Z3-str: A Z3-Based String Solver for Web Application Analysis

Yunhui Zheng Xiangyu Zhang
Department of Computer Science, Purdue University, USA
{zheng16,xyzhang}@cs.purdue.edu

Vijay Ganesh
Electrical and Computer Engineering, University of Waterloo, Canada
vganesh@uwaterloo.ca

ABSTRACT

Analyzing web applications requires reasoning about strings and non-strings cohesively. Existing string solvers either ignore non-string program behavior or support limited set of string operations. In this paper, we develop a general purpose string solver, called Z3-str, as an extension of the Z3 SMT solver through its plug-in interface. Z3-str treats strings as a primitive type, thus avoiding the inherent limitations observed in many existing solvers that encode strings in terms of other primitives. The logic of the plug-in has three sorts, namely, bool, int and string. The string-sorted terms include string constants and variables of arbitrary length, with functions such as concatenation, sub-string, and replace. The int-sorted terms are standard, with the exception of the length function over string terms. The atomic formulas are equations over string terms, and (in)-equalities over integer terms. Not only does our solver have features that enable whole program symbolic, static and dynamic analysis, but also it performs better than other solvers in our experiments. The application of Z3-str in remote code execution detection shows that its support of a wide spectrum of string operations is key to reducing false positives.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Algorithms

Keywords

String Analysis, String Constraint Solver, Web Application

1. INTRODUCTION

Constraint solving plays an important role in web program analysis for the purpose of test generation for coverage [22], bug finding [3, 32] and vulnerability detection [17, 5]. The reason is that solver-based analysis tools enable more precise analysis with the ability to generate interesting bug-revealing inputs. Furthermore, solver-based analysis tools are often more robust and easier to build than otherwise. While most powerful constraint solvers like Z3 [19] support a rich input language for traditional program analysis, they typically don't support combined logics over strings and non-string operations essential for web program analysis.

In this paper, we present Z3-str, a satisfiability solver that supports a rich combined logic over strings and non-string operations aimed at symbolic, static and dynamic analysis of web applications. Z3-str is built as an extension of the powerful Z3 SMT solver [19] using its plug-in interface. Z3-str treats strings as a primitive type, thus avoiding the inherent limitations observed in many existing string solvers that encode strings in terms of other primitives. The supported logic has three sorts, namely, bool, int and string. The string-sorted terms include string constants and variables of arbitrary length, with functions such as concatenation, sub-string, and replace. The int-sorted terms are standard, with the exception of the length function over string terms. The atomic formulas are equations over string terms, and (in)-equalities over integer terms. Formulas are constructed in the usual way through boolean combination of atomic formulas. Z3-str takes a formula in this logic as input, and decides if it is satisfiable.

Many string solvers such as HAMPI [17], DPRLE [13], and Rex [27] support only string operations. However, such logics are not sufficiently expressive for many program analysis since non-string operations are also widely used in web applications. More importantly, the string and non-string operations interact in subtle ways leading to program errors that are hard for humans to find without automation. Finally, a string-only analysis will likely miss pure integer or string-to-integer constraints (e.g., length of string) thus resulting in path constraints that are not precise enough, leading to false positives.

Many recent works [22, 8, 26, 21] on string solvers have addressed exactly this problem, namely, how to efficiently reason about string and non-string constraints generated during analysis of web applications. A key feature of all these approaches to string solvers is to transform both string and non-string constraints to a uniform domain.

Some solvers [22, 8] (including early versions of Kaluza) convert strings into bit-vectors, which are also used to represent integer constraints. The resulting constraints can be solved using an existing SMT solver. However, to convert a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491456

string into a bit-vector, its length has to be decided beforehand as a prerequisite of the bit-vector encoding. Hence, current techniques have to enumerate the possible length values and then encode the string constraints based on the concrete length values. The encoded constraints are then passed to an SMT solver. This requires encoding repetitively and querying the SMT solver many times. Consequently, these techniques are more suitable for dynamic analysis, wherein string length values can be easily computed from concrete execution paths. However, for static analysis technique, such concretization-based solvers may be either too inefficient or even non-terminating. The reason is that during static analysis of a program, the range of lengths of a string term in the program may not be precisely knowable.

By contrast, our algorithm is designed in an incremental fashion to follow the plugin interaction protocol of Z3, leveraging its power of incremental solving. Given a set of string equations, the algorithm systematically breaks down constant strings into sub-strings and splits variables to sub-variables to denote their sub-structures, until the breakdown is so fine-grained that the variables are bounded with constant strings/characters. Any implied length constraints are checked internally by Z3 for consistency with the given input length constraints.

Contributions. Our contributions include the following:

- We develop a novel string solver that has a rich input language supporting string and non-string terms and operations.
- Since it is unknown if the satisfiability problem of string equations with length constraints in general is decidable, we carefully refine the problem scope so that we can develop a sound and terminating algorithm. Our system is complete for positive equations and length. We do handle dis-equations, but we haven't established completeness for it. We believe the refined problem definition is sufficient for constraints generated from real-world web application.
- We empirically compare Z3-str with other solvers [22, 33] and show that our solver out-performs others in terms of efficiency, in addition to providing a richer input language and supporting both static and dynamic analysis. We also apply the solver in remote code execution vulnerability detection. Our experimental results show that the capability of modeling commonly used string operations is key to reducing false positives.
- Z3-str is publically available at: `http://www.cs.purdue.edu/homes/zheng16/str` [1].
- Z3-str has been successfully evaluated by the ESEC/FSE artifact evaluation committee and found to meet expectations.

