Symbolic execution of programs with strings

Article · October 2012

DOI: 10.1145/2389836.2389853

CITATIONS
15

READS
56

3 authors, including:

Willem Visser
Stellenbosch University
156 PUBLICATIONS **7,173** CITATIONS

SEE PROFILE

Jaco Geldenhuys
Stellenbosch University
43 PUBLICATIONS **490** CITATIONS

SEE PROFILE

Symbolic execution of programs with strings

Gideon Redelinghuys
VASTech
Octoplace Block C
Electron Str, Technopark
Stellenbosch, South Africa

Willem Visser
Computer Science Division
Dept. Mathematical Sciences
Stellenbosch University
Stellenbosch, South Africa
wvisser@cs.sun.ac.za

Jaco Geldenhuys
Computer Science Division
Dept. Mathematical Sciences
Stellenbosch University
Stellenbosch, South Africa
jaco@cs.sun.ac.za

ABSTRACT

Symbolic execution has long been a popular technique for automated test generation and for error detection in complex code. Most of the focus has however been on programs manipulating integers, booleans, and references in object oriented programs. Recently researchers have started looking at programs that do lots of string processing; this is motivated by the popularity of the web and the risk that errors in such programs may lead to security violations. Attempts to extend symbolic execution to the domain of strings have mainly been divided into one of two camps: automata-based approaches and approaches based on efficient bitvector analysis. Here we investigate these two approaches in one setting: the symbolic execution framework of Java PathFinder. First we describe the implementations of both approaches and then do an extensive evaluation to show under what circumstances each approach performs well (or not so well). We also illustrate the usefulness of the symbolic execution of strings by finding errors in real-world examples.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Test case generation, Verification

Keywords

Symbolic Execution, Strings, Automata, SMT Solvers

1. INTRODUCTION

The main problem with manual testing is the time spent producing inputs to the program that will cover interesting behavior. It is no wonder that many techniques have come to the fore to automate this process. Although random generation of inputs is remarkably effective, for some types of programs this doesn't work. Programs that manipulate strings are notoriously difficult to test in this way. The problem here is that often only a very small subset of input strings trigger interesting behavior. A different approach is symbolic execution [11], which is a white-box approach and therefore looks at the structure of the code to determine which inputs should be produced. The idea is that a program is executed with symbolic inputs and at each branch the branching conditions are added to a path condition that contains the constraints to reach that point in the code. Additionally, the path condition is checked for satisfiability to determine if the branch is feasible. The focus of this paper is to specifically consider the analysis of string-manipulating Java programs with the use of symbolic execution. Section 2 gives a brief overview of symbolic execution.

Why is testing of string manipulating programs important? With the advent of the web, and maybe more specifically the interactive web where users constantly interact with services and/or other users via social networks (or simply email), text inputs have become commonplace. These inputs however can come not just from *nice* users, but also from those with malicious intent. Take for example the real-world case (described in Section 5.1) that sanitizes its input by stripping some characters from a string to ensure that no malicious actions can result. The problem is that the input "`<< HREF=""<A HREF=">`" causes the code to go into an infinite loop. This error crashed one server after another as a user issued the same service request to the system, over and over. In this case, the fact that potentially malicious input was sanitized actually caused an error (even though the input was not malicious).

Why is symbolic execution of string manipulations hard? The main reason is that string operations mix two domains, namely strings and integers (for example, operations that uses the string length or characters at specific indexes). In fact many of the current solutions to symbolic execution for strings, either do not consider all integer interactions [14] or none at all [7, 8, 10, 9]. We take an iterative approach where we first solve the integer constraints and then use the results to determine *candidate* string lengths, which are subsequently used to see if the complete constraints are satisfiable for those lengths. If so, we are done and symbolic execution can proceed to the next instruction. If no solution can be found given the candidate length bounds, we use this fact to determine new candidate results and repeat the process (Section 4.4).

How are we doing the symbolic analysis? Existing approaches to symbolic execution of string code can be divided into two groups: automata-based [3, 7, 8, 15, 16] and bitvector-based [1, 9, 10, 14, 17, 18]. One of our main con-

SAICSIT'12 October 01 - 03 2012, Pretoria, South Africa

tributions in this paper is to compare these two approaches within one setting. The setting we chose is that of symbolic execution of Java programs, and specifically using the symbolic execution extension of the Java PathFinder (JPF) model checker [12]. This extension — called Symbolic PathFinder (or SPF) — supports symbolic analysis of many domains, including integers, booleans, floats as well as object references. There is also a proprietary implementation for string analysis used by Fujitsu based on the automata approach [15, 16]. SPF supports a wide variety of decision procedures to handle the non-string domains, and our solution for strings is engineered such that it can be used in combination with any of these, with one important caveat: as stated earlier we solve integer constraints to enable us to determine string lengths, and therefore we can only use the decision procedures that have the capability to provide satisfying solutions, i.e. solve constraints. For this reason we refer to the decision procedures for the integer domain as constraint solvers in the rest of the paper. In this paper we will use the automata package of the Java String Analyzer (JSA) [3] for our automata approach and the Z3 SMT solver [5] for bitvectors. In both cases these solutions are also used in other string symbolic execution engines: Fujitsu and JSA itself use the automata package from JSA, and PEX [17] uses Z3.

How much can we solve without the sophisticated tools? For every string constraint that the tool encounters during the analysis we first build a constraint graph, called a *string graph*. Using some straightforward heuristics we simplify the graph and detect any inconsistencies that would immediately show the constraint to be unsatisfiable. The string graph can be seen as an intermediate notation (language), since after simplification it is translated into the back-end format required by either the automata or bitvector approach (see the second and third columns of Table 2 for the respective translation rules). Besides finding unsatisfiable constraints it is during the string graph analysis that new integer constraints are derived for the lengths of each string variable in the graph (see Figure 7).

What did we find? From an implementation point of view there is a considerable difference between the two approaches: one needs to build a string decision procedure on top of the automata package, whereas the SMT solver has many of the required functionality already built in. (After translation of the string graph into bitvectors, it is essentially a push-button operation.) In order to evaluate the relative performance we did a number of experiments and also ran it on some real-world examples. Our technique found the error mentioned above in a few minutes and detected the error described in Section 3 (that formed the basis of an actual security attack) in a few seconds. Interestingly we found that, on the whole, automata- and bitvector-based backends perform quite similar, and that the important part of the system is how one handles the interaction between string and integer constraints.

