Scalable Hybrid Constraint Solving for Analyzing Common Injection Vulnerabilities: An automaton and search-based approach

Julian Thome, Lwin Khin Shar, and Lionel Briand

# Abstract

Symbolic execution is a state of the art technique for security analysis, in which constraint solving is an essential component. Sound security analysis for real world programs often requires solving of constraints that involve complex string operations, string-numeric interactions, and interactions between multiple variables. Current symbolic execution techniques suffer from scalability issue as they are either unable to handle the above complexities or they could only handle atomic string operations. Further, despite its known usefulness, symbolic execution has not been applied to detect some of the common injection vulnerabilities such as XML, XPath, and LDAP injection. In this paper, we propose a security analysis technique that combines automata-based and search-based constraint solving. The approach is scalable as it handles program operations at an abstract level using automata and incorporates search-based technique to overcome complex constraints. This approach is then used to analyze security vulnerabilities based on sound threat models. Specifically, the approach detects four types of injection vulnerabilities—XML, XPath, SQL, and LDAP injection. We evaluated the approach based on a benchmark set of constraints collected from various vulnerable Web programs. The approach solved xx% of the collected constraints and detected xx% of the vulnerabilities present in the programs with xx% false alarms.

# Introduction

Symbolic execution is one state of the art approach used in security testing to generate test inputs that could reach to various part of the program by solving the constraints imposed on the input variables along the program paths and could observe their vulnerable behaviors. The main strength of symbolic execution approach is that the detected vulnerabilities come at *zero cost* (no false alarm) because it generates actual test inputs that exploit the vulnerabilities.

In our previous work [46], we developed a tool for extracting security slices from Java bytecode. The main idea of this approach was to make security auditing more scalable by helping security auditors to focus on parts of interest (i.e. vulnerable execution paths) in their programs. As our future work, we intended to improve the scalability of symbolic execution in the context of security by performing symbolic execution only on security slices instead of the whole program and focusing on the coverage of security-sensitive operations (*sinks*) instead of every branch in the program.

However, there are two main challenges to achieve the above objective—1) efficient and scalable path exploration strategy to explore the slices and collect the path conditions along; and 2) scalable solving of constraints in the path conditions collected.

Regarding the second challenge, despite the availability of many solvers, they are currently ineffective at solving complex constraints or have scalability issues. Especially in the Web domain, many constraints involve various string functions and string-numeric interactions. Most of the current solvers support atomic string operations and typically handle non-atomic operations by symbolically executing into them. This arises the major challenge of symbolic execution—*scalability*. For example, Figure 1 shows the implementation of Java Trim() function, which contains 2 While-loops and a few predicates. Thus, symbolic execution would be costly if this function is not supported at an abstract level.

Figure 1. Implementation of Java Trim() function

This approach addresses this issue by proposing an approach that solves string constraints at an abstract level and also incorporates search-based technique into constraint solving. Scalability is thus achieved in two-fold, first by abstracting the implementation details of commonly-used string operations using finite-state automata and second by using a search algorithm at the encounter of un-handled constraints.

On the other hand, despite there are a few approaches that apply symbolic execution to detect vulnerabilities, these approaches fail to specifically address some of the common vulnerabilities in Web systems such as XML, XPath, and LDAP injection. An out-of-the-box approach that automatically detects such vulnerabilities would be beneficial.

Hence, our approach focuses on specifically addressing SQL, XML, XPath, and LDAP injection vulnerabilities. That is, we build threat models that reflect possible security attack patterns for these vulnerabilities. Depending on how an attack is possible, we categorize the types of sinks. Each type of sink is associated with a regular language that describes attack patterns (*threat model*). Then, the approach analyzes the satisfiability of a threat model in conjunction with the path condition that leads to a given sink.

The specific contributions of our approach are listed as follows.

* *Targeted security analysis*. An approach that specifically address XML, XPath, SQL, and LDAP injection vulnerabilities. Threat models that reflect attack patterns associated with different types of sinks are provided.
* *Scalability.* An approach that models common string operations at function level and solves their constraints at an abstract level. It incorporates a search technique into constraint solving in order to handle complex constraints that cannot be handled by the automata-based technique.
* *Zero false alarm*. The detected vulnerabilities come at a zero cost because when the approach detects a vulnerability, it needs to be investigated and audited.
* *Practicality*. A prototype tool that fully implements the proposed approach. The tool and the user manual are publicly available at our tool Website [47].
* *Systematic evaluation*. The approach is evaluated based on a benchmark set of constraints collected from real-world vulnerable Web programs. In the experiments, the approach was able to detect xx% of the vulnerabilities.

The rest of the paper is organized as follows. Section 2 provides a background on the common injection vulnerabilities that we address. Section 3 presents our proposed approach. Section 4 provides the Java string and numeric operations supported by our prototype tool. Section 5 evaluates our approach. Section 6 discusses the limitations of our current work. Section 7 compares our approach with related approaches. Section 8 concludes our study.

# Background

***SQL Injection***attacks occur when a web site uses user-supplied information to construct an SQL query to interact with the database.