2. MOTIVATING EXAMPLE

In this section, we use a piece of php script as an example to demonstrate the limitations of existing solvers and motivate our approach.

The second column of the table in Fig. 1 shows the example php script. In lines 1-13, $role can be assigned to one of four different strings according to the value of the session variable $_SESSION['usergroup']. Lines 14-16 calculate the total cost based on the variable $_POST['price'] submitted by the client. And finally, in lines 17-19, if the cost is larger than 500, a function notifyAdmin() is invoked for the user whose $role starts with the character 'n'.

To successfully invoke the function notifyAdmin(), the input and session variables have to satisfy certain branch conditions at lines $2, 3, 9, 17$ and 18, where both string and non-string constraints are involved. More importantly, the string condition at line 18 is actually related to but not control dependent on the integer conditions at lines $2, 3$ and 9. Therefore, we have to reason about the string and non-string parts together since they are closely correlated with each other.

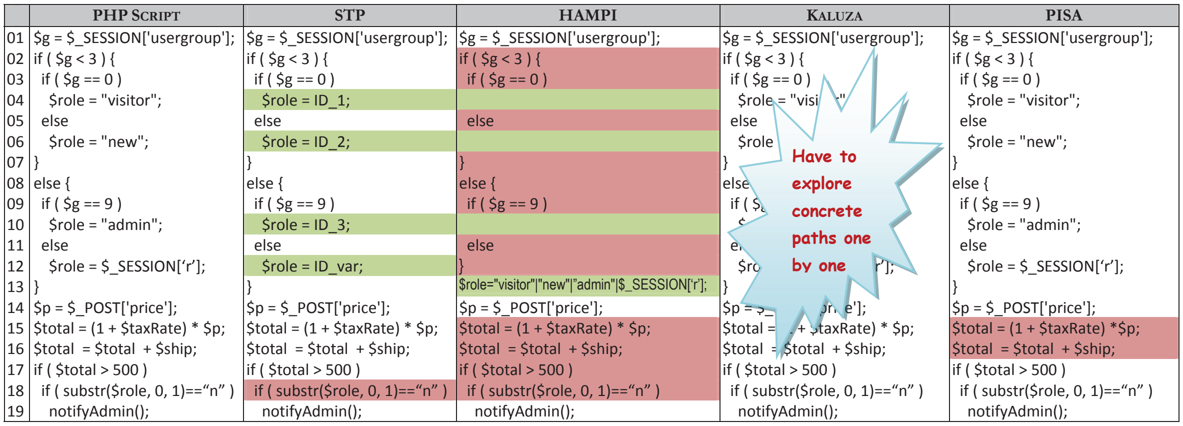
Assume we are interested in identifying the variable valuations that drive the execution to the statement where notifyAdmin() is invoked. Let us compare the ways that existing solvers handle this problem.

STP [10] is a SMT constraint solver. It supports integer and boolean operations but does not support strings. Part of the string equivalence problem could be solved by using integer ids to denote constant strings, as shown by the modification in lines $4, 6, 10$ and 12 marked in green in the STP column of the table in Fig. 1. However, more complex string operations such as concat and substr() are not natively supported so that we have to ignore the predicate at line 18. Therefore, the satisfying solution got from STP states the function will be called as long as the price posted makes the total cost larger than 500, which is imprecise.

HAMPI[17] is one of the most popular open-source string solvers. It was originally designed for detection of SQL injection vulnerabilities. It was then widely used in other web application analysis [5, 22, 25]. It models strings as context free grammars (CFG) and supports regular expressions. String concatenation is modeled based on the CFG. It can handle queries about whether a string expression is in a regular expression or a CFG. However, it does not model arithmetic and boolean operations. More importantly, HAMPI is path insensitive, meaning that it is not able to reason about path feasibility. Thus, in Fig. 1, statements related to the arithmetic computation as well as the path conditions, which are marked in red, are not supported and will be ignored. The grammar rule at line 13 is generated by HAMPI, modeling the possible values of $role. Consequently, HAMPI determines that notifyAdmin() will always be executed. Besides, HAMPI doesn't support direct string comparison and other string operators like indexof() and substr().

Other solvers [9, 24, 13, 14, 27, 31, 26, 2, 21] that work exclusively in the string domain have similar limitations.

Kaluza is the core of a JavaScript dynamic test generation framework [22]. It extends both STP and HAMPI, and supports int, boolean and string constraints generated from an execution path. Kaluza leverages HAMPI's frontend to model strings as bit-vectors so they can be reasoned uniformly with other types of constraints in STP. However, a pre-requesite is that the lengths of bit-vectors have to be known before-hand. Hence, before solving the string values, Kaluza first tries to find a satisfying solution to string lengths. Then it encodes string constraints based on the concrete length values. However, string lengths vary from path to path, and sometimes may be unknown in the static analysis context. For example, in Fig. 1, the length of $role can change in different paths. Hence, multiple paths cannot be encoded. To use Kaluza, one has to have a path exploration engine and perform per-path encoding. Even given a path, if there are multiple satisfying solutions of length constraints, the solver has to enumerate individual concrete solutions and encode separately. This suggests that the STP solver needs to be frequently queried, inducing high cost.