The contributions of this work can be summarized as follows:

- a first comparison for automata- and bitvector-based back-ends for string symbolic execution;
- open-source contribution as part of the jpf-symbc extension[1] of Java PathFinder, where all the examples in this paper can also be found;
- new approach to preprocessing the constraints in a string graph to optimize the analysis;
- detailed description of how mixed integer and string constraints are handled; all `java.lang.String` operations are handled with all parameters allowed to be symbolic (with the exception of `replace` where the target string and parameters must be concrete);
- evaluation on both artificial examples (to determine the strengths and weaknesses of each approach) and real-world programs to show the effectiveness of the tool.

[1] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`

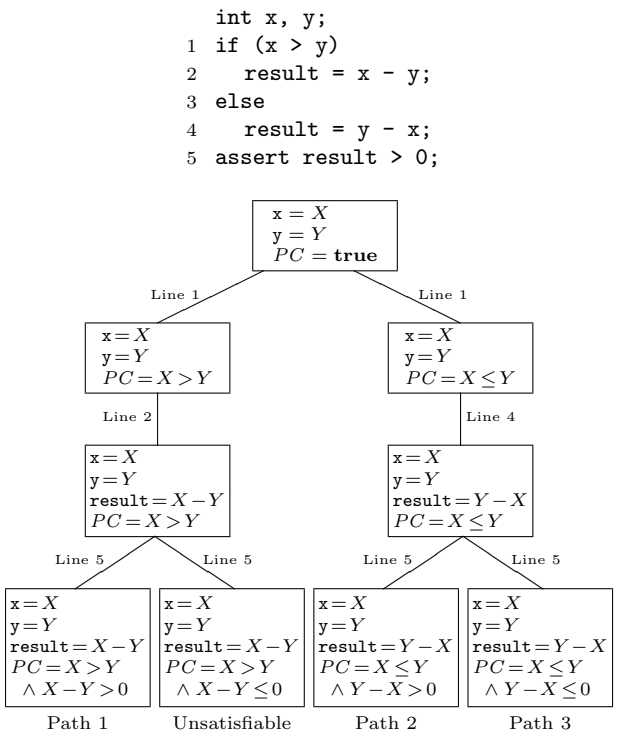

Figure 1: Example program and its execution tree

2. SYMBOLIC EXECUTION

Symbolic execution [4, 11] is a program analysis technique. Instead of concrete values, the program inputs are treated as symbolic, and expressions are expressed as functions of the symbolic inputs. At any point during its execution, the state of a sequential, deterministic program is fully described by the program counter and a propositional symbolic expression known as the *path condition* (PC). It encodes the constraints that the inputs must satisfy in order for a program to reach its current location. (In the case of concurrent and/or nondeterministic programs a little more information is required, but this is beyond the scope of this paper.) For the sake of convenience, we include the symbolic values of program variables as a part of the state. The set of all possible executions of a program is represented by a *symbolic execution tree*.

As an illustrative example (taken from [12]), consider the code at the top of Figure 1 that computes the absolute difference between two input integers `x` and `y`. For the concrete values $\texttt{x} = 2$ and $\texttt{y} = 1$, only one path is followed through the code: line 1, line 2, and line 5. This execution follows the **true** branch of the `if` statement at line 1, because $2 > 1$, and the assertion in line 5 is not violated.

Symbolic execution starts with symbolic, rather than concrete, input values, $\texttt{x} = X$, $\texttt{y} = Y$, and sets the initial value of PC to **true**. The corresponding (simplified) execution tree is shown in the lower part of Figure 1. At each branch point, PC is updated with constraints on the inputs in order to choose between alternative paths. For example, after executing line 1, both alternatives of the `if` statement are viable, and PC is updated accordingly. If the path condition becomes **false**, it means that the corresponding path is infeasible (and symbolic execution does not continue for that path). The execution tree is not always built explicitly, but rather explored according to some strategy (depth-first, breadth-first, or according to some order based, for example, on heuristics).

In the example, symbolic execution explores three different feasible paths, with the following path conditions (see Figure 1 (right)):

- $PC_1 : X > Y \wedge X - Y > 0$ for Path 1
- $PC_2 : X \leq Y \wedge Y - X > 0$ for Path 2
- $PC_3 : X \leq Y \wedge Y - X \leq 0$ for Path 3

The last path condition PC_3 characterizes the concrete program executions that violate the assertion, because

$$\texttt{result} = Y - X \leq 0.$$

(In this particular case, the problem is that the assertion is incorrect. It should have been `assert result >= 0;`.) For test input generation, the obtained path conditions are solved, using off-the-shelf decision procedures, and the solutions are used as test inputs that are guaranteed to exercise all the paths through this code.

3. MOTIVATING EXAMPLE

Consider function `site_exec` in Figure 2. It is part of the `wu_ftpd` implementation of the file transfer protocol, converted from C to Java. Its purpose is to receive and execute remote commands. If the command extracted from the input contains the substring "`%n`", a runtime exception is thrown. Although this situation is harmless in Java, in the original C implementation it could potentially allow the user to alter the program stack and to take control of the FTP server. Detecting this kind of code injection is one of the important applications of symbolic string execution, and this example, although somewhat artificial, illustrates a typical scenario. This example is taken from [6] and is based on a real error [2].