***XML Injection***is an attack technique used to manipulate or compromise the logic of an XML application or service. The injection of unintended XML content and/or structures into an XML message can alter the intend logic of the application. Further, XML injection can cause the insertion of malicious content into the resulting message/document.

***XPath Injection*** attacks occur when a web site uses user-supplied information to construct an XPath query for XML data in a way similar to SQL injection. By sending intentionally malformed information into the web site, an attacker can find out how the XML data is structured, or access data that he may not normally have access to. He may even be able to elevate his privileges on the web site if the XML data is being used for authentication (such as an XML based user file).

(Note: XML modifies XML document structure whereas XPath modifies a query structure)

***LDAP Injection*** attacks compromise Web sites that construct LDAP (Lightweight Directory Access Protocol) statements from data provided by users. This is done by changing LDAP statements so dynamic Web applications can run with invalid permissions, allowing the attacker to alter, add or delete content. LDAP is a protocol that facilitates the location of organizations, individuals and other resources in a network.

***Threats posed by injection attacks*:** A successful XML, XPath, SQL, or LDAP injection attack poses a very high risk for a website. The attacker can seal the entire database, and can even log in as the administrator of the website. This means that all the sensitive data stored in the database will be accessible to the hacker and he can make any kind of changes he would like to make to the website. This is the biggest threat that the injection poses to the security of a website.

As an example, Figure 2 shows the code vulnerable to SQL injection.

…

Figure 2. Sample code vulnerable to SQL injection

# Approach

As depicted in Figure 3, our proposed approach consists of the following steps.

1. *Attack condition generation*: An attack condition is a condition that could trigger a security attack over a given path condition. This involves conjoining a path condition with its corresponding threat models. Threat models are regular languages that describe attack patterns and are expressed in the form of regular expressions.
2. *Constraint pre-processing*: pre-analysis of constraints contained in the given attack condition for efficient and effective solving. It simplifies them, derives additional constraints to reduce the input domains, identifies non-satisfiable conditions if possible, and also passes the constraints to the solver (automata-based or search-based) suitable for the type of constraints.
3. *Automata-based constraint solving*: many common string and numeric operations are modeled (*interpreted operations*) and solved by automata-based constraint solving techniques.
4. *Search-based constraint solving*: The un-interpreted operations (i.e. operations not modeled by the automaton method) are solved by search-based approach.

Figure 3. Overview of the approach

## Attack Condition Generation

Given a Web program, we can identify paths leading to sinks in the program and collect path conditions using static analysis. We can then reason if a sink is vulnerable using symbolic execution and pre-defined threat models. That is, for a given path, we could conjunct its path condition with a threat model and then check for satisfiability using a constraint solver.

Consider the example vulnerable program in Figure 2, by analyzing the type of sink and the context in which user input is used, attack conditions that could trigger SQL injection attacks could be generated as shown in Figure 4. That is, if a condition is satisfiable, then the sink is vulnerable to security attack.

Identifying the sinks, the types and the contexts and collecting path conditions can be done using existing static analysis techniques [?], and it is done in our previous work [46]. Here, we assume that such information is available to us. However, to generate such attack conditions, it is required for us to pre-define threat models.

Figure 4. Attack conditions generated for the program in Figure 1, which could trigger SQL injection attacks

***Threat models***. We build threat models based on the type of sink and the context in which the user input is used in the sink because different class of scenarios requires different use of meta-characters to conduct attacks. Using the same threat model for every given sink could result in false alarms.

In many of approaches [31][32][29], *regular expressions* are used to describe security vulnerabilities. Likewise, we also use regular expressions to model vulnerabilities.

Table 1 shows our pre-defined threat models. As shown in Table 1, we classify four types of sinks corresponding to the common injection vulnerabilities.

1. SQLi sink
2. XMLi sink
3. XPathi sink
4. LDAPi sink
5. JSONi sink

For each type sink, we classify different types of contexts. For example, the contexts for SQLi sinks are:

1. Input used as ‘value’ in *Where*-clause of SQL query (Select \* From Where uid=’$input’)
2. Input used as ‘attribute’ in SQL query (Select \* From ‘$input’ Where …)

The contexts of XMLi sinks are:

1. Input used as ‘value’ in XML document body (<node>’$input’</node>)
2. Input used as ‘value’ for an XML attribute (<node attr=’$input’/>)

Table . Regular expression patterns that reflect various attack rules

## Pre-processing of Constraints

As shown in Figure 3, the constraint pre-processor consists of three components:

1. Normalizer
2. Constraint network generator
3. Network analyzer

### Normalizer

### Constraint network generator

***Constraint network***. A constraint network is a labeled directed hypergraph H = (X, E, C) where X is the set of string and numeric variables. The set C denotes string and numeric operations. The set E is the set of directed labeled hyperedges. Each labeled hyperedge is a tuple <c, S> where the first component is an operation and the other component S is a set of vertices. In the graph, a vertex corresponds to either a variable in X or a constant/literal value.

Similar to the string graph proposed by Redelinghuys et al. [40], each constraint in the attack condition contributes exactly one hyperedge to the network graph. Predicate operations (e.g. X.matches(Y)) are mapped straightforwardly to hyperedges. For other transformational operations (e.g. X.replace), a *new auxiliary symbolic variable* is introduced to represent the result. This variable is added as a vertex to the graph, and the transformation operation is added as a hyperedge that connects the vertices of the original variable and the new auxiliary variable.