	PHP Script	STP	HAMPI	Kaluza	PISA
01	$g = $_SESSION['usergroup'];	$g = $_SESSION['usergroup'];	$g = $_SESSION['usergroup'];	$g = $_SESSION['usergroup'];	$g = $_SESSION['usergroup'];
02	if ($g < 3) {	if ($g < 3) {	🟥 if ($g < 3) {	if ($g < 3) {	if ($g < 3) {
03	if ($g == 0)	if ($g == 0)	🟥 if ($g == 0)	if ($g == 0)	if ($g == 0)
04	$role = "visitor";	🟩 $role = ID_1;	🟩	$role = "visi r"	$role = "visitor";
05	else	else	🟥 else	else	else
06	$role = "new";	🟩 $role = ID_2;	🟩	$role	$role = "new";
07	}	}	🟥 }	}	}
08	else {	else {	🟥 else {	else	else {
09	if ($g == 9)	if ($g == 9)	🟥 if ($g == 9)	if ($g	if ($g == 9)
10	$role = "admin";	🟩 $role = ID_3;	🟩	$	$role = "admin";
11	else	else	🟥 else	e	else
12	$role = $_SESSION['r'];	🟩 $role = ID_var;	🟥 }	$ro r'];	$role = $_SESSION['r'];
13	}	}	🟩 $role="visitor"\|"new"\|"admin"\|$_SESSION['r'];	}	}
14	$p = $_POST['price'];	$p = $_POST['price'];	$p = $_POST['price'];	$p = $_ pr e'];	$p = $_POST['price'];
15	$total = (1 + $taxRate) * $p;	$total = (1 + $taxRate) * $p;	🟥 $total = (1 + $taxRate) * $p;	$total = + $taxRate) * $p;	🟥 $total = (1 + $taxRate) *$p;
16	$total = $total + $ship;	$total = $total + $ship;	🟥 $total = $total + $ship;	$total = $total + $ship;	🟥 $total = $total + $ship;
17	if ($total > 500)	if ($total > 500)	🟥 if ($total > 500)	if ($total > 500)	if ($total > 500)
18	if (substr($role, 0, 1)=="n")	🟥 if (substr($role, 0, 1)=="n")	🟥 if (substr($role, 0, 1)=="n")	if (substr($role, 0, 1)=="n")	if (substr($role, 0, 1)=="n")
19	notifyAdmin();	notifyAdmin();	notifyAdmin();	notifyAdmin();	notifyAdmin();

Have to explore concrete paths one by one

🟥 Not supported 🟩 Modification needed

Figure 1: Existing solver comparison.

```
1  ...
2  if (substr($x,0,3)=='abc') {
3     if ...
4  } else { ... }
```

Figure 2: Example for Search Space Elimination

Furthermore, building the string solver outside the underlying SMT solver prevents the SMT solver from pruning search space by leveraging the string solving results. Consider the example in Fig. 2. Assume the string comparison at line 2 is UNSAT. With the separated design, the comparison is invisible to the path exploration engine so that the engine has to explore both branches and the paths inside the branches. If the string solver is built inside the SMT solver like in our proposed design, the SMT solver can easily avoid exploring the paths involving the true branch.

A few recent solvers [8, 21] work in a way very similar to Kaluza and thus have the same limitations. They are mostly used in the context of dynamic test generation or symbolic execution in which individual paths are explored, and cannot be directly applied in static analysis. Some of them do not support certain important operations. For example, Kaluza has very limited support of replace, demanding hints from concrete execution traces.

PISA[26] is the first path- and index-sensitive string solver that is applicable for static analysis. It's a part of a commercial tool and the source is not publically available. It supports int, boolean and string operations by translating them all into a third party language M2L(Str). However, its expressiveness for arithmetic operations is restricted due to the limitations of M2L. It does not support binary operations between two variables since PISA requires at least one of them is constant. Therefore, the statements in lines 15, 16 are not supported and the predicate at line 17 may not be correctly modeled. As a result, the reachability analysis is incomplete. Furthermore, it doesn't support numeric multiplications and divisions.

From the above discussion, we make two observations. First, reasoning about both strings and non-strings simultaneously is needed. Second, existing techniques aiming at solving string and no-string constraints together fall short as they rely on other theories (e.g bit-vector) or languages (e.g. M2L), inheriting their limitations. Hence, we aim to develop an independent, more general and more capable string theory inside an existing SMT solver.

3. DESIGN OF Z3-str

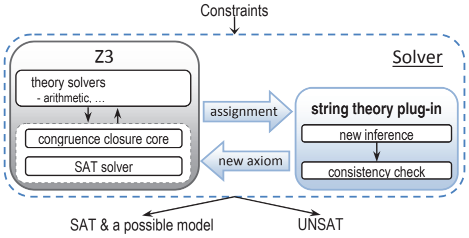


Figure 3: Architecture overview

In this section, we present the details of Z3-str. A key choice in the design is to support strings as a primitive type so that we do not need to convert them to other representations, such as bit-vectors. This avoids determining string lengths before the solving process, which can be very difficult through static analysis. More importantly, it allows us to support unbounded string variables and related operations that otherwise cannot always be supported if lengths have to be determined a priori. Furthermore, building Z3-str using Z3's plugin API also has the advantage of leveraging the state-of-the-art capabilities of Z3.

At the beginning of this section, we briefly introduce the Z3 SMT solver, especially how it interacts with a theory plug-in. Then, we describe the details of Z3-str.