An input string s_1 triggers the runtime exception if it satisfies the following constraints (s_2 and i are auxiliary variables):

$$\begin{aligned} & s_1.\texttt{indexOf}(\text{`␣'}) = -1 \\ \wedge\ & s_1.\texttt{lastIndexOf}(\text{`/'}) \geq 0 \\ \wedge\ & s_1.\texttt{lastIndexOf}(\text{`/'}) = i \\ \wedge\ & s_1.\texttt{substring}(i) = s_2 \\ \wedge\ & s_2.\texttt{length}() < 19 \\ \wedge\ & s_2.\texttt{contains}(\text{"\%n"}) \end{aligned}$$

```
1  public void site_exec(String cmd)\ {
2    String r; // result
3    String p = "/home/ftp/bin";
4    int j, sp = cmd.indexOf(' ');
5    if (sp == -1) {
6      j = cmd.lastIndexOf('/');
7      r = cmd.substring(j);
8    } else {
9      j = cmd.lastIndexOf('/', sp);
10     r = cmd.substring(j);
11   }
12   if (r.length() + p.length() > 32) {
13     return; // buffer overflow
14   }
15   String buf = p + r;
16   if (buf.contains("%n")) {
17     throw new Exception("THREAT");
18   }
19   execute (buf);
20 }
```

Figure 2: Example of code injection

We refer to the last constraint as a (pure) string constraint, because it involves only string variables and constants. The second last constraint, on the other hand, is a (pure) integer constraint, since s_2.`length` is in essence an integer variable and 19 is an integer constant. The first constraints are *mixed* (i.e., combined integer and string) constraints.

This classification is clearly important, because different decision procedures and constraint solvers are required for different kinds of constraints. The sections that follow describe the details of how the constraints are represented, how and when information is passed between the integer and string solvers, and how string and mixed constraints are handled by automata or bitvector constraint solvers.

4. SOLVING STRING CONSTRAINTS

The core algorithm for our extension is shown in Figure 3. JPF uses an abstract syntax tree representation for constraints and path conditions. Our approach to string and mixed integer/string constraints requires an additional data structure, known as a *string graph* (described in detail in Section 4.1), and the first step is naturally to construct such a graph.

This is followed by a two-step iterative process: (1) the current JPF constraint solver is invoked to take care of integer, real, and boolean constraints, and (2) if this is successful, some of the values obtained (those involved in the mixed constraints) are propagated to the string graph and a string constraint solver is invoked to solve them. In our case, STRINGSOLVER is either AUTOMATONSOLVER or BITVECTORSOLVER. It may be that the string graph constraints are not satisfiable for the given values, in which case the string solver adds new constraints to the path condition and the process is repeated.

Even if the string problem is decidable, it is possible that the combination of the current solver and the string solver is not able to reach a conclusion on its satisfiability. To guarantee that the process terminates, a time limit is used; once this limit is exceeded, the algorithm concludes (perhaps incorrectly) that the constraints are not satisfiable. This is

```
SOLVE(PathCondition pc)
 1  StringGraph sg
 2  (pc, sg) ← BUILDSTRINGGRAPH(pc)
 3  boolean sat ← false
 4  while ¬sat ∧ ¬timeout:
 5      (sat, pc, sg) ← CURRENTSOLVER(pc, sg)
 6      if sat: (sat, pc, sg) ← STRINGSOLVER(pc, sg)
 7  return sat

BUILDSTRINGGRAPH(PathCondition pc)
 8  StringGraph sg ← ∅
 9  for string or mixed constraint c ∈ pc:
10      sg ← sg ∪ HYPEREDGE(c)
11  return PREPROCESS(pc, sg)
```

Figure 3: Core algorithm

the only alternative, since otherwise it will have to produce values that satisfy the constraints.

The subsections that follow describe the string graph and its construction and preprocessing, the automaton-based approach, the bitvector-based approach, and, finally, how new constraints are generated when the string constraints are unsatisfiable, and how this information is fed back to the integer constraint solver.

4.1 String Graphs

Constraints can be classified as either integer, string, mixed integer/string, or other (such as real or boolean). In our implementation, string and mixed constraints are represented in a special kind of graph:

Definition: A *string graph* is a labeled directed hypergraph $H = (X, E, \Omega)$ where the set of vertices $X = I \cup S$ is the union of two disjoint sets I and S that represent integer and string variables, respectively. The set Ω denotes string operations, and the set of directed labeled hyperedges is $E \subseteq E_1 \cup E_2 \cup E_3 \cup E_4$ where $E_n = \Omega \times X^n$. In other words, each labeled hyperedge is a tuple where the first component is an operation, and the other components are vertices.

The vertices in I and S correspond to either constant values or integer or string variables. As explained below, extra vertices of either kind are sometimes added to the string graph. Hypergraphs are not essential to the operation of the algorithm in the sense that it does not rely on any particular property unique to hypergraphs. They are merely a convenient data structure; other graph-based approaches are discussed in Section 6. The use of hypergraphs is common in constraint satisfaction programming [13, pp. 211–212].

4.1.1 Construction

Each string or mixed constraint of the path condition contributes exactly one hyperedge to the string graph, and it is constructed constraint by constraint. Predicate operations (those that return `boolean`) are mapped straightforwardly to hyperedges. For example, the constraint s_1.`equals`(s_2) contributes the hyperedge (equals, s_1, s_2). For other, transformational operations (those that return `char`, `integer`, or `String`) a new auxiliary variable is introduced to represent the result. This variable is added as a vertex to the string graph, which allows the hyperedge to be added as in the case of predicate operations.

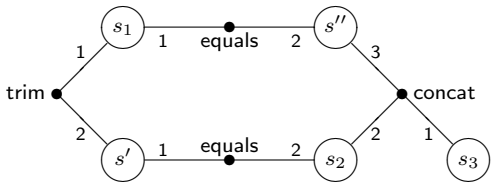

Figure 4: Example of a string graph

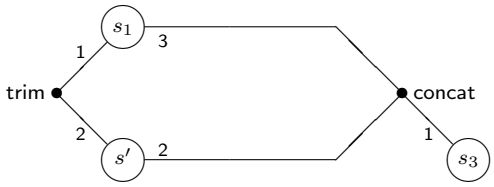

Figure 5: Effect of eliminating equals hyperedges

A more substantial example is shown in Figure 4. Circles depict vertices, dots and lines depict hyperedges. Each hyperedge is labeled with its operation, and small numbers indicate the order of its component vertices. The string graph shown here corresponds to the constraints

$$s_1.\texttt{trim}().\texttt{equals}(s_2) \wedge s_1.\texttt{equals}(s_3.\texttt{concat}(s_2)).$$

The string variables s_1, s_2, and s_3 appear as vertices of S. For this graph, $\Omega = \{\mathsf{trim}, \mathsf{equals}, \mathsf{concat}\}$. The `trim` operation leads to the introduction of auxiliary variable s' which is added to S. The operation itself is a hyperedge (trim, s_1, s') labeled with the trim element of Ω and connected to vertices s_1 and s'. Similarly, the `concat` operation in the second constraint leads to the introduction of auxiliary variable s'' and the hyperedge $(\mathsf{concat}, s_3, s_2, s'')$. Finally, the two `equals` operations are responsible for two more hyperedges.