This constraint network also implicitly models the dependencies between different symbolic variables. Maintaining a list of dependencies between symbolic variables is important so that the chain reaction of refinement of a solution automaton can be propagated through the hierarchy according to their operation to ensure that relationships between symbolic variables remain true.

Figure . Constraint network

### Network analyzer

***Refining***. Given a constraint network, we traverse each hyperedge *e* and refine it so that it allows for efficient solving. Specifically, the following lists the refinement operations performed for different string/numeric operations.

* Equality check between two variables (e.g. X.equals(Y), X==Y). The hyperedge *e* and one of the variables (vertex) are removed from the network, connecting the hyperedges that connect to the removed vertex to the other remaining variable (vertex).
* Equality check between one variable and one literal (e.g. X.equals(‘abc’), X==5). Same as above, except here, the variable is removed. The literal remains in the network.
* String numeric interaction (e.g. X.length() > c, X.indexOf(str) = c). Symbolic integer variable ($Ix) is created for each symbolic string variable $X that has an integer constraint. Two common, important operations in Java that take strings as input and return integer are indexOf() and length().
* X.startsWith(Y) 🡪 not sure yet! Constraints performed on X should be constraints performed on symbolic Y (e.g., X.startsWith(Y) & X.contains(“abc”) 🡪 $Y.concat($Z).contains(“abc”)

Solving constraints with string variables is easy when the constraint involves only one string variable and string literals. Hence, by solving for each string variable and then later by refining solution automaton to ensure that relationships between symbolic variables remain true (which may also cause adding additional constraints on the path condition that account for dependencies), it allows for accurate and efficient automata-based solving.

***Additional constraint generation***. For both numeric and string operations, additional constraints may be generated to reduce the search space or efficiency. Table 2 shows some of the derived constraints corresponding to the Java expressions. The \* denotes the new constraints proposed by us. The rest are from the two related works [38][39]. The full list of derived constraints is provided in our Website [47].

Table . String operations and derived integer constraints

**For search-based solving**, the choice of the symbolic variable to start the search process has a huge impact on the efficiency of our approach, because by picking a value for a variable, we can narrow down the search-spaces for other variables that are related to the first one. Hence, before solving a given constraint, a pre-processing is required to determine with which variable to the search process starts. When picking the first variable the following criteria are important:

1. #Constraints: The higher the amount of constraint in which a variable is involved in, the higher the probability that this choice impacts the search-spaces of the other variables.

2. Domain size: If a variable has a small domain, there are less options to check with respect to the other variables with whom this variable is associated with.

## Constraint Solving

Table 3 shows the regular language recipes that are used by automaton solving approach.

Table . Regular language recipes corresponding to Java expressions

***Automaton-based solving***: Our constraint solving technique is based on an automata-based string constraint solver called *Sushi*. It supports regular expression operations in java.regex library. It handles regular expression-based string equality check, concatenation, replacement, substring. Many other solvers are not able to support regular expression operations, which is essential in the context of security because input validation and sanitization functions are often implemented using such operations.

It can handle unbounded string lengths, but suffers from undecidability problem. It addresses this problem by imposing restrictions on the number of occurrence of variables. It also cannot handle Boolean and numeric operations as it is mainly intended to support string constraints.

*Sushi*’s regular expression operations are tested for compatibility with the behaviors of Java’s regular expression library *java.util.regex*, which is important as our project is mainly designed to support Java.

Approach. Once the constraint network is constructed, the approach explores the hyperedges in the network.

Redelinghuys et al.’s approach explores the edges in string graph (hypergraph) in a completely arbitrary order. There is no ordering, sorting or forcing on the sequence of edges. Alternatively, our approach can use some simple heuristic to determine the sequence in which edges are to be explored. For example, we can start with the edges involving attack.

Heuristics for exploring the graph:

Start from the edge that involves attack

Prioritize the edge that involves more constants (i.e. edges that contain less variables)

It translates each hyperedge and an automaton Mi is constructed for each variable si represented at vertex *v*.

As the hyperedges are processed, the automata are modified to reflect the effect of the constraint. The processing algorithm is a typical work list pattern. Initially W contains all hyperedges. The positive hyperedges are removed one-by-one and processed. Any change to Mi causes those hyperedges connected to si to be placed in the work list again. Eventually though, the algorithm must terminate because the language of each automaton either stays the same or is restricted during each assignment (to check this!). When a fixed point is reached, each automaton contains exactly the solution set that would satisfy the constraints.

If during any assignment an automaton is reduced to the empty language, the corresponding constraints, hence the network, are known to be unsatisfiable or inconsistent. This causes the procedure to terminate.

If new constraints are found to be necessary to be added during the course of exploring edges and solving, stop the while-loop, add the new constraints to the graph, and repeat the loop. If the loop terminates with no addition to the constraints, we conclude that the constraint satisfaction problem is satisfiable or not, depending on the output of the solver.