3.1 Z3 SMT Solver and Its Plug-in API

Z3 [19] is an SMT solver developed by Microsoft Research. Input constraints are provided to Z3 as a boolean combination of atomic formulas, where these atomic formulas are defined over theories supported by Z3 such as integer and real arithmetic, bit-vectors, arrays and functions. The architecture of Z3 is shown in Fig. 3. It is mainly composed of the following modules: the congruence closure engine, a SAT solver-based DPLL(T) layer, and several default theory solvers or plug-ins (solvers specialized in solving the satisfiability problem of theories such as integer linear arithmetic, bit-vectors etc.). The congruence closure engine is a key module of Z3. Briefly, it detects if functional terms are equivalent and if so maintains them in appropriate equivalence classes. The SAT-based DPLL(T) layer is responsible for handling the boolean structure of the input formula. For example, it may assert conjunction of theory formulas to the respective solvers, and backtrack if its assertions to the theory solver are unsatisfiable.

The Z3 solver works roughly as follows: The core (congruence closure and DPLL(T) layer) traverses the boolean

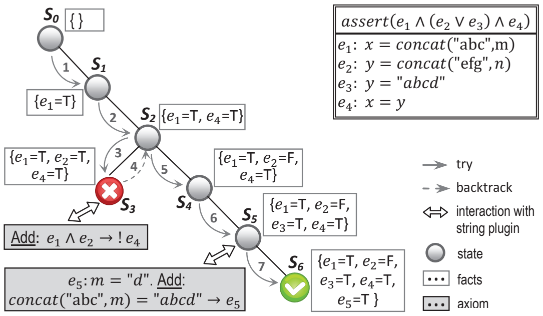


Figure 4: Example of String Solving

structure of the input formula, and asserts a conjunction of literals or facts (atomic formulas or their negation) to the appropriate theory plug-ins based on the type T of the terms used in the literals. The theory plug-in derives new facts in domain T from the asserted facts and conveys them back to the core as new *axioms*. Any equivalence between the terms in the newly derived axioms and existing terms is detected by the congruence closure engine, and appropriate equalities are derived. If the derived equalities and disequalities are over shared variables (i.e., shared by multiple theory plug-ins), then the core asserts them to all such plug-ins. The core detects if the input formula and derived facts are unsatisfiable under the current assignment. If so, it backtracks. This process repeats until a satisfying assignment is produced or the input formula is deemed unsatisfiable.

3.2 String Solving Procedure: Overview

Next, we will use an example to explain how Z3-str works. Consider the string constraint on the right of Fig. 4. Since the core cannot interpret the string operations, it treats them as four independent boolean variables (e_1, e_2, e_3 and e_4) and tries to assign values to them. Initially (state S_0), there is no fact or axiom. The core starts by setting e_1 and e_4 to *true* and reaches state S_2. Then, assume the core tries *true* for e_2 before assigning e_3 at state S_3.

Recall that the core can detect functionally equivalent terms (i.e., based on the theory of uninterpreted functions). Hence, from the facts e_1=*true*, e_2=*true* and e_4=*true*, the core puts $x, y, concat(\text{“abc”}, m)$ and $concat(\text{“efg”}, n)$ into one equivalence class and notifies the plug-in. The plug-in thus knows the two *concat*s above are equivalent. However, from the semantics of concatenation, these two *concat*s cannot equal to each other under any circumstances because they do not share a same prefix. The plug-in informs the core about the new finding through an axiom ($e_1 \wedge e_2 \rightarrow \neg e_4$). With the new axiom and the existing facts, the core detects a conflict on e_4 (in the boolean domain). The core backtracks to state S_2 and tries the other option for e_2. Note that when the core backtracks, it discards the recent fact and any insertions into equivalence classes as the consequence of the fact. Then, the core assigns *true* to e_3 so that “abcd” will be put in the same equivalence class as $concat(\text{“abc”}, m)$. Again, based on the concatenation semantics, the value of string variable m can be inferred by the string theory plug-in, which must be “d”. This new finding is formulated by introducing a new variable (e_5) in an axiom *concat*(“abc”, m)=“abcd”→ e_5, which is sent back to the core. The new state is S_6 in the figure. From the existing facts and the new axiom, the core derives e_5 is *true*.

At state S_6, all boolean expressions have been assigned and the assignments are consistent. Besides, the satisfying values of string variables x, y and m can be retrieved from their equivalence classes. So, a set of consistent and satisfying solutions for the input constraint has been found and the search procedure terminates.

```
Term:bool   ::= (Var:bool)
              | true
              | false
              | contains ( (Term:string), (Term:string) )
Term:int    ::= (Var:int)
              | Number
              | (Term:int) {+, −, ×, ÷} (Term:int)
              | length ( (Term:string) )
              | indexof ( (Term:string), (Term:string) )
Term:string ::= (Var:string)
              | ConstString
              | concat ( (Term:string), (Term:string) )
              | substring ( (Term:string), (Term:int), (Term:int) )
              | replace ( (Term:string), (Term:string), (Term:string) )
Expr:bool   ::= (Term:bool)
              | (Term:bool) = (Term:bool)
              | (Term:int) {<, ≤, =, ≥, >} (Term:int)
              | (Term:string) = (Term:string)
              | not  (Expr:bool)
              | (Expr:bool) ∧ (Expr:bool)
              | (Expr:bool) ∨ (Expr:bool)
              | if (Expr:bool) then (Expr:bool) else (Expr:bool)
              | (Expr:bool) implies (Expr:bool)
Assertion   ::= assert ( Expr:bool )
```

Figure 5: Constraint syntax

3.3 Constraint Syntax

The constraint syntax is presented in Fig. 5. For simplicity, we only list three primitive types: *int, bool* and *string*[1]. Our plug-in supports the following string operations: *string equation*, *concatenation*, *length*, *substring*, *contains*, *indexof*, *replace* and *split*[2].