Once the string graph has been constructed, it is preprocessed in two ways. First, it is simplified to determine whether the constraints are trivially unsatisfiable. Second, additional constraints are derived and added to the path condition.

4.1.2 Preprocessing

A number of heuristics are used to simplify the string graph and to detect trivially inconsistent constraints. This step can avoid expensive invocations of a string constraint solver.

One simple heuristic is the elimination of equals hyperedges. One of the vertices of such a hyperedge is chosen (arbitrarily) and eliminated, and its connected hyperedges are redirected to the remaining vertex. As an example, the string graph in Figure 4 is transformed to the string graph shown in Figure 5. This also allows us to quickly determine cases where a set of constraints represented by a string graph is "trivially" unsatisfiable. Consider Figure 6. After the removal of the equals hyperedges in (a), the s_1 vertex is connected to itself with the notEquals hyperedge as shown in (b), and it is clear that no string can satisfy this constraint.

Most of the other heuristics deal with cases of constant operands. For instance, a constraint such as

$$s_1.\texttt{concat}(\text{“}\texttt{xyz}\text{”}) = \text{“}\texttt{vwxyz}\text{”}$$

would lead to the removal of the hyperedge involved and

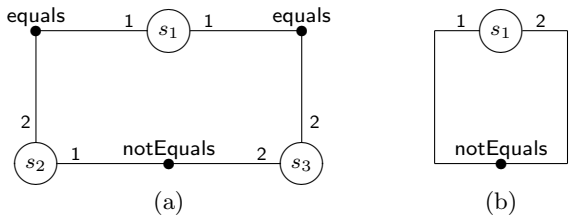

Figure 6: Trivial unsatisfiability

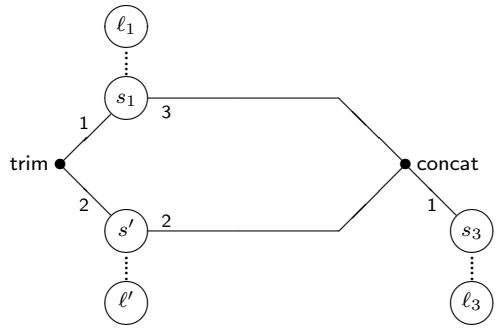

Figure 7: New vertices represent string lengths

instead replace the s_1 vertex with the constant string "`vw`". As another example, the constraints

$$s_1.\texttt{startsWith}(\text{“}\texttt{abc}\text{”}) \wedge s_1.\texttt{charAt}(0) \neq \text{‘}\texttt{a}\text{’}$$

are clearly inconsistent and easy to detect. If such a situation arises, the preprocessing will short-circuit the rest of the SOLVE procedure and immediately return **false** to the caller.

4.1.3 Constraint Generation

Before any constraints are generated, new vertices are added to the string graph: For each non-constant string vertex $s_i \in S$, an integer variable vertex $\ell_i \in I$ is added, if not already present, to represent the length of the string. Figure 7 shows the new vertices for the string graph of Figure 5. For clarity, each new vertex is connected to the corresponding string vertex by a dotted line, but the lines are not technically part of the string graph.

Some of the new vertices may be involved in other string constraints, but it is also possible that some, perhaps all, are not and that the string graph is therefore unconnected. Nevertheless, the vertices play a crucial role. The built-in JPF constraint solver will find values for these variables (in line 5 of Figure 3), the values will be incorporated into the string graph, and the string constraint solver will use them and if necessary update them with new constraints. For each string-length vertex ℓ_i, the constraint $\ell_i \geq 0$ is added to the path condition.

Next, more constraints are added based on the hyperedges. These are shown in Table 1. Note that all but the last three operations introduce new auxiliary variables. In the case of `charAt` the new variable c is a character (which, for all intents and purposes, is treated as an integer); for `indexOf` and `lastIndexOf` the new variable is an integer x; in the other cases it is a string s_x.

4.2 The Automaton Approach

```
AUTOMATONSOLVER(PathCondition pc,
      StringGraph sg = (I ∪ S, E, Ω))
1  for string vertex s_i ∈ S:
2     FiniteAutomaton M_i ← [ℓ_i]
3  W ← E
4  while W contains a positive hyperedge:
5     remove any positive hyperedge e from W
6     for s_i connected to e:
7        update M_i based on the recipe for e
8        if M_i is empty:
9           pc ← pc ∪ FRESHCONSTRAINTS(pc, sg)
10          return (false, pc, sg)
11       else if M_i has changed:
12          W ← W ∪ {all hyperedges connected to s_i}
13 return CHECKNEGATIVEEDGES(pc, sg, (M_1, M_2, ...))
```

Figure 8: The automaton-based solver algorithm

Finite automata are a natural choice for the representation of string variables: the language of an automaton is a set of words in the same way that a string variable can store any one of a set of words that satisfy the constraints. Constraints themselves correspond to automaton operations (such as intersection and union), as we explain shortly.

We are not aware of a freely available automaton-based string constraint solver that accepts a set of constraints and returns a satifiable/unsatisfiable result along with values for the strings. Instead, our implementation uses a home-made solver that takes the whole of the string graph as input, iteratively translates the hyperedges, and uses a standard automaton toolkit (JSA [3] enhanced with some of our own routines) to produce its results.

The automaton-based solution is shown in Figure 8. An automaton M_i is constructed for each string vertex s_i of the string graph. (For clarity, the notation $[n]$ denotes the automaton for the regular language $\cdot^n$, and $[cm]$ for the regular language c^m.) The ℓ_i string-length variables are instantiated by the integer constraint solver, and hence, for each automaton constructed in line 2, all of its words have a fixed length. Of course, different automata may accept words of different lengths.

