has inherent parallelism in which multiple candidate solutions can be searched in parallel for efficiency.
- It is stochastic in nature, which allows for escaping from local optima.
- It is typically used for finding good solutions (i.e., paths that return good fitness values) in graphs [12]. 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, modifying a candidate solution to get a different one is straightforward, since it only requires having an ant exploring the solution automaton.

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

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

[1]In our implementation based on *Sushi*, function SEARCHSOLVE is also used to solve numeric constraints, which are also unsupported by *Sushi*. Nevertheless, we expect that most of the numeric constraints are already solved by the external solver in the first stage. Integrating a separate numeric constraint solver before the meta-heuristic search is part of future work.

of this work we deal with both numeric and string constraints, we use fitness functions specific to these domains.

For numeric constraints we use the Korel function [22], which is a standard fitness function for this domain. We consider numeric constraints of the form $C \equiv E_1 \bowtie E_2$, where $\bowtie \in \{=, \neq, <, \leq, >, \geq\}$ and E_1, E_2 are numeric expressions that can be numeric variables, numeric constants, or any other expression whose evaluation results in a numeric value (e.g., the `length` operation for strings); notice that we treat boolean expressions also as numeric expressions. Let $\mathbf{s} = [s_1, \ldots, s_n]$ be the vector of candidate solutions for the numeric variables $x_1, \ldots, x_n$ in C, and $a(\mathbf{s}), b(\mathbf{s})$ be the numeric values resulting from the evaluations of E_1 and E_2 respectively, after replacing the variables in them with the corresponding solutions in $\mathbf{s}$; the fitness of $\mathbf{s}$ is defined as:

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

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 [23] and the equality cost function for regular expression matching [24]; both functions have been shown to be useful for search-based generation of string values [24]. 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 $\mathbf{s} = [s_1, \ldots, s_n]$ be the vector of candidate solutions for the string variables $x_1, \ldots, x_n$ in C, and $a(\mathbf{s}), b(\mathbf{s})$ 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 $\mathbf{s}$; the fitness of $\mathbf{s}$ is defined as:

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

where ψ 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), and ψ is 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.

```
1: function SEARCHSOLVE(Set of Constraint H, Set of Solution-automaton K)
2:     Tuning-parameters ⟨α,β,ρ,ξ_max,ξ_min⟩ ← SETTUNINGPARAMS()
3:     Population-size A ← SETNUMBERANTS()
4:     Set of Desirability-value Δ ← SETDESIRABILITYVAL(K)
5:     Set of Pheromone Ξ ← SETPHEROMONES(K)
6:     Set of Solution-component T_Best ← ∅
7:     Fitness F_Best ← 1; Fitness F_pBest ← 1
8:     Array of Fitness tempF ← ∅; Array of Set of Solution-component tempT ← ∅
9:     repeat
10:        loop A times
11:            Set of Solution-component T ← CONSTRUCTSOLUTIONS(K,Δ,Ξ)
12:            Fitness F ← COMPUTEFITNESS(T,H)
13:            tempF ← APPEND(tempF,F); tempT ← APPEND(tempT,T)
14:        end loop
15:        ⟨F_Best,T_Best⟩ ← BESTSOLUTION(tempF,tempT)
16:        if F_Best < F_pBest then
17:            ⟨F_Best,T_Best⟩ ← 2OPTLOCALSEARCH(K,T_Best)
18:        end if
19:        UPDATEPHEROMONES(K,Ξ,F_Best,T_Best)
20:        F_pBest ← F_Best
21:    until F_Best = 0 or timeout
22:    if timeout then
23:        return ∅
24:    end if
25:    return T_Best
26: end function

27: function CONSTRUCTSOLUTIONS(Set of Solution-automaton K,
                                Set of Desirability-value Δ, Set of Pheromone Ξ)
28:    Set of Solution-component S ← ∅
29:    repeat
30:        Automaton k ← RANDOMSELECT(K)
31:        FSMState v ← GETSTARTSTATE(k)
32:        repeat
33:            Set of FSMTransition E ← GETOUTTRANSITIONS(v)
34:            FSMTransition e ← SELECTTRANSITION(E,Δ,Ξ)
35:            S ← S∪{e}
36:            v ← GETNEXTSTATE(e)
37:        until ISACCEPTSTATE(v)
38:        MARKASVISITED(k,K)
39:    until all the automata in K have been traversed
40:    return S
41: end function
```

Figure 4: Ant colony search for string constraint solving

2) Search Algorithm: The pseudocode of our instantiation of the ACO meta-heuristic for solving string constraints is shown in Figure 4. The main function SEARCHSOLVE takes in input a set of constraints H (corresponding to the constraints not solved by the automata-based solver in the first step) and a set of solution automata K (as determined by the automata-based solver for the solution of the constraints with supported operations).