3.4 Supporting String Operations

In this section, we will explain how the string operations are supported. We first discuss three primitive operations: *string equation, concatenation* and *string length*. Then we will explain how to reduce other string operations to an equivalent formula based on these primitives. Z3 core can directly model the equivalence relation between objects, i.e. variables and constants, by putting them into the same equivalence class. Hence, our overall strategy is to reduce various string operations to simple equivalence relations. The Z3-str algorithm is incremental, driven by the try-and-backtrack procedure of the Z3 core.

3.4.1 String Concatenation

String concatenation is a very commonly used operation. It is also a primitive operation in Z3-str. The plug-in is notified by the Z3 core when a string equation is asserted as part of the try-and-backtrack process. In particular, the core invokes a call back function in the plug-in, providing the abstract syntax tree (AST) of the equation as a parameter. The call back function inspects the AST, if it involves string concatenations, the function tries to perform AST transformation to reduce the AST to a new one that is simpler and easier to resolve.

The reduction is conveyed to the core by adding an axiom with the form of “$AST \rightarrow AST'$” with AST and AST' the original and the transformed syntax trees, respectively. Recall that since the core does not understand the string domain, it treats both AST and AST' independent bool variables. As AST has been assigned a (true) value, with the new axiom, the core will assign a (true) value to AST',

[1] Z3 supports more primitive types [19]

[2] *Split* is not presented in the syntax due to its special format.

DEFINITIONS:

$$
\begin{array}{lll}
Y \in \textbf{CompoundString} & ::= & s \cdot \textbf{CompoundString} \mid x \cdot \textbf{CompoundString} \mid \epsilon \\
x \in \textbf{Var} : \textbf{String} & & s \in \textbf{ConstString} \\
\textit{split} & ::= & \textbf{CompoundString} \to \mathcal{P}(\textbf{CompoundString} \times \textbf{CompoundString} \times \textbf{Expr} : \textbf{bool})
\end{array}
$$

AUXILIARY FUNCTIONS:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textit{split}(Y) = \textit{recSplit}(\textit{nil}, Y)} \\
\quad \textit{recSplit}(Y_L,\ s \cdot Y_R) & = \bigcup_{\forall s_h \cdot s_t = s} \{\langle Y_L \cdot s_h,\ s_t \cdot Y_R,\ \textit{true}\rangle\} \bigcup \textit{recSplit}(Y_L \cdot s,\ \ Y_R) \\
\quad \textit{recSplit}(Y_L,\ x \cdot Y_R) & = \{(Y_L \cdot x_1,\ x_2 \cdot Y_R,\ x_1 \cdot x_2 = x)\} \bigcup \textit{recSplit}(Y_L \cdot x,\ Y_R) \\
\quad \textit{recSplit}(Y_L,\ \textit{nil}) & = \phi
\end{array}
$$

$\textit{len}(s \cdot Y) = \text{“}|s| + \text{”} \cdot \textit{len}(Y)$ $\quad \textit{len}(x \cdot Y) = \text{“}\textbf{length}(x) + \text{”} \cdot \textit{len}(Y)$ $\quad \textit{len}(\textit{nil}) = \text{“0”}$

Figure 6: Definitions for Algorithm 1 and Table 1

which is a new fact triggering further plug-in processing. The reduction continues until the simple equivalent form of expressions, e.g. $x = Y$, with Y a constant string, or a compound string composed of constants and variables, are reached. The dependences of these simple expressions are then constructed so that the plug-in can try assigning values to free variables (i.e. variables do not depend on others), if any, to produce a satisfying solution.

Algorithm 1 Concatenation Reduction Algorithm

```
CALLBACK($Y_1 = Y_2$)
1: REDUCE ($Y_1 = Y_2$);
2: for an object $Y \in Y_1/Y_2$'s equivalence class do
3:     REDUCE ($Y_1 = Y$);
4:     REDUCE ($Y_2 = Y$);
5: if $Y_1 = Y_2$ is of the form $x = s$ then
6:     for any $Y$ involving $x$ in existing eq-classes do
7:         REDUCE ($Y$);
```

Algorithm. The high level concatenation reduction process is presented in Algorithm 1. Upon the core assigning true to a string equation of two compound strings (i.e. concatenations of constants and variables as defined in Fig. 6), function CALLBACK() in Z3-str is invoked. The core idea behind the concatenation reduction given in Algorithm 1 is to derive new equations between the sub-strings on two sides of the equation, and over all terms equivalent to these substrings (recall that Z3's core maintains equivalence classes of terms). For instance, assume a string variable x has its equivalence class $\{x, \text{“}a\text{”} \cdot x_1\}$. If a new equation $x = \text{“}ab\text{”}$ is asserted to the plug-in, we can use the concatenation reduction algorithm to derive $x_1 = \text{“}b\text{”}$ from this new fact and the equivalence class. Furthermore, if the new equation is of the form $x = s$ with s a constant string, in lines 5-7, the algorithm traverses all the expressions in existing equivalence classes that involve x and replaces x with s, which may trigger further reduction.