As the hyperedges are processed, the automata are modified to reflect the effect of the constraint. The exact nature of the modification depends on the constraint. For example, given hyperedge $(\mathsf{contains}, s_a, s_b)$ corresponding to the constraint $s_a.\texttt{contains}(s_b)$, suppose that M_a is the automaton for vertex s_a, and M_b for s_b. The updated M'_a and M'_b are

$$\begin{aligned} M'_a &:= M_a \cap ([*] \oplus M_b \oplus [*]) \\ M'_b &:= M_b \cap \mathit{substrings}(M_a, 0, \infty). \end{aligned}$$

In other words, whatever set of words M_a currently accepts, it is restricted to those words that contain s_b as a substring. Correspondingly, whatever set of words M_b accepts is restricted to substrings of the words of M_a.

Similar recipes for other operations are shown in the centre column of Table 2. The notation $M :\overset{\cap}{=} X$ is shorthand for $M' := M \cap X$, $\oplus$ denotes language concatenation, $\mathit{substrings}(M, i, j)$ is an operation that returns all possible subwords of words of M starting at the ith character and ending before the jth, $\mathit{suffixes}(M)$ and $\mathit{prefixes}(M)$ return

Table 1: String operations, hyperedges, and constraints

Java expression	Hyperedge	New constraints
For each non-constant vertex $s_i \in S$		$\ell_i \geq 0$
s_a.`charAt`(n)	(charAt, s_a, n, c)	$\ell_a \geq n$
s_a.`concat`(s_b)	(concat, s_a, s_b, s_x)	$\ell_x = \ell_a + \ell_b$
s_a.`indexOf`(s_b)	(indexOf, s_a, s_b, x)	$(x = -1) \vee (\ell_a \geq \ell_b + x)$
s_a.`lastIndexOf`(s_b)	(lastIndexOf, s_a, s_b, x)	$(x = -1) \vee (\ell_a \geq \ell_b + x)$
s_a.`substring`(n)	(substring, s_a, n, s_x)	$\ell_a = n + \ell_x$
s_a.`substring`(n, k)	(substring, s_a, n, k, s_x)	$(\ell_a \geq n + k) \wedge (\ell_x = k)$
s_a.`trim`	(trim, s_a, s_x)	$\ell_a \geq \ell_x$
s_a.`contains`(s_b)	(contains, s_a, s_b)	$\ell_a \geq \ell_b$
s_a.`endsWith`(s_b)	(endsWith, s_a, s_b)	$\ell_a \geq \ell_b$
s_a.`startsWith`(s_b)	(startsWith, s_a, s_b)	$\ell_a \geq \ell_b$

all possible suffixes and prefixes of words of M, respectively, and $trim(M)$ returns all words of M without leading and trailing whitespace.

Unfortunately, the automaton approach is unable to generate the solutions for negative constraints (i.e., notEquals, notContains, notStartsWith, and notEndsWith). Suppose that M_1 and M_2 correspond to two strings that are constrained to be unequal. Three cases arise: (1) if both automata accept only a single word, the inequality is satisfiable if and only if the words differ; (2) if only one of the automata accepts a single word, the word can be removed from the language of the other automaton and the inequality is satisfiable; (3) in all other cases, the constraint is satisfiable and neither automaton needs to be modified. Unfortunately, this logic begins to break down when more than two automata are involved and other kinds of negative constraints are added. In short, it is not feasible to impose these constraints on the automaton level. The hyperedges are therefore partitioned into positive and negative hyperedges, the latter being edges of one of the four kinds mentioned at the start of this paragraph.

The main body of the algorithm in Figure 8 follows a typical worklist pattern. Initially W contains all hyperedges. The positive hyperedges are removed one-by-one and processed. Any change to M_i causes those hyperedges connected to s_i to be placed in the worklist again. Eventually, though, the algorithm must terminate because all of the automaton modifications are of the form $M' := M \cap X$. In short, the language of each automaton either stays the same or is restricted during each assignment. When a fixed point is reached, each automaton contains exactly those words that would satisfy the constraints.

If during any assignment an automaton is reduced to the empty language, the string graph and the corresponding constraints are known to be unsatisfiable. When this happens, new integer constraints are generated and control returns to the repeat–until loop of procedure SOLVE (in Figure 3). Since the same thing happens in the case of bitvectors, the new constraints are described in Section 4.4.

If control reaches line 13 of the AUTOMATONSOLVER routine, the M_i's contain solutions that satisfy all of the positive constraints. All that remains is to check for and find words

```
CHECKNEGATIVEEDGES(PathCondition pc,
      StringGraph sg, M = (M_1, M_2, ..., M_k))
 1  i ← 1, M'_1 ← M_1, M'_2 ← M_2, ..., M'_k ← M_k
 2  while 1 ≤ i ≤ k:
 3     if M_i is empty:
 4        M_i ← M'_i, i ← i − 1
 5     else:
 6        v_i ← pick some a ∈ M_i
 7        M_i ← M_i \ {a}
 8        for each hyperedge e such that last(e) = s_i:
 9           if e is violated by v_i:
10              break out of for-loop and go to line 2
11        i ← i + 1
12  return (i > k, pc, sg)
```

Figure 9: The automaton-based solver algorithm

that also satisfy the negative constraints. Because the words of each automaton have a fixed length, one solution would be to simply enumerate all the words of all the automata until a combination is found that satisfies the negative constraints. This is effectively what happens in Figure 9 where a depth-first search of the combinations is used to find a satisfying assignment $(v_1, v_2, \ldots, v_k)$. The code assumes that there is a total ordering among the string vertices and that function $last(e)$ returns the "greatest" string vertex connected to hyperedge e. Our actual implementation uses an optimized version of this approach.

4.3 The Bitvector Approach

The bitvector approach to string constraints is in some sense more straightforward than that of automata, but involves difficulties of its own. Each of the hyperedges is translated as a constraint. The constraints are conjoined and then passed to a constraint solver. Unfortunately, the translation requires a fixed length for each of the vertices. Since these lengths are not known *a priori*, the translation process and the invocation of the solver need to be repeated for all possible lengths, up to a preset bound. Clever use of heuristics can limit the search space, but there are cases that are beyond the scope of this technique.