The first steps of function SEARCHSOLVE (lines 2–5) 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 Δ of desirability values $\delta_e = 1$ for each transition e of each automaton in K; the set Ξ 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 [20] and on our own preliminary experiments. Notice that for each transition e, the parameter ξ_e is initialized

to the value ξ_{max}; as discussed in [20], 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 following three main steps (lines 9–21) until the termination conditions are met:

Construction of solutions. This step (lines 10–15) consists of three sub-steps:

1) *Building the set of solution components*. This step is represented by the call to function CONSTRUCTSOLUTIONS, which takes in input the set of solution automata K, the set of desirability values Δ, and the set of pheromones values Ξ. This function goes through (lines 29–39) 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. Upon reaching a state where there are multiple outgoing transitions, it selects (line 34) 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 ξ_e and the desirability δ_e of the transition. The selected transition is added to the local set of solution components S (line 35) and its reaching state is retrieved (line 36). The traversal/selection of the transitions of an automaton is repeated until the final state is found[2], 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, the function returns a set of solution components, with one solution for each string variable.

2) *Determining the fitness of solution components.* This step computes the fitness for the solution components identified in the previous step. Function COMPUTEFITNESS evaluates each constraint in H with the solution components, and computes the corresponding fitness f using one of the aforementioned fitness functions (depending on the type of constraint). To ensure that the search process is not biased towards solving the constraints with larger-scale fitness values, each fitness value f is normalized using the normalization function proposed in [25], resulting in a normalized fitness value $\hat{f} = f/(f+1)$. We use this normalization function since it has proven to be useful in the similar domain of search-based test input generation of string data types [26]. 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 $\hat{f}$.

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.

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

Application of local search. This step (lines 16–18) 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 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 [27], which finds other paths (or sets of solution components) in each automaton in K 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}.

Update of pheromone values. This step (line 19) 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 $\xi_e = (1-\rho)\xi_e + \triangle\xi_e$, where $\triangle\xi_e = \frac{1}{F_{Best}}$ 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 $\xi_e = \xi_{min}$.

The termination conditions of the loop at line 21 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 returns T_{Best}.

V. EVALUATION

We have implemented our search-driven string constraint solving approach for vulnerability detection in the *ACO-Solver* tool [13]. The tool is implemented in Java, uses *Sushi* as automata-based constraint solver in the second stage, and has a plugin architecture to support different solvers in the first stage; we have developed plugins for *CVC4* and *Z3-str2*.

In this section we report on the evaluation of *ACO-Solver* in the context of vulnerability detection for Java Web applications. We assess the benefits and costs of combining the proposed string constraint solving approach with two state-of-the-art solvers, by answering the following research questions:

RQ1: *How does the proposed approach improve the effectiveness of state-of-the-art solvers for solving constraints related to vulnerability detection?* (subsection V-B)

RQ2: *Is the cost (in terms of execution time overhead) of using our technique affordable in practice?* (subsection V-B)

RQ3: *Does the automata-based solver in the first step of the second stage of our approach contribute to the effectiveness of the search-based procedure?* (subsection V-C)

A. Benchmark and Evaluation Settings

To evaluate our approach in terms of vulnerability detection capability, we use a benchmark composed of nine realistic, open source Java Web applications/services, with known XSS,

XML, XPath, LDAP, and SQL injection vulnerabilities. *WebGoat* [28] is a deliberately in-secured Web application/service for the purpose of teaching security vulnerabilities in Web applications. *Roller* [29] and *Pebble* [30] are blogging applications that also expose Web service APIs. *WebGoat*, *Roller*, and *Pebble* have been already used as benchmarks in the vulnerability detection literature [31]–[36]. *Regain* [37] 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 [38] is the most popular Java project related to the *PubSubHubbub* protocol in the Google Code archive. The *rest-auth-proxy* (shortened as *RAP*) microservice [39] 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 [40] 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).