## Search-based Solving:

* Search method: Hill climbing based AVM
* Our search space for each input variable: FSA
* Search operator (to modify/adjust input value): DFS on FSA transitions on improvement of fitness value. Backtrack on non-improvement. Switch to another entry transition when a max DFS is reached and no solution found.
* Fitness functions:
* with variable involved: matches on FSA
* only concrete values:
  + Numeric: Korel’s numerical fitness functions,
  + String: Levenshtein distance

***Metaheuristics search*** techniques are high-level frameworks which utilize heuristics in order to find solutions to combinatorial problems at a reasonable computational cost. This technique is not problem-specific. But the heuristics and thus, objective (fitness) functions are problem-specific.

The advantage of using automata-based solving as a first step is that queries can be made about the state of variables, up to the state where constraints can be solved by automata approach, to help guide the search in solving the un-handled operation. For example, given the constraint:

S1.indexOf(“ab”) = S2.unknown();

For example, given the constraint, we can first solve S1.indexOf and query the resulting automaton for s1 to determine if it contains the language .\*ab.\* or the index of ‘ab’ = -1 or not.

In our security analysis context, the main objective is to solve the attack conditions. Hence, the constraints that require search-based solving are of the form: , where is an unknown function (possibly a set of string and numeric functions) applied on variables , and is an attack pattern. An example of such constraints involving one and two variables is shown in Figure 5.

Figure 6. Constraint that requires search-based solving

*Algorithm*.The set of possible values for each variables up before is represented with the automaton as shown in in Figure 5. Initially, concrete values for the variables are randomly generated from these automata. And is run on these concrete inputs. From the concrete output of , the fitness of current input values of the variables is computed using a fitness function.

*Fitness function*. Depending on the types of predicates (Boolean, numeric, string), we use different fitness functions, which are listed in Table 4.

{Given , we first check if the attack condition is sat (i.e. ). It the condition is sat, the fitness is zero.

If unsat, the fitness is computed using the following criteria:

1. Number of characters in that match
2. The length of the longest sequence of characters in that match
3. Best Levenshtein distance [42] between and a string generated from . ‘Best’ distance value is achieved as follows. From , enumerate the set of strings with length equal to . Compute Levenshtein distance between and each string in until a time out is reached or all the strings are exhausted (choosing first the strings that satisfy the above two criteria). The smallest distance value among them is the best value.}

Table . Fitness functions

***AVM method***: Based on local search, hill climbing algorithm using Korel’s alternating variable method (AVM) [41], each variable is modified/adjusted/mutated in turn while all other variables remain fixed. When the selected mutation improves fitness, the algorithm continues to mutate the same variable. But, to avoid ‘over shoot’, when fitness is close to zero, it switches to other variables.

AVM method is a local search for deriving input values in accordance with the fitness function. Each input variable is taken in turn and its value adjusted, keeping the other variable values constant. The first stage of manipulating an input variable is called the exploratory phase. This probes the neighborhood of the variable by increasing and decreasing its original value. If either move leads to an improved fitness value, a pattern phase is entered. In this phase, a larger move is made in the direction of the improvement. A series of similar moves is made until a minimum for the fitness function is found for the variable. The next input variable is then selected for an exploratory phase.

***Direction.*** The fitness value of the current inputs is compared against the fitness value of the previous inputs so as to determine which direction to proceed, that is, to make a larger move, a smaller move, or a change in direction (select different FSA transition). If the changes on one input do not make any improvement on the fitness value, switch to another input variable.

AVM is essentially an adaptation of Hill Climbing [44] to numeric problems (ours is both numeric and string). It starts the search with *one* vector of assignments to input variables and alternates across different variables during the search. In each step it makes small positive and negative increments to the values associated to the selected variable and re-evaluates fitness to decide whether to go up or down hill. As the mutation is fine-grained, AVM often incorporates random-restarts to escape local maxima.

We use three basic string mutation operators following Alshraideh and Bottaci [43]: deletion , insertion , and substitution . -operator deletes a random character from a given string. -operator inserts an English-like character into a given string. -operator randomly selects a character from a given string and replaces with another random character.

We need to handle three types of scenarios regarding unknown functions:

1. ***String predicate:*** Unknown transformation function that returns string and perform string comparison with another string. In this case, we define ***fitness function*** based on how much the output string from unknown function matches the regular expression of another string; or the Levenshtein distance between the two concrete strings (if another string is concrete)
2. ***Numerical predicate:*** Unknown transformation function that returns numeric and perform numerical relation (=, !=, < , >). Fitness function is defined based on how much the output numeric value from unknown function matches the regular expression of another numeric variable (e.g. I1 < V2) or the normal numerical fitness functions (e.g. I1 – I2 < 0) if the value of another variable is concrete. For numerical predicates with concrete values, we use standard, Korel’s fitness functions [41] which are shown in Table 5:
3. ***Special numeric predicate:*** Unknown function that returns Boolean (which can be considered as a special case of numeric predicate). An unknown function returns a boolean value which is checked against either 1 or 0. In this case, we use some simple heuristics as a fitness function. Heuristics can be randomly pick an entry transition (of FSA) and do a DFS until a certain depth is reached. If reached, then pick another entry different transition randomly and do a DFS again.