Reduction Rules. The first four rules in Table 1 are concatenation reduction with relevant definitions in Fig. 6. In Table 1, the second column shows the transformation and the third column is the condition of the reduction. Note that the transformation rule is in the form of “*orig_formula→new_formula*”, which is essentially the axiom we add to the Z3 core. For easier presentation, we flatten an AST to a compound string that is a flat concatenation of constants and variables. Note that during flattening, all the consecutive constant strings are concatenated to a longer constant string.

Rule (HEAD-CONST-RMV) removes the left-most common constant substring of the left-hand side (LHS) and the right-hand side (RHS) of an equation. Note that we assume the RHS has the longer constant string without losing generality. If the LHS and the RHS do not share a common prefix, an axiom “¬*orig_formula*” is added, causing backtrack. Rule (TAIL-CONST-RMV) is similar.

After applying the constant removal rules, we get to a point that either the LHS or the RHS starts with a variable. Rule (SPLIT) allows further reduction for such expressions. Without losing generality, assume the LHS is of the form $x_1 \cdot Y_1$, in which x_1 is a variable and Y_1 a compound string. The rule divides the RHS compound string Y_2 to two substrings Y_h and Y_t, by calling function *split*(), and asserts $x_1 = Y_h$ and $Y_1 = Y_t$. The split function may also generate a new constraint C that denotes the side-effect of splitting. The condition is conjoined with the two assertions. Note that there are many ways of splitting Y_2. Hence the *split*() function returns a set of splittings, which are associated with $\veebar$ operator. The operation is similar to a boolean *or* but it requires only one option can be true. For example, $x \veebar y = true$ implies either $x = true, y = flase$ or $x = false, y = true$.

The details of function *split*(Y) is shown in Fig. 6. It is a recursive process by function *recSplit*(Y_L, Y_R), which returns the set of possible splittings. Particularly, it splits the leading literal of Y_R at a time and moves it to the end of Y_L. Hence, the process starts with $Y_L = \textit{nil} \wedge Y_R = Y$ and terminates with $Y_L = Y \wedge Y_R = \textit{nil}$. If the literal is a constant s, it creates multiple splittings that split at different positions in s. If the literal is a variable x, it creates one splitting, which is to split x to x_1 and x_2 with x_1 and x_2 fresh variables. The correlation of variables is asserted in the splitting condition, which will be part of the reduced equation (i.e. C in Rule (SPLIT)).

Consider a compound string “*ab*”·x. The generated splits are $\{(\text{“”}, \text{“}ab\text{”}\cdot x, true), (\text{“}a\text{”}, \text{“}b\text{”}\cdot x, true), (\text{“}ab\text{”}, x, true), (\text{“}ab\text{”}\cdot x_1, x_2, x_1 \cdot x_2 = x)\}$.

An important observation of the split rule is that *the individual options (of the $\veebar$ operator) in the transformed formula are simpler than the original formula*. This guarantees monotonicity in reducing the original formula. Note that these simpler options will be explored and further reduced by Z3-str separately. However, we also add new constraint C to the system during splitting, which may raise the concern of termination. We will further discuss this issue in Sec. 3.5.

Rule (CONCRETIZE) replaces a variable x in $Y_1 \cdot x \cdot Y_2$ with a constant string when they become equivalent. Note that in order to ensure the inserted formula is an axiom, we have to conjoin $x = s$ with the original expression.

Simple Equations. When no more reduction can be conducted, if each equivalence class has a constant, the solving process terminates with a SAT solution. Otherwise, the plug-in builds a dependence graph of variables involved in simple equations, which are equations in the form of $x = Y$ and the equivalence class of x has no more than one non-singleton compound string terms (i.e. not a constant or a variable). For example in $x = Y$, x depends on all vari-

Table 1: Reduction Rules

Rule	Reduction			Condition
HEAD-CONST-RMV	$s_1 \cdot x_1 \cdot Y_1 = s_2 \cdot Y_2$	→	$x_1 \cdot Y_1 = s_3 \cdot Y_2$	if $s_1 \cdot s_3 = s_2$
	$s_1 \cdot x_1 \cdot Y_1 = s_2 \cdot Y_2$	→	$\neg(s_1 \cdot x_1 \cdot Y_1 = s_2 \cdot Y_2)$	if s_1 is not a prefix of s_2
TAIL-CONST-RMV	$Y_1 \cdot x_1 \cdot s_1 = Y_2 \cdot s_2$	→	$Y_1 \cdot x_1 = Y_2 \cdot s_3$	if $s_3 \cdot s_1 = s_2$
	$Y_1 \cdot x_1 \cdot s_1 = Y_2 \cdot s_2$	→	$\neg(Y_1 \cdot x_1 \cdot s_1 = Y_2 \cdot s_2)$	if s_1 is not a suffix of s_2
SPLIT	$x_1 \cdot Y_1 = Y_2$	→	$\veebar_{\forall \langle Y_h, Y_t, C\rangle \in split(Y_2)} (x_1 = Y_h) \ \wedge \ (Y_1 = Y_t) \ \wedge \ C$	
CONCRETIZE	$(x = s) \wedge (Y_1 \cdot x \cdot Y_2)$	→	$Y_1 \cdot s \cdot Y_2$	if $x = s$
STRINGLEN	$Y_1 = Y_2$	→	$len(Y_1) = len(Y_2)$	
FREE-VAR	$Context$	→	$x =$ “” $\veebar$ $x =$ “@” $\veebar$ $x =$ “@@”...	x a free variable

ables inside Y. Note that complex formulas will cause reduction and generate simple formulas. From the dependence graph, we identify free variables, i.e. variables that do not depend on others and do not have constants in their equivalence class. We then assign concrete values to free variables by adding new axioms. Such assignments will be explained when we discuss the string length operation.