This benchmark contains in total 104 paths to sinks: 64 vulnerable paths and 40 non-vulnerable ones. We generated the corresponding 104 attack conditions using a Java program slicing and symbolic execution tool developed in our previous work [41]. For each attack condition, we established the ground truth (i.e., whether it is vulnerable or not) via manual inspection and consultation of the vulnerability report of the corresponding application in the US National Vulnerability Database (NVD) [42].

We conducted our evaluation on a machine equipped with an Intel Core i7 2.4 GHz processor, 8 GB memory, running Apple Mac OS X 10.11 and *Sushi* v2.0. We set the time-out for solving each attack condition to 30 s.

B. Effectiveness and Cost of Vulnerability Detection

We assess the benefits and costs of combining the proposed approach with two state-of-the-art string constraint solvers: *CVC4* (version 1.4) and *Z3-str2* (from the repository head, commit 2e52601). For each of these solvers, we run our benchmark first through the standalone solver and then through the solver *combined* with *ACO-Solver*.

The evaluation results are shown in Table I. Columns *vp* and *nvp* indicate, respectively, the number of vulnerable and non-vulnerable paths per application. Column *t* indicates the cumulative time taken to solve all the attack conditions of each application. Column × indicates the number of failing cases, i.e., the number of attack conditions that the solver failed to solve, due to an error or crashing; we omit this column for *Z3-str2 + ACO-Solver* and *CVC4 + ACO-Solver* since they did not fail. Column ⊘ indicates the number of cases in which the solver timed out; we omit this column for *Z3-str2* and *CVC4* since they did not time out. Column ✓ indicates the number of non-failing cases. Column Δ indicates the number of cases, out of the failing cases of *Z3-str2* or *CVC4*, that *ACO-Solver* helped solve. Columns *tp*, *tn*, *fp*, *fn*, and ∇ denote, respectively, true positives (number of vulnerable cases correctly identified), true negatives (number of non-vulnerable cases correctly identified), false positives (number of non-vulnerable cases reported as vulnerable), false negatives (number of vulnerable cases not detected), number of additional vulnerable cases uncovered by *ACO-Solver*. Column *pd* reports the *recall*, i.e., the percentage of vulnerable cases detected among the total vulnerable cases, and is computed as $pd = tp/(tp+fn) * 100$. Notice that, in the context of vulnerability detection, when a solver fails or times out to solve an attack condition, it neither detects a vulnerability nor produces a false alarm. Hence, a failing (or time-out) case may result either in a false negative or in a true negative, depending on whether the attack condition is actually vulnerable.

We answer RQ1 by examining the number of failing and time-out cases and the recall in Table I, first when using a solver standalone, and then when combined with *ACO-Solver*.

When used standalone, *Z3-str2* and *CVC4* could not solve, respectively, 85 and 32 cases. We manually inspected these failing cases and observed they are due to unsupported operations contained in the attack conditions. For example, both *Z3-str2* and *CVC4* do not handle some string operations (e.g., `toLowerCase`, `toUpperCase`, `equalsIgnoreCase`) and the sanitization operations of the standard Apache security library [11]. Also, *Z3-str2* was not able to handle `Integer.parseInt`, and `String.valueOf` conversions and many of the regular expressions that reflect security threats in our attack conditions. *Z3-str2* missed 61 vulnerable cases (out of 85 failing cases), resulting in a low recall of 4.7%. *CVC4* missed 9 vulnerable cases (out of 32 failing cases), resulting in a recall of 85.9%.

Z3-str2 + ACO-Solver helped solve 46 out of the 85 failing cases of *Z3-str2*, revealing 43 additional vulnerabilities. It timed out on 39 cases; however, 21 out of these 39 time-out cases are non-vulnerable cases (i.e., the corresponding attack condition is UNSAT) and thus the search is obviously expected to time out. *CVC4 + ACO-Solver* solved 11 failing cases of *CVC4*, revealing 9 additional vulnerabilities. It timed out on 21 cases; however, all of them are actually non-vulnerable ones. *Z3-str2 + ACO-Solver* improved the recall of *Z3-str2* from 4.7% to 71.9%. *CVC4 + ACO-Solver* improved the one of *CVC4* from 85.9% to 100.0%, detecting all vulnerabilities.