Fitness functions for matching regular expression may differ in terms of the particular costs attached to the particular operations. For example, it may be more important to match digits than letters or leftmost character matches may be more important than other matches in other positions. In our security context, it should be utmost important to match special characters associated with a particular sink.

***Fitness function for regular expression matching*** is the approximate regular expression matching problem [45] which is to find a sequence r matching R whose optimal alignment with A is the highest scoring of all such sequences (r is closet to A), given a sequence A and a regular expression R. Basically, this is a *cost function* used to compute *the distance* between an input string and a regular expression.

Table . Korel’s fitness functions for numerical predicates

# Supported String and Numeric Operations

This section provides the string operations supported by our approach. According to G. Redelinghuys’s survey [39] on 38 Java projects, most commonly used string operations are concatenation (70%), equals (10%), length (4%), substring, indexOf, and charAt, According to his survey, the following operations should be supported if the realistic Java programs are to be run on the constraint solver.

|  |  |  |  |
| --- | --- | --- | --- |
| capacity | endsWith | parseFloat | startsWith |
| charAt | equalsIgnoreCase | parseInt | subSequence |
| compareTo | Intern | regionMatches | toCharArray |
| concat | isEmpty | replace | toLowerCase |
| contains | lastIndexOf | setCharAt | toUpperCase |
| contentEquals | matches | setLength | Trim |
| copyValueOf | parseDouble | split | valueOf |

These operations are all supported by our approach. In addition, we support many other operations that are essential in our security analysis context.

We handle a set of standard Java library classes ( http://docs.oracle.com/javase/7/docs/api/java/lang/package-summary.html)—java.lang.String class (15 constructors and 65 string operations/methods), java.lang.Boolean (2 constructors and 11 methods), java.lang.Character. Furthermore, we also handle input sanitization operations provided by Apache Commons Lang 3 (org.apache.commons.lang3.StringEscapeUtils), OWASP, and Sprint framework. These security libraries provide sanitization methods for escaping SQL, XPath, HTML entities, Json, and XML entities. In security analysis, it is important to be able to reason with such sanitization operations (even though they are provided by security experts and expected to be correct) because developers, who are often not skilled in security, might incorrectly use these operations.

# Evaluation

Like JST [38], we could not compare our results with any other freely available tool because we found all of them to be completely inadequate in handing all the String operations that existed in our examples and the specific interactions between the numeric and string domain. There is also currently no standard format for expressing String constraints. Thus it is not possible to translate all the complex constraints to operations that other String-Numeric constraint solving tools can understand.

We will compare with CVC4 [37] and Z3-str [36] if they can handle the Java expressions as it is without the need for us to translate them into formats they can understand. It is not trivial to translate all the complex Java operations to formats that these tools can understand. And we are not evaluating our solver in terms of solving capability in general. We are evaluating its effectiveness at detecting vulnerabilities automatically.

# Limitation and Discussion

Our approach of detecting vulnerabilities is *sound* with respect to the threat models that we define above. The models reflect the known attack patterns for the four types of vulnerabilities we address. Such an approach is considered to be *blacklist-based* approach, which tends to miss vulnerabilities due to unknown or new type of attack patterns. However, from the interfaces provided by our tool, it is not difficult to define additional threat models that reflect new attack patterns.

The more important point is that it is straightforward to convert our approach to *whitelist-based*. It can be done by using valid input conditions instead of attack conditions. However, this requires user to define valid input conditions (it could also generate many false alarms). Our approach is designed to work out-of-the-box. As shown in our experiments, it automatically detects vulnerabilities with almost zero false warning. Thus, at least, the detected vulnerabilities come with zero cost.

In addition, our approach currently does not handle the following:

* Non-linear integer constraint on string variable length
* Context-free grammar
* Bounded vs unbounded string length: handling unbounded string length could lead to undecidability problem. This is addressed by imposing the restriction that every string variable occurs at most once in the constraint.

# Related Work

***Solvers used for security analysis***. ***Fu et al. [29]*** provide a *transducer* model for Perl-style regex replacement operations. This type of operation is difficult to model because the semantics are subtle across, for example, eager vs. non-eager replacement. It would be interesting to combine *transducer-based analyses* with a string constraint solver.

Minamide [30] uses context-free gram- mars and finite state transducers to perform basic XHTML validity and cross-site scripting checks. Wassermann and Su [31] build on Minamide’s analysis to detect SQL injection vulnerabilities and cross-site scripting vulnerabilities [32], by combining it with conservative static taint analysis.

Weimer present an automata-based solver, DPRLE, for matching problems of the form e ⊆ r where, in essence, r is a regular expression over a given alphabet and e is a concatenation of alphabet symbols and string variables. The solver has been used to check programs against SQL injection vulnerabilities.

JSA [??] uses static analysis to build a flow graph of a Java program. Then a “special” context-free grammar is defined from the model and the Mohri- Nederhof algorithm [21] is applied to obtain an approximate regular expression which expresses the set of inputs which satisfies the majority of the Java program’s string constraints.

With the resulting regular expression that JSA provides, one can verify if it contains the subset of any known security vulnerabilities, such as SQL injections.