Table 2: Hyperedge recipes (automaton approach) and hyperedge constraints (bitvector approach)

Hyperedge	Recipes	Constraints
$(\mathsf{charAt}, s_a, n, x)$	$M_a :\sqsubseteq [n] \oplus \{x\} \oplus [*]$	$s_a[n] = x$
$(\mathsf{concat}, s_a, s_b, s_x)$	$M_a :\sqsubseteq \mathit{substrings}(M_x, 0, \ell_a)$	$(s_a = s_x[0{:}\ell_a]) \wedge (s_b = s_x[\ell_a{:}\ell_x])$
	$M_b :\sqsubseteq \mathit{substrings}(M_x, \ell_b, \infty)$	
	$M_x :\sqsubseteq M_a \oplus M_b$	
$(\mathsf{indexOf}, s_a, s_b, x)$	$M_a :\sqsubseteq ([x] \cap \neg M_b) \oplus M_b \oplus [*]$	$(s_a[x{:}x+\ell_b] = s_b) \wedge \left(\bigvee_{0 \le i < x} s_a[i{:}i+\ell_b] \neq s_b\right)$
	$M_b :\sqsubseteq \mathit{substrings}(M_a, c, c+\ell_b)$	
$(\mathsf{lastIndexOf}, s_a, s_b, x)$	$M_a :\sqsubseteq [*] \oplus M_b \oplus ([c-\ell_b] \cap \neg([*] \oplus M_b \oplus [*]))$	$(s_a[x{:}x+\ell_b] = s_b) \wedge \left(\bigvee_{x < i \le \ell_a - \ell_b} s_a[i{:}i+\ell_b] \neq s_b\right)$
	$M_b :\sqsubseteq \mathit{substrings}(M_a, c, c+\ell_b)$	
$(\mathsf{substring}, s_a, n, s_x)$	$M_a :\sqsubseteq [n] \oplus M_x \oplus [*]$	$s_a[n{:}n+\ell_x] = s_x$
	$M_x :\sqsubseteq \mathit{substrings}(M_a, n, \infty)$	
$(\mathsf{substring}, s_a, n, k, s_x)$	$M_a :\sqsubseteq [n] \oplus M_x \oplus [*]$	$s_a[n{:}n+\ell_x] = s_x$
	$M_x :\sqsubseteq \mathit{substrings}(M_a, n, k)$	
$(\mathsf{trim}, s_a, s_x)$	$M_a :\sqsubseteq [\textvisiblespace, *] \oplus M_x \oplus [\textvisiblespace, *]$	$(s_x[0] \neq \textvisiblespace) \wedge (s_x[\ell_x - 1] \neq \textvisiblespace)$
	$M_b :\sqsubseteq \mathit{trim}(M_a)$	$\wedge \bigvee_{0 \le i < \ell_a - \ell_x} \left(\begin{array}{ll} & \bigwedge_{0 \le j < i} s_a[j] = \textvisiblespace \\ \wedge & s_a[i{:}i+\ell_x] = s_x \\ \wedge & \bigwedge_{i+\ell_x \le j < \ell_a} s_a[j] = \textvisiblespace \end{array} \right)$
$(\mathsf{contains}, s_a, s_b)$	$M_a :\sqsubseteq [*] \oplus M_b \oplus [*]$	$\bigvee_{0 \le i < \ell_a - \ell_b} s_a[i{:}i+\ell_b] = s_b$
	$M_b :\sqsubseteq \mathit{substrings}(M_a, 0, \infty)$	
$(\mathsf{endsWith}, s_a, s_b)$	$M_a :\sqsubseteq [*] \oplus M_b$	$s_a[\ell_a - \ell_b{:}\ell_a] = s_b$
	$M_b :\sqsubseteq \mathit{suffixes}(M_a)$	
$(\mathsf{startsWith}, s_a, s_b)$	$M_a :\sqsubseteq M_b \oplus [*]$	$s_a[0{:}\ell_b] = s_b$
	$M_b :\sqsubseteq \mathit{prefixes}(M_a)$	

Table 3: Results for real-world inputs

	Automata			Bitvectors				
Model	Str.	Int.	Iter.	Str.	Int.	Iter.	Preproc.	Constr.
Short	3125	22	5	752	7	0	0	65
Full	388508	107	294	482515	204	0	219	35682
WU-FTPD	2677	500	119	1465	376	59	0	6
EasyChair	1481	0	2	282	0	1	1	15

(all times in milliseconds)

Consider, again, the hyperedge $(\mathsf{contains}, s_a, s_b)$ that corresponds to $s_a.\texttt{contains}(s_b)$. Suppose that the current estimates for the lengths of s_a and s_b are 5 and 3, respectively. The resulting constraint for this hyperedge is

$$\begin{array}{rl} & (s_a[0] = s_b[0] \wedge s_a[1] = s_b[1] \wedge s_a[2] = s_b[2]) \\ \vee & (s_a[1] = s_b[0] \wedge s_a[2] = s_b[1] \wedge s_a[3] = s_b[2]) \\ \vee & (s_a[2] = s_b[0] \wedge s_a[3] = s_b[1] \wedge s_a[4] = s_b[2]) \end{array}$$

If, as in our case, the underlying constraint solver supports arrays, this can be simplified to

$$s_a[0{:}2] = s_b \vee s_a[1{:}3] = s_b \vee s_a[2{:}4] = s_b$$

The translation of string graph hyperedges to bitvector constraints are shown in the right-hand column of Table 2.

4.4 Generating Fresh Constraints

If the string constraints are found to be unsatisfiable, new integer constraints are added to the path condition, and the while-loop in lines 4–6 of Figure 3 is repeated. New constraints are produced by the freshConstraints routine (line 9 of Figure 8 and by a similar invocation in the bitvector solver).