We remark that, while most of these vulnerabilities had already been reported to the NVD [42], we also discovered two new XSS vulnerabilities (one in *Regain* and one in *Pebble*) while performing this evaluation; we reported them to NVD and also to the corresponding developers. The vulnerability in *Regain* was detected by both *Z3-str2* and *CVC4*, used standalone; the one in *Pebble* was detected when both solvers were combined with *ACO-Solver*. Though not shown in Table I for space reasons, we also remark that both solvers achieved 100% precision (i.e., they reported no false positive).

The answer to RQ1 is that the proposed approach, when combined with a state-of-the-art solver, significantly improves the recall (from 4.7% to 71.9% for *Z3-str2*, from 85.9% to 100.0% for *CVC4*), and solves several cases on which the solvers failed when used stand-alone (46 more solved cases

Table I: Comparison of vulnerability detection effectiveness and execution time between standalone solvers (*Z3-str2* and *CVC4*) and the same solvers *combined* with *ACO-Solver* (*Z3-str2 + ACO-Solver* and *CVC4 + ACO-Solver*)

App	Paths		*Z3-str2*								*Z3-str2 + ACO-Solver*										*CVC4*								*CVC4 + ACO-Solver*									
	vp	*nvp*	*t*(s)	×	✓	*tp*	*tn*	*fp*	*fn*	*pd*	*t*(s)	⊘	✓	Δ	*tp*	*tn*	*fp*	*fn*	∇	*pd*	*t*(s)	×	✓	*tp*	*tn*	*fp*	*fn*	*pd*	*t*(s)	⊘	✓	Δ	*tp*	*tn*	*fp*	*fn*	∇	*pd*
WebGoat	11	4	0.10	11	4	0	4	0	11	0.0	22.23	0	15	11	11	4	0	0	11	100.0	1.40	1	14	10	4	0	1	90.9	10.90	0	15	1	11	4	0	0	1	100.0
Roller	3	10	0.00	13	0	0	10	0	3	0.0	333.96	10	3	3	3	10	0	0	3	100.0	0.53	10	3	3	10	0	0	100.0	307.24	10	3	0	3	10	0	0	0	100.0
Pebble	6	7	0.01	12	1	0	7	0	6	0.0	199.34	5	8	7	6	7	0	0	6	100.0	0.04	12	1	0	7	0	6	0.0	205.14	5	8	7	6	7	0	0	6	100.0
Regain	3	3	86.71	0	6	3	3	0	0	100.0	84.12	0	6	0	3	3	0	0	0	100.0	0.61	0	6	3	3	0	0	100.0	1.47	0	6	0	3	3	0	0	0	100.0
PSH	1	3	13.33	4	0	0	3	0	1	0.0	61.64	2	2	2	1	3	0	0	1	100.0	0.00	4	0	0	3	0	1	0.0	61.34	2	2	2	1	3	0	0	1	100.0
RAP	1	0	0.00	1	0	0	0	0	1	0.0	0.40	0	1	1	1	0	0	0	1	100.0	0.00	1	0	0	0	0	1	0.0	0.93	0	1	1	1	0	0	0	1	100.0
TPC-APP	6	6	0.02	10	2	0	6	0	6	0.0	217.79	7	5	3	2	6	0	4	2	33.3	0.57	3	9	6	6	0	0	100.0	93.22	3	9	0	6	6	0	0	0	100.0
TPC-C	30	4	0.09	31	3	0	4	0	30	0.0	596.40	15	19	16	16	4	0	14	16	53.3	1.50	1	33	30	4	0	0	100.0	47.12	1	33	0	30	4	0	0	0	100.0
TPC-W	3	3	0.02	3	3	0	3	0	3	0.0	2.45	0	6	3	3	3	0	0	3	100.0	0.31	0	6	3	3	0	0	100.0	1.21	0	6	0	3	3	0	0	0	100.0
Total	64	40	100.28	85	19	3	40	0	61	4.7	1,518.33	39	65	46	46	40	0	18	43	71.9	4.96	32	72	55	40	0	9	85.9	728.57	21	83	11	64	40	0	0	9	100.0

for *Z3-str2*, and 11 more for *CVC4*). Hence, combining a state-of-the-art solver with our approach proved to be very effective to vulnerability detection. Since time-outs with *CVC4 + ACO-Solver* are all unsatisfiable/non-vulnerable cases, if such results were to be confirmed by additional benchmarks, then one could conclude that the most cost-effective and realistic decision strategy for the security analyst would be to treat time-outs as non-vulnerable cases.