Importantly, it seems as if this approach can only handle a single symbolic string variable and cannot deal with symbolic integer inputs. Of course, this restriction severely limits the usefulness of this technique.

**Z3-str [36]** supports only a few string operations: string equation, concatenation, length, substring, contains, indexof, replace and split. Other string operations are reduced to an equivalent formula based on three primitive string operations: string equation, concatenation and string length. According to Z3-str’s paper, string-numeric operation concatenation and length operations are supported by performing incremental reduction and adding new axioms gradually, driven by the try-and-backtrack process. And other operations are supported in a different way, meaning that different operations require different treatment. Therefore, for those operations that are not yet supported, it would be nontrivial for users to add the support.

***Generic solvers.*** STRSolver [34] proposes a search-based solving approach, a constraint-solving algorithm for equations over string variables. Their algorithm has similar features to existing string decision procedures, but is designed to yield faster answers to yes-instances for large input constraint systems. We achieve this by treating the constraint solving problem as ***an explicit search problem***. A key feature of our algorithm is that we instantiate the search space in an on-demand fashion. But selecting a few solutions with a guarantee that they are correct is tricky, also backtracking unnecessarily is difficult to avoid. They do not support solving string-numeric interaction operations.

***CVC4 [37] and Z3-str [36]*** are SMT solvers based on DPLL(T) theory of strings. CVC4 was evaluated against Z3-str and Kaluza. Their solving theories are incomplete (may not recognize an unsatisfiable problem, i.e. false positive) and non-terminating in general. They could handle string and numeric constraints together. However, they do not solve Java operations at an abstract level. CVC4 [37] also does not support many important string constraints such as contains (according to the discussion from their conclusion section).

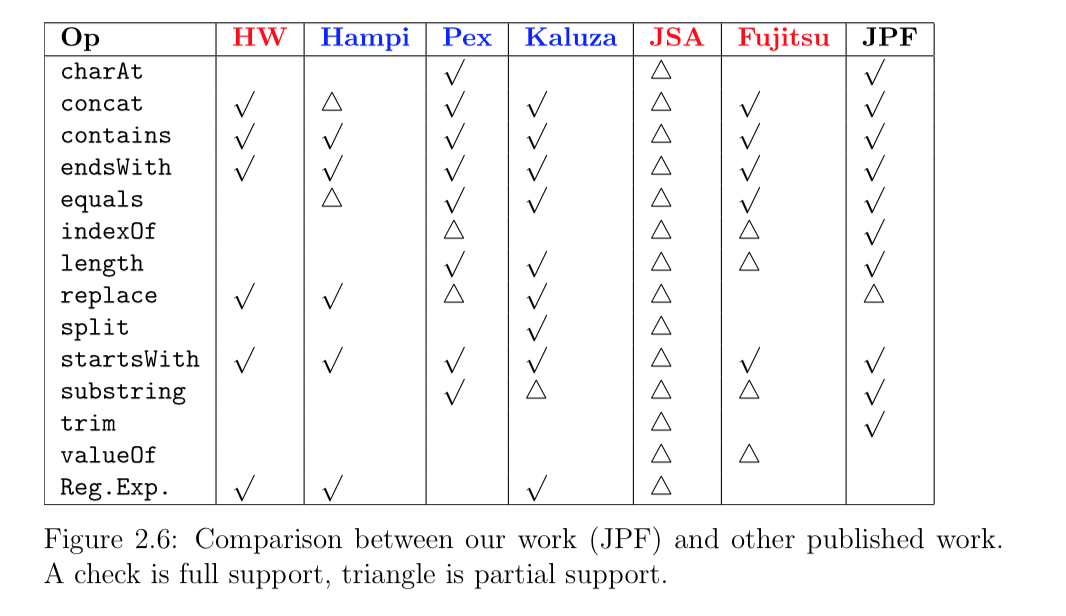


Figure 7. Java operations supported by existing solvers (red-automata, blue-bit vector) (detail comparison of these approaches can be found in Redelinghuys’s thesis [39])

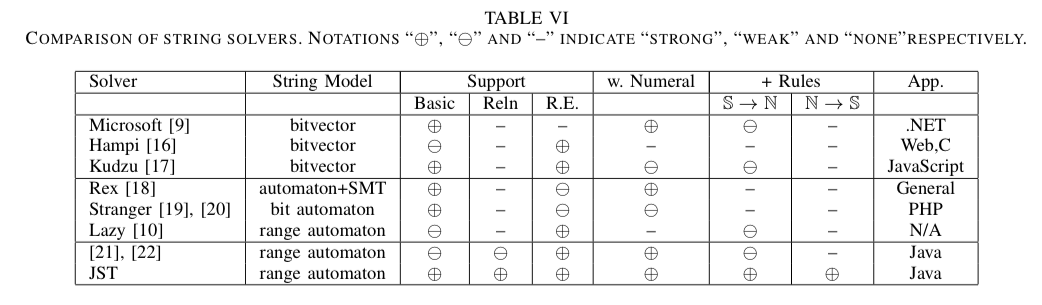


Figure 8. Comparison by JST *[38]*

***JST [38]*** is based on JPF and is an industrial-strength solver that can handle both string and numeric operations. They focus on handling non-linear numeric equations in a scalable way. It has support for string operations and string-numeric interactions from java.lang.string, but not other string libraries like from Apache (but JST handles string-numeric interactions in a iterative way similar to Redelinghuys [39][40], this have the same limitation). It also has (limited) support for Java classes like Maps, Arrays and data structures like BigDecimal and BigInteger. It is not clear how JST handles *negative constraints* (e.g. !equal). They also propose *relaxed solving* technique that checks for satisfiability at the end of a symbolic path (last branching condition), solving conditions at intermediate branches without multiple interatactions.