The most straightforward approach is to simply add the negation of the conjunction of the current set of integer assignments. For example, the integer variables in the example in Section 3 are ℓ_1 (the length of s_1), ℓ_2 (the length of s_2) and i. Suppose that the current integer solutions for these variables are 5, 4, and 3, respectively. If no solution is found for s_1 and s_2, the following new constraint is added:

$$\ell_1 \neq 5 \vee \ell_2 \neq 4 \vee i \neq 3.$$

If information about the particular string constraint that failed is available, more nuanced constraints are possible. For example, the system may "realize" that only the value of i is incorrect and add the simpler new constraint:

$$i \neq 3.$$

Such simpler constraints require more analysis to generate. Also, although they may generally improve the execution time for string constraint solving, there is no guarantee that this is always this case.

5. EXPERIMENTAL ANALYSIS

We evaluate our work in two dimensions: first we see how well our implementation performs on three real-world examples, and secondly, we take a closer look at the differences between automata- and bitvector-based back-end analyses to determine the strengths and weaknesses.

5.1 Real-World Examples

We looked at four variations of three real-world programs. The software (mentioned in the introduction) that led to an infinite loop during input sanitization was the cause of a long outage at a prominent internet-based company. We use two versions: a concise version (**Short**, 54 LOC) that has only enough of the code to expose the infinite loop and the complete code (**Full**, 311 LOC). The motivating example in Section 3 comes from a paper by Hooimeijer et al. [6] (**WU-FTPD**, 20 LOC). According to the authors, an error in the string formatting was exploited in a real-world security attack. We use a semantically equivalent version of the original, since Java string operations support `lastIndexOf` which was not available in the original C code. Note that although the example was used to motivate the work in [6], it was never stated whether their tool could actually find the error in this code. Lastly, the running example from [1] forms part of the code for the EasyChair reviewing system (**EasyChair**, 21 LOC). Here we are interested in generating all the paths through the code.

The results are shown in Table 3. We recorded the amount of time spent (milliseconds) on solving the String constraints (the "Str." columns) and on solving Integer constraints (the "Int." columns), the number of times the string and integer constraints had to be iterated before a solution was found (the "Iter." columns), the number of times the preprocessor found the constraints to be unsatisfiable (column "Preproc.") and lastly the total number of times constraints were checked for satisfiability (column "Constr."). Except in the case of EasyChair, the runs stopped once an error was reached. For the EasyChair example, all paths were generated. The number of times the preprocessor was called and the total number of constraints seen, is independent of the back-end solver, so these numbers are the same for both cases.

From the results it is clear that solving the string part of the constraints dwarfs the time spent to solve the integer parts. Secondly, there is not much iteration between string and integer constraints, except for WU-FTPD where the satisfiability checks are done just 6 times, but there are 119 and 59 iterations respectively, i.e., 20 and 10 iterations per constraint on average. From the code for WU-FTPD (see Figure 2), one can see there are `indexOf`, `lastindexOf` and `length` calls which all introduce mixed integer and string constraints and can lead to iterations between the string and integer solvers. Besides the full version of Termination, all the runs finish in less than 5 seconds and Z3 is faster than automata. For the full version, automata is about 25% faster than Z3 and the total time is around 400 seconds, versus 500 seconds. A last observation here is that preprocessing didn't find many cases trivially unsatisfiable, which is in stark contrast to what happens with the random constraints described below.

5.2 Random Inputs

In order to better understand the behavior of automata versus bitvectors we decided to generate random inputs and compare the performance. Firstly, it must be said that this proved to be an invaluable exercise for us to find subtle errors in our implementation! Random testing for this kind of application, where you have two approaches and hence a ready-made oracle, is simply a requirement (rather than an option). Table 4 shows the results. As before we are interested in the time spent for String versus Integer solving and the number of times we iterate between them. For each of these we show the maximum, minimum, average and median values. In addition we partition the data into five groups: both techniques timed-out (set for 120 seconds), automata timed-out, bitvectors (i.e., Z3) timed-out, and then two categories where neither timed-out, but where we look at the difference between results for satisfiable and unsatisfiable constraints. We generated 3069 inputs each with at most 10 conjuncts and the preprocessor found 2887 trivially unsatisfiable, which again is to be expected with random generation of constraints. The results shown here pertains to the ones that passed preprocessing and was handled by

the two back-end analyses.

From the large differences one can observe between the average and median values it is obvious that some outliers influence the average results. We will discuss these in more detail below and for now we only consider the median values. As before we can see that the string analysis takes longer than the integer analysis, but not by as much as in the real-world examples. Next notice that when Z3 times out then the automata takes a long time to determine satisfiability, which means that the constraints are somehow "hard". However when automata times out, Z3 finishes in a blink of an eye, which indicates there is some weakness in the automata approach for these examples. From the last two categories one can see that in general showing satisfiability is much faster than showing constraints are unsatisfiable. Lastly, when automata do not timeout they are in the median case considerably faster than Z3. Although we don't show the data, the more iterations there are the longer the analysis will take, hence optimizing the information gathered during the string analysis to better constrain new length calculations are very important.

Table 4: Results for random inputs

	Automata			Bitvectors		
	Str.	Int.	Iter.	Str.	Int.	Iter.
Both automata and bitvectors time out: 13 cases						
Bitvectors time out: 3 cases						
Min	38278	38092	810			
Max	53375	52840	900			
Mean	44869	44614	850			
Median	42953	42910	840			
Automata time out: 5 cases						
Min				7	0	1
Max				21313	13422	511
Mean				4662	2792	129
Median				17	0	1
UNSAT input problem: 25 cases						
Min	0	0	0	4	0	1
Max	49872	49255	900	57052	53697	900
Mean	11342	10977	305	14673	12956	311
Median	115	49	60	796	160	84
SAT input problem: 136 cases						
Min	0	0	0	1	0	1
Max	54678	46894	900	54547	52130	901
Mean	774	351	12	437	389	11
Median	2	0	0	7	0	1

(all times in milliseconds)

6. RELATED WORK

Work similar to ours is ongoing. There are three main axes of comparison: a graph-like representation of constraints, interaction between integer, string, and mixed constraints, and the use of automata or bitvectors.

Christensen, Møller, and Schwartzbach focus on static analysis, not symbolic execution [3]. They extract the string operations from an input program and construct a *flow graph* which is then translated to *multi-level finite automata*. Their graph is very different from our *string graph*, which captures the relationship between string variables at a given moment; as the name suggests, their graph is akin to a control flow graph and describes the transformations string variables undergo. From the graph they construct a context-free grammar which they approximate with automata, and then use to produce a set of minimal deterministic automata that capture upper bounds on the values of the string variables.