To answer RQ2, we compare the execution time (t) for running the standalone solvers to the one for running the solvers combined with *ACO-Solver*. The total execution of *Z3-str2* took less than two minutes and solved 19 cases; *Z3-str2 + ACO-Solver* took about 25 minutes and solved 65 cases; the execution of *CVC4* took about five seconds and solved 72 cases; *CVC4 + ACO-Solver* took about 12 minutes and solved 83 cases. Despite the increase in terms of absolute values, the total execution time of our approach is still affordable, considering that 1) it can handle many cases that would otherwise fail, and thus can detect more vulnerabilities; 2) vulnerability detection is typically an offline activity, with no real-time requirements. Hence, we answer RQ2 positively.

C. The Role of the Automata-based Solver

To address RQ3 and thus investigate the role played by the automata-based solver in reducing the search space explored by the meta-heuristic search, we used a modified implementation of our approach. We switched off the automata-based solver in the first step of the second stage, meaning that the search-based algorithm is executed using a set of solution automata that accept any string and thus has a much larger search space.

We run our benchmark through both solvers, combined with this *modified* version of *ACO-Solver*, which does not call *Sushi* internally. Because of space reasons, we present only the results of running *CVC4* combined with this modified *ACO-Solver*; the results for *Z3-str2* are similar.

When executed with 30 s time-out, *CVC4* + modified *ACO-Solver* timed out on 31 cases, with a recall of 87.5% and an overall execution time of 15.5 min. Therefore, the version of *ACO-Solver* without *Sushi* helped solve only 1 more case, whereas the unmodified *ACO-Solver* helped solve 11 more cases. Since in this scenario we expected the search to explore a larger search space, we also ran the solver with an increased time-out of 300 s; however, we obtained the same results as above in terms of solved cases and recall, but the execution time increased to almost three hours.

From the above results, we answer RQ3 by saying that the automata-based solver plays a fundamental role in achieving a higher effectiveness, since it contributes to reducing the number of failing cases and increasing the recall.

D. Verifiability and Threats to Validity

Verifiability: The applications composing the benchmark, the related attack conditions, the instructions and scripts to obtain the *ACO-Solver* tool and run the benchmark, and the detailed evaluation results are available on our website [13].

Threats to Validity: Our results are based on solving the constraints corresponding to attack conditions extracted from a specific benchmark; hence, they cannot necessarily be generalized to all types of constraints. We minimized this threat by choosing applications that vary in functionality and by sampling realistic projects, which in many cases represent well-known benchmarks in the context of vulnerability detection. There are other benchmarks (e.g., the one used in [43] and the Kaluza suite [3]) that are widely-used for comparing constraint solvers. However, they are not specific to the security domain (e.g., they are not annotated with vulnerability information), and thus the constraints they contain cannot be used to assess the effectiveness of a solver in terms of vulnerability detection. Furthermore, we remark that our results should be interpreted in the *specific context of vulnerability detection*, and cannot (and do not aim to) be extrapolated to the more general case of string constraint solving.

As shown in subsection V-C, the role of an automata-based solver is essential for our approach in order to reduce the input domains and scale the meta-heuristic search process. Instead of *Sushi*, which supports only basic operations, we could use other, more powerful automata-based solvers like Stranger [6] and JST [18], which support more operations. Nevertheless, *Sushi* was available from the authors, fully functional, and yielded a significant reduction in search space that was sufficient to make the approach practical. By using an automata-

based solver with support for a larger set of operations, we expect a reduction of the time taken by the meta-heuristic search, since it will have to explore a smaller search domain. Hence, our results should be interpreted as the lowest bound for our search-driven constraint solving approach.

VI. Related Work

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

Constraint solving through heuristic search. Heuristic search has been already proposed [44] 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. Contrastingly, our approach targets solving string constraints with unsupported, string-manipulating operations and its search heuristics is optimized, in terms of search strategy and fitness functions, for string constraints. Further, the approach in [44] is evaluated in terms of coverage of test generators, while we evaluated our approach in the context of vulnerability detection.