***Comparison with bitvector approach***. Bitvector-based solving requires a fixed length for each string variable. Since these lengths are not known a priori, the solving process typically has to be repeated for all possible lengths, up to a preset bound.

Ref: Anand et al., symbolic execution with abstract subsumption checking, 2006

Symbolic execution allows one to analyze programs with un-initialized inputs. The main idea is to use symbolic values, instead of actual (concrete) data, as input values and to represent the values of program variables as symbolic expressions. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs.

Ref: Automated whitebox fuzz testing, abstracting symbolic execution with string analysis

Fuzz testing is a form of black-box random testing which randomly mutates well-formed inputs to generate new test inputs. In some cases, grammars are used to generate the well-formed inputs, which also allows encoding application-specific knowledge and test heuristics.

In theory, symbolic execution can lead to full program path coverage. In practice, however, the search is typically incomplete because 1) the number of execution paths in the program is huge (path explosion) and 2) symbolic execution, constraint generation, and constraint solving might be imprecise (imperfect symbolic execution).

Symbolic execution of large programs is bound to be imprecise due to complex program statements (pointer manipulations, arithmetic operations, etc.) and calls to operating system and library functions that are hard or impossible to reason about symbolically with good enough precision at a reasonable cost.

The scalability of symbolic execution can be mitigated if the analysis is only required to check the main code, that is, trusting the correctness of the implementation of selected libraries.

Early implementations of symbolic execution mostly dealt with primitive types, such as integers and arithmetic operations. But recent development has generalized symbolic execution to handle references and array types.

By abstracting out the implementation details of common library classes, we can reduce the complexity of a program and enhance the scalability of symbolic execution. For example, a program that uses a library implementation of the abstract data type Set. Suppose the method adds an element e to a set S, and then checks if e belongs to S. Executing the program using representation level manipulations involves executing methods that implement the add and membership check operations.

We handle a set of standard Java library classes (Java 7, http://docs.oracle.com/javase/7/docs/api/java/lang/package-summary.html)—java.lang.String class (15 constructors and 65 string operations/methods), java.lang.Boolean (2 constructors and 11 methods), java.lang.Character. We could also handle commonly-used methods from other library classes such as lists (java.util.ArrayList, java.util.Vector, java.util.LinkedList), trees, or maps. We also handle string operations provided by Apache Commons Lang 3 (http://commons.apache.org/proper/commons-lang/) such as (org.apache.commons.lang3.StringUtils, org.apache.commons.lang3.StringEscapeUtils). String escape utility library from Apache provide sanitization methods for HTML entities, Java, Json, XML entities.

Ref: Generalized symbolic execution for model checking and testing, TACAS 2003.

Handling Input Data Structures SPF uses lazy initialization [5] to handle unbounded input data structures. The execution starts on data structures with un-initialized fields and it initializes them lazily, when the fields are first accessed. A field of class T is initialized non-deterministically to (1) null, (2) a reference to a new instance of class T with uninitialized fields, or (3) a reference to an object of type T created during a prior field initialization; this systematically treats aliasing. The *HeapChoiceGenerator* is used to generate the choices. We have recently extended SPF to provide support for polymorphism. Step (2) above is replaced with non-deterministically assigning new instances of class T and of all the classes that inherit from T. Similarly, step (3) is replaced with assigning previously created objects to class T and objects from classes that inherit from T.

Possible infinite branches (loops or recursion) are bounded by JPF-symbc (by bounding JPF-core’s search depth).

JPF executes on function level, not on the whole program, and user needs to define which function parameters or which variables in the function are to be treated as symbolic.

*Ardilla* is based on input generation, taint propagation, and input mutation to find variants of an execution that exploit a vulnerability.

*Vulnerability scanners* rely on either testing or static taint analysis. Automated testing is ill-suited to finding errors in input validation code, because even flawed validation code catches most malicious uses, and exploits must be crafted specifically for a certain validation’s weakness in order to work.

*Taint analysis* takes as input a list of functions designated as sanitizers, but it does not perform any analysis on them, so it will not catch errors caused by weak input validation.

# Conclusion

*Scalability* is paramount importance in constraint solving. Our approach is a step in that direction.