Hooimeijer and Weimer describe a lazy approach to solving string constraints [7, 8]. They too construct an intermediate graph representation of the problem and use depth-first search to traverse and solve the graph. Their goal is to use automata to expend the minimal amount of work to find a set of satisfying solutions. They do not consider the interaction between integer and string constraints, and it is not clear how this could be implemented using their approach. On the other hand,

Kieźun et al. developed HAMPI, a static solver for string constraints [10, 9]. HAMPI translates string constraints (expressed in a custom language) to bitvector constraints, which are then given over to a third-party SMT solver. HAMPI is also able to solve constraints over fixed-size context-free languages. There is no intermediate graph and no support for integer or mixed constraints.

Saxena et al. [14] describe the Kudzu tool for symbolic execution of JavaScript code. Its subtool Kaluza uses HAMPI to translate string constraints from Kudzu's core notation to bitvector constraints. A graph is used, but only to keep track of concatenation dependencies. As in our work, the tool iterates between integer constraints and string constraints, but the only integer constraints are those on string lengths.

Bjorner, Tillmann, and Voronkov investigate string constraint solving as part of Pex, another symbolic execution tool [1, 17, 18]. Integer, string, and mixed constraints are supported and, as in our approach, the system iterates between different decision procedures and constraint solvers.

Shannon et al. also extend JPF to handle strings during symbolic execution [16, 15]. They use automata to represent strings and a somewhat ad-hoc approach to integrate integer and string constraints.

Yu, Bultan, and others [19, 20, 21, 22, 23] use static analysis and automata in their STRANGER tool. Automata are presented as multi-terminal binary decision diagrams (MBDDs) which are used to compute fixpoints, based on a dependency graph produced by the static analysis. In some sense, the dependency graph represents all constraints at once; they are not computed one by one as the execution paths are explored.

One advantage that some of the approaches above offer, is that they handle regular expressions, and some support the `valueOf` operation. We believe that our approach can be extended to include these features.

7. CONCLUSION

The experimental results paint a mixed picture. Except for a small set of inputs, automata clearly outperformed bitvectors for random string graphs, while bitvectors fared better on the real-world examples except for the largest model.

What can we conclude from this? We believe that the choice of decision procedure is not the determining factor for performance. The real critical points are (1) the heuristics implemented in the preprocessor, and (2) the "glue" that

passes information between the integer and string solvers.

This confirms the observations of Veanes et al., who suggest that solvers for different domains should be combined in one solver [18]. It is worth pointing out, however, that even inside a single solver a serious problem remains: if an array-like representation of strings (such as bitvectors) is used, the solver may be committed to a set of string lengths. When the constraints are not satisfied, the string lengths need to be adjusted and the constraints need to be re-solved. Furthermore, this process may need to iterate until suitable lengths are found or some predetermined bound is reached. (The latter is necessary because the problem is fundamentally undecidable.)

8. REFERENCES

[1] Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proc 15th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #5505, pages 307–321. Springer, March 2009.

[2] Cert Advisory. Multiple vulnerabilities in WU-FTPD. Technical Report CA–2001–33, CERT/CC, 2001.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc 10th Intl Symposium on Static Analysis*, LNCS #2694, pages 1–18. Springer, June 2003.

[4] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans Software Engineering*, 2(3):215–222, May 1976.

[5] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc 14th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #4963, pages 337–340. Springer, March 2008.

[6] Pieter Hooimeijer, David Molnar, Prateek Saxena, and Margus Veanes. Modeling imperative string operations with transducers. Technical Report MSR–TR–2010–96, Microsoft, July 2010.

[7] Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proc 2009 ACM SIGPLAN Conf on Programming Language Design and Implementation*, pages 188–198. ACM, June 2009.

[8] Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proc 25th IEEE/ACM Intl Conf on Automated Software Engineering*, pages 377–386. ACM, September 2010.

[9] Adam Kieżun. *Effective Software Testing with a String-Constraint Solver*. PhD thesis, Massachusetts Institute of Technology, USA, June 2009.

[10] Adam Kieżun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: a solver for string constraints. In *Proc 18th ACM/SIGSOFT Intl Symposium on Software Testing and Analysis*, pages 105–116. ACM, July 2009.

[11] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[12] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *Proc 17th ACM/SIGSOFT Intl Symposium on Software Testing and Analysis*, pages 15–26. ACM, July 2008.

[13] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[14] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proc 31st IEEE Symposium on Security and Privacy*, pages 513–528. IEEE Computer Society, May 2010.

[15] Daryl Shannon, Indradeep Ghosh, Sreeranga P. Rajan, and Sarfraz Khurshid. Efficient symbolic execution of strings for validating web applications. In *Proc 2nd Intl Workshop on Defects in Large Software Systems*, pages 22–26. ACM, July 2009.

[16] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *Proc Testing: Academic and Industrial Conf, Practice and Research Techniques*, pages 13–22. IEEE Computer Society, July 2007.

[17] Nikolai Tillmann and Jonathan de Halleux. Pex–white box test generation for .NET. In *Proc 2nd Intl Conf on Tests and Proofs*, LNCS #4966, pages 134–153. Springer, April 2008.

[18] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *Proc 3rd Intl Conf on Software Testing, Verification and Validation*, pages 498–507. IEEE Computer Society, April 2010.

[19] Fan Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for PHP. In *Proc 16th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #6015, pages 154–157. Springer, March 2010. Tool paper.

[20] Fan Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *Proc 15th Intl SPIN Workshop on Model Checking Software*, LNCS #5156, pages 306–324. Springer, August 2008.

[21] Fan Yu, Tevfik Bultan, and Oscar H. Ibarra. Relational string verification using multi-track automata. In *Proc 15th Intl Conf on Implementation and Application of Automata*, LNCS #6482, pages 290–299. Springer, August 2010.

[22] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *Proc 24th IEEE/ACM Intl Conf on Automated Software Engineering*, pages 605–609. IEEE Computer Society, November 2009.

[23] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proc 15th Intl Conf on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS #5505, pages 322–336. Springer, March 2009.