Example. Consider the following constraint composed of three clauses with x, y, and z string variables.

$$z = x \cdot y \qquad z = \text{“a”} \cdot w \qquad x \cdot \text{“d”} = \text{“abd”}$$

The solving process is presented in Table 2. The second column shows the fact/assignment from the core; the third column shows the corresponding equivalence class; the fourth column shows the reduction/action. In step 1, the new fact does not trigger any reduction. In step 2, the two compound strings in the equivalence class of z causes reduction by Rule (SPLIT). In step 4, the axiom added in step 3 causes the fact $x =$ “ab”. In steps 5 and 6, the core tries to explore the [1] and [2] options in the axiom added in step 2 but detects conflicts. It then explores the [3] option in step 7. In step 8, no more reduction can be performed, Z3-str hence looks at variable dependences and identifies that w_2 is a free variable. Note that in the current context, we have $y = w_2$ and $w_1 \cdot w_2 = w$ from the [3] option. Assigning an empty string to w_2 produces a SAT solution. □.

3.4.2 String Length

String length is another important primitive operation as other operations can be reduced to it. Since the Z3 core does not understand the semantics of the operation, it simply treats it as an integer variable. Z3-str hence needs to ensure the correlation between the length variable and the corresponding string. The basic design is to generate the corresponding length constraints (in the integer domain) when the plug-in is invoked by the core upon new facts in the string domain. If the new length constraints cause any conflict in the integer domain, the core will backtrack and try a different solution in the string domain.

Rule (STRINGLEN) describes how Z3-str generates length constraint upon a new fact $Y_1 = Y_2$. It calls function $len()$ (defined in Fig. 6) to translate the two compound strings to integer expressions and asserts their equivalence. It translates a constant string to its length and introduces a length variable **length(x)** for a string variable x.

Example. Consider the following three clauses.

$$x_1 = \text{“a”} \qquad x_2 = x_1 \cdot \text{“efg”} \vee x_2 = x_1 \cdot \text{“e”} \qquad \mathbf{length}(x_2) < 3$$

Suppose the core first assigns `true` to $x_1 =$“a”. This fact allows Z3-str to add the axiom $(x_1 =$“a”$) \rightarrow (\mathbf{length}(x_1) = 1)$. Now assume the core tries the first option of the second clause. Z3-str adds the axiom $x_2 = x_1 \cdot$ “efg” $\rightarrow \mathbf{length}(x_2) = \mathbf{length}(x_1) + 3$. It causes a conflict with the third clause in the integer domain. The core then turns to the second option and finds the SAT solution. □

In the previous discussion of concatenation rules, we mention that we can assign any value to a free variable. However a free variable x in the string domain may be constrained by length assertions (in the integer domain) on the variable itself or on other variables that are dependent on x. Since length constraints do not constrain string values but rather their length, we introduce a free variable Rule (FREE-VAR) to allow the core to try to assign predefined constant strings of various lengths to a free variable in order to satisfy length constraints. According to the rule, if the plug-in detects that a string variable x is a free variable in the current context, it adds an axiom which states that the current context implies that x may have constant strings of various lengths. The context is the conjunction of all the facts set by the core upto this point. We have to use the context as the antecedent as x may not be a free variable in a different context. Any conflicts by length constraints will cause the core to try a constant string of different length. Consider the example in the previous section (Table 2). If we have an additional clause $\mathbf{length}(z) > 2$, the assignment of $w_2 =$“” will be UNSAT, the core will further try to assign $w_2 =$“@”, which generates a SAT solution.

3.4.3 Other Operations

The concatenation and length operations are supported by performing incremental reduction and adding new axioms gradually, driven by the try-and-backtrack process. The other operations are supported in a different way. We perform pre-processing to translate them to formulas using concatenation and length operations.

Substring. The substring operator, $\mathbf{substring}(x, i, j)$, takes three arguments. It returns a substring of x starting at position i, and its length is j. The substring operator can be reduced to a formula with concatenation and string length operations. Particularly, as shown by Rule (SUBSTRING), since the return string is a part of the first argument, we break the first argument into three pieces x_1, x_2 and x_3, and assert the middle piece x_2 equals to the return string. We assert the lengths of x_1 and x_2 to respect the position and length constraints.

Contains. The string membership operator $\mathbf{contains}(x_1, x_2)$ takes two parameters and checks whether x_2 is a substring of x_1. According to Rule (CONTAINS), we break x_2 into three pieces x_p, x_2 and x_s. Note that the middle piece is x_2. The negation of **contains** is more challenging as it is essentially an operation with universal quantifier. Our method is to generate solutions for x_1 and x_2 as if the constraint did not exist and use post-processing to check if x_2 is contained in x_1. If so, we force the core to backtrack.