(String) constraint solving. There are many constraint solvers that provide, to a certain degree, support for strings: bit-vector based solvers like Hampi [45] and Kaluza [3]; automata-based solvers like Violist [46], Stranger [6], [14], ABC [47], StrSolve [15], Pass [16], StringGraph [17], and JST [18]; word-based solvers like Norn [48], S3 [8], and the aforementioned Sushi, CVC4, and Z3-str2. Among them, Stranger, JST, StringGraph, S3, Z3-str2, and CVC4 support the most 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 [7]–[9], [48]), 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, 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.

Code-based security analysis. Code-based security analysis approaches can be broadly categorized into two types: taint analysis and symbolic execution. Taint analysis approaches (such as [31], [32], [35], [49], [50]) check whether application inputs are used in sinks without passing through known sanitization functions. However, these approaches tend to generate many false alarms since they cannot reason about the implementation of sanitization functions. References [51], [52] incorporate string analysis into taint analysis, improving the precision in the analysis of SQLi and XSS vulnerabilities. The SANER tool [53] and the approaches in [54], [55] reason about the adequacy of input sanitization code by combining taint analysis and string constraint solving using finite state automata operations. Symbolic execution approaches [2], [3], [5] 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) of the constraint solvers discussed above.

Search-based test input generation for string data types. There are a few proposals [24], [26], [56] 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 [24], [26] or by hybridizing global search and local search [56]. 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 reason to select Ant Colony Optimization, which resulted in a significantly different search strategy than the ones proposed in the above-mentioned approaches.

VII. Conclusion and Future Work

This work addresses the issue of adding support for (complex) string operations in existing string constraint solvers in the context of vulnerability detection. We have proposed a search-driven constraint solving technique that *complements* the support for complex string operations provided by any existing string constraint solver. This technique uses a hybrid constraint solving procedure based on the Ant Colony Optimization meta-heuristic. The experimental results, based on a benchmark derived from nine realistic Web applications, show that our approach, when combined in a state-of-the-art solver, significantly improves the number of detected vulnerabilities and solves several cases on which the solver fails when used stand-alone, while still keeping the execution time affordable in practice. In the future we plan to integrate our search-driven string constraint solver in a comprehensive framework for vulnerability detection, together with state-of-the-art numeric constraint solvers and automata-based ones.

Acknowledgment

We would like to thank Dr. Xiang Fu for sharing his tool *Sushi*. This work is supported by the National Research Fund, Luxembourg FNR/P10/03, INTER/DFG/14/11092585, and the AFR grant FNR9132112.

REFERENCES

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

[2] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL injection and Cross-Site Scripting attacks," in *Proc. of ICSE'09*. IEEE, 2009, pp. 199–209.

[3] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for JavaScript," in *Proc. of SP'10*. IEEE, 2010, pp. 513–528.

[4] 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.

[5] Y. Zheng and X. Zhang, "Path sensitive static analysis of Web applications for remote code execution vulnerability detection," in *Proc. of ISSRE'13*. IEEE, 2013, pp. 652–661.

[6] 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.

[7] 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 *Proc. of CAV'14*. Springer, 2014, pp. 646–662.

[8] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in Web applications," in *Proc. of CCS'14*. ACM, 2014, pp. 1232–1243.

[9] 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 *Proc. of CAV'15*. Springer, 2015, pp. 235–254.

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

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

[12] M. Dorigo and M. Birattari, *Ant Colony Optimization*. MIT Press, 2010.

[13] J. Thomé, "ACO-Solver: a hybrid constraint solving tool," https://github.com/julianthome/acosolver, 2016.

[14] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *Proc. of TACAS'10*. Springer, 2010, pp. 154–157.

[15] P. Hooimeijer and W. Weimer, "StrSolve: solving string constraints lazily," *Autom. Softw. Eng.*, vol. 19, no. 4, pp. 531–559, 2012.

[16] G. Li and I. Ghosh, "PASS: String solving with parameterized array and interval automaton," in *Proc. of HVC'13*. Springer, 2013, pp. 15–31.

[17] G. Redelinghuys, W. Visser, and J. Geldenhuys, "Symbolic execution of programs with strings," in *Proc. of SAICSIT'12*. ACM, 2012, pp. 139–148.

[18] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang, "JST: An automatic test generation tool for industrial Java applications with strings," in *Proc. of ICSE'13*. IEEE, 2013, pp. 992–1001.

[19] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[20] T. Stützle and H. H. Hoos, "MAX-MIN Ant System," *Future Gener. Comput. Syst.*, vol. 16, no. 9, pp. 889–914, 2000.

[21] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *ACM Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, 2010.