# Reference

1. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proceedings of TACAS (2003)
2. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI (2005)
3. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: Proceedings of ESEC/FSE (2005)
4. C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. Int J Softw Tools Technol Transfer (2009), 11:339-353.
5. Pasareanu, C.S., Visser, W.: Verification of java programs using symbolic execution and invariant generation. In: Proceedings of SPIN (2004)
6. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Proceedings of TAIC-PART (2007)
7. Tiwari, A., Rues, H., Saidi, H., Shankar, N.: A technique for invariant generation. In: Proceedings of TACAS (2001)
8. Hantler S.L., King, J.C.: An introduction to proving the correctness of programs. ACM Comput. Surv. **8**(3), 331–353 (1976)
9. Cousot, P.: The role of abstract interpretation in formal methods.  In: Proceedings of SEFM (2007)
10. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic execution with abstract subsumption checking. In: Proceedings of SPIN (2006)
11. Godefroid, P.: Compositional dynamic test generation. In: Proceedings of POPL (2007)
12. Anand,S.,Godefroid,P.,Tillmann,N.:Demand-drivencomposi- tional symbolic execution. In: Proceedings of TACAS (2008)
13. Babic,D.:ExploitingStructureforScalableSoftwareVerification. Ph.D. thesis, University of British Columbia, Vancouver, Canada, Aug (2008)
14. Arons, T., Elster E., Ozer S., Shalev J., Singerman, E.: Efficient symbolic simulation of low level software. In: Proceedings of DATE (2008)
15. Xiang Fu, Michael C. Powell, Michael Bantegui, Chung-Chih Li: Simple linear string constraints. Formal Asp. Comput. 25(6): 847-891 (2013)
16. Java PathFinder: <http://javapathfinder.sourceforge.net>
17. O. Tkachuk, M. B. Dwyer, and C. Pasareanu, “Automated environment generation for software  model checking,” in Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Oct. 2003.
18. S. Rajan, O. Tkachuk, M. Prasad, I. Ghosh, N. Goel, and T. Uehara, “Weave: Web application validation environment,” in Proceeding of the International Conference on Software Engineering, 2009.
19. Appelt D., Alshahwan, N., Nguyen, D. C., Briand, L., “Black-box SQL injection testing,” Technical report, Interdisciplinary Centre for Security, Reliability, and Trust, University of Luxembourg, Jan 2014. <http://hdl.handle.net/10993/15121>
20. Harman, M. “The current state and future of search based software engineering,” in Future of Software Engineering (FOSE’07), pp. 342-357, 2007.
21. Spring. <http://spring.io>
22. Java PathFinder: <http://javapathfinder.sourceforge.net>
23. Kieżun, A., Guo, P. J., Jayaraman, K., and Ernst, M. D. “Automatic creation of SQL injection and cross-site scripting attacks,” in Proceedings of the 31st International Conference on Software Engineering, pp. 199-209, May 2009.
24. Shar, L. K., Tan, H. B. K, and Briand, L. “Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis,” in Proceedings of the 35th International Conference on Software Engineering, pp. 642-651, May 2013.
25. Spring vulnerability database, <http://support.springsource.com/security/springsource-all>
26. OWASP Top 10 vulnerabilities, 2013, <https://www.owasp.org/index.php/Top_10_2013-Top_10>
27. 96% of applications have an average of 14 (1/4 are XSS) vulnerabilities, 2014 <http://www.infosecurity-magazine.com/view/37037/96-of-applications-have-an-average-of-14-vulnerabilities/>
28. M. Borges et al., “Symbolic execution with interval solving and meta-heuristic search,”
29. Fu, X., Powell, M., Bantegui, M., Li, C.C.: Simple linear string constraints. Form. Asp. Comput. 1–45 (2012). doi:10.1007/s00165-011-0214-3
30. Minamide, Y.: Static approximation of dynamically generated web pages. In: International Conference on the World Wide Web, pp. 432–441 (2005). http://doi.acm.org/10.1145/1060745.1060809
31. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Programming Languages Design and Implementation, pp. 32–41 (2007)
32. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: International Conference on Software Engineering (2008)
33. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: International Symposium on Software Testing and Analysis, pp. 105–116 (2009)
34. P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 377–386. ACM, 2010.
35. P.Hooimeijer and W.Weimer. A decision procedure for subset constraint solver regular languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198. ACM, 2009
36. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.
37. T. Liang et al. A DPLL (T) theory solver for a theory of strings and regular expressions, CAD 2014.
38. I. Ghosh, N. Shafiei, G. Li, and W. Chiang. JST: An automatic test generation tool for  industrial Java applications with strings. In *Proceedings of the 2013 International Con- ference on Software Engineering*, ICSE ’13, pages 992–1001, Piscataway, NJ, USA, 2013. IEEE Press.
39. G. Redelinghuys, “Symbolic string execution,” Master’s thesis, University of Stellenbosch, 2012.
40. G. Redelinghuys, W. Visser, and J. Geldenhuys, “Symbolic execution of programs with strings,” in *SAICSIT Conf.*, 2012.
41. B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.
42. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.
43. Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203,  2006.
44. D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.
45. Myers EW, Miller W. Approximate matching of regular expressions. *Bulletin of Mathematical Biology* 1989; **51**(1):7–37.
46. Thome, J., Shar L.K., and Briand L., Security slicing.
47. J. Thome, “ACo-Solver: a hybrid constraint solving tool,” http://wwwen.uni.lu/snt/ research/software verification and validation lab/tools from svv lab, 2015.