IndexOf. The operation $\mathbf{indexof}(x_1, x_2)$ returns the starting position of substring x_2 in x_1. If x_2 is not a substring of x_1, it returns -1. As shown by Rule (INDEXOF), we break x_1 into three pieces x_{s1}, x_{s2} and x_{s3}. The result value i has

Table 2: Example for concatenation reduction

step	fact	eq-class	reduction/action
1	$z = x \cdot y$	$\{z,\ x \cdot y\}$	
2	$z = \text{“a”} \cdot w$	$\{z,\ x \cdot y, \text{“a”} \cdot w\}$	$x \cdot y = \text{“a”} \cdot w \ \rightarrow$ $\boxed{x = \text{“”} \wedge y = \text{“a”} \cdot w}^{[1]} \ \veebar \ \boxed{x = \text{“a”} \wedge y = w}^{[2]} \ \veebar \ \boxed{x = \text{“a”} \cdot w_1 \wedge y = w_2 \wedge w_1 \cdot w_2 = w}^{[3]}$
3	$x \cdot \text{“d”} = \text{“abd”}$	$\{x \cdot \text{“d”}, \text{“abd”}\}$	$x \cdot \text{“d”} = \text{“abd”} \ \rightarrow \ x = \text{“ab”}$
4	$x = \text{“ab”}$	$\{x, \text{“ab”}\}$	
5	[1]	...	conflict between $x = \text{“ab”}$ and $x = \text{“”}$, backtrack
6	[2]	...	conflict between $x = \text{“ab”}$ and $x = \text{“a”}$, backtrack
7	[3]	$\{x, \text{“ab”}, \text{“a”} \cdot w_1\}$	$\text{“ab”} = \text{“a”} \cdot w_1 \ \rightarrow \ w_1 = \text{“b”}$
8			Identify w_2 as the free variable based on dependences: $z \rightsquigarrow y \rightsquigarrow w_2$ and $w \rightsquigarrow w_2$, assign $w_2 = \text{“”}$
SAT solution: $w_2 = y = \text{“”}$, $x = z = \text{“ab”}$, $w_1 = \text{“b”}$			

Table 3: Preprocessing Rules for Other Operations

Rule	New Formula
SubString	$\textbf{substring}(x, i, j) = x_t \ \rightarrow \ x = x_1 \cdot x_2 \cdot x_3 \ \wedge \ x_2 = x_t \ \wedge \ \textbf{length}(x_1) = i \ \wedge \ \textbf{length}(x_2) = j$
Contains	$\textbf{contains}(x_1, x_2) \ \rightarrow \ x_1 = x_p \cdot x_2 \cdot x_s$
IndexOf	$\textbf{indexof}(x_1, x_2) = i \ \rightarrow \ (x_1 = x_{s1} \cdot x_{s2} \cdot x_{s3}) \ \wedge \ (i = -1 \vee i \geqslant 0) \ \wedge \ ((i = -1) \leftrightarrow \neg\textbf{contains}(x_1, x_2)) \ \wedge$ $((i \geqslant 0) \leftrightarrow (i = \textbf{length}(x_{s1}) \ \wedge \ x_{s2} = x_2 \ \wedge \ (\neg\textbf{contains}(x_{s1}, x_2))))$
Replace	$\textbf{replace}(x_1, x_2, x_3) = x_t \ \rightarrow \ (x_1 = x_{s1} \cdot x_{s2} \cdot x_{s3}) \ \wedge \ i = \textbf{indexof}(x_1, x_2) \ \wedge$ $\textbf{if } (i \geqslant 0) \textbf{ then } x_t = x_{s1} \cdot x_3 \cdot x_{s3} \ \wedge \ \ \textbf{length}(x_{s1} = i) \ \textbf{ else } x_t = x_1$
Split	$\textbf{split}(x_1, x_2) = [x_{t1}, x_{t2}] \ \rightarrow \ x_1 = x_{t1} \cdot x_2 \cdot x_{t2} \wedge \neg\textbf{contains}(x_{t1}, x_2) \wedge \neg\textbf{contains}(x_{t2}, x_2)$

two options: if and only if x_1 doesn't contain x_2, i is -1. Otherwise, if and only if x_{s2} equals to x_2, its predecessor x_{s1} doesn't contain x_2, and t equals to the length of x_{s1}.

Replace. The transformation is shown in Rule (Replace). In the rule, we assume only one substring is replaced. It can be easily extended to support cases in which less than n replacements occur with n a pre-defined constant.

Split. We support **split** in a similar way to that of **replace**. In the rule, we assume only one substring matches the pattern x_2. It can be extended to support cases with less than n occurrences.

3.5 An Improved Algorithm for a Restricted Theory of Strings

Unfortunately, it is unknown if the satisfiability problem of string constraints with length operations is decidable in general [11, 18]. This has a bearing on our algorithm in that it may not terminate for certain cases, stuck in infinite splitting. To avoid such non-termination we restrict the theory in certain ways as described below. For this restricted theory, the satisfiability problem becomes decidable and our algorithm is one such decision procedure.

To better illustrate the problem, observe in Rule (SPLIT), while we are simplifying the original equation, we may introduce new ones, denoted by C, which may cause further splitting. In some cases, the splitting becomes infinite.

$$x \cdot \text{“}b\text{”} = \text{“}a\text{”} \cdot x \rightarrow \ldots \veebar \qquad \textbf{(Ex-I)}$$
$$(\boxed{x = \text{“}a\text{”} \cdot x_1}^{[1]} \wedge x_2 = \text{“}b\text{”} \wedge \boxed{x = x_1 \cdot x_2}^{[2]}) \veebar \ldots$$

Consider the above example, which shows part of the reduction of a string equation with the same variable appears as the prefix of LHS and the suffix of RHS. The equation is not satisfiable. However, according to our reduction rule (SPLIT), one of the options as shown induces new equivalence between the RHS's of [1] and [2]. Observe that the