[22] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[23] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[24] 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.

[25] A. Arcuri, "It does matter how you normalise the branch distance in search based software testing," in *Proc. of ICST'10*. IEEE, 2010, pp. 205–214.

[26] P. McMinn, M. Shahbaz, and M. Stevenson, "Search-based test input generation for string data types using the results of Web queries," in *Proc. of ICST'12*. IEEE, 2012, pp. 141–150.

[27] S. Lin, "Computer solutions of the traveling salesman problem," *Alcatel-Lucent Bell Syst. Tech. J.*, vol. 44, no. 10, pp. 2245–2269, 1965.

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

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

[30] Pebble, "A lightweight, open source, Java EE blogging tool," http://pebble.sourceforge.net/, 2015.

[31] V. B. Livshits and M. S. L. Lam, "Finding security vulnerabilities in Java applications with static analysis," in *Proc. of SSYM'05*. USENIX, 2015, pp. 271–286.

[32] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of Web applications," in *Proc. of PLDI'09*. ACM, 2009, pp. 87–97.

[33] Y. Liu and A. Milanova, "Practical static analysis for inference of security-related program properties," in *Proc. of ICPC'09*. IEEE, 2009, pp. 50–59.

[34] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "ASIDE: IDE support for Web application security," in *Proc. of ACSAC'11*. ACM, 2011, pp. 267–276.

[35] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "ANDROMEDA: Accurate and scalable security analysis of Web applications," in *Proc. of FASE'13*. Springer, 2013, pp. 210–225.

[36] A. Møller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, pp. 29:1–29:30, 2014.

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

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

[39] "rest-auth-proxy," https://github.com/kamranzafar/rest-auth-proxy.

[40] N. Antunes and M. Vieira, "Assessing and comparing vulnerability detection tools for Web services: Benchmarking approach and examples," *IEEE Trans. Serv. Comput.*, vol. 8, no. 2, pp. 269–283, 2015.

[41] J. Thomé, L. K. Shar, and L. Briand, "Security slicing for auditing XML, XPath, and SQL injection vulnerabilities," in *Proc. of ISSRE'15*. IEEE, 2015.

[42] NIST, "NIST: National vulnerability database," https://nvd.nist.gov/.

[43] S. Kausler and E. Sherman, "Evaluation of string constraint solvers in the context of symbolic execution," in *Proc. of ASE'14*. ACM, 2014, pp. 259–270.

[44] P. Dinges and G. Agha, "Solving complex path conditions through heuristic search on induced polytopes," in *Proc. of FSE'14*. ACM, 2014, pp. 425–436.

[45] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A solver for string constraints," in *Proc. of ISSTA'09*. ACM, 2009, pp. 105–116.

[46] D. Li, Y. Lyu, M. Wan, and W. G. J. Halfond, "String analysis for Java and Android applications," in *Proc. of ESEC/FSE'15*. ACM, 2015, pp. 661–672.

[47] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Proc. of CAV 2015*. Springer, 2015, pp. 255–272.

[48] 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 *Proc. of CAV'15*. Springer, 2015, pp. 462–469.

[49] W. Halfond, A. Orso, and P. Manolios, "WASP: Protecting web applications using positive tainting and syntax-aware evaluation," *IEEE Trans. Softw. Eng.*, vol. 34, no. 1, pp. 65–81, 2008.

[50] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for Java Web applications," in *Proc. of FASE'14*. Springer, 2014, pp. 140–154.

[51] G. Wassermann and Z. Su, "Sound and precise analysis of Web applications for injection vulnerabilities," in *Proc. of PLDI'07*. ACM, 2007, pp. 32–41.

[52] ——, "Static detection of Cross-Site Scripting vulnerabilities," in *Proc. of ICSE'08*. ACM, 2008, pp. 171–180.

[53] 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 *Proc. of SP'08*. IEEE, 2008, pp. 387–401.

[54] 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 *Proc. of ISSTA'16*. ACM, 2016, pp. 189–200.

[55] F. Yu, M. Alkhalaf, and T. Bultan, "Patching vulnerabilities with sanitization synthesis," in *Proc. of ICSE'11*. ACM, 2011, pp. 251–260.

[56] G. Fraser, A. Arcuri, and P. McMinn, "A memetic algorithm for whole test suite generation," *J. Syst. Software*, vol. 103, pp. 311–327, 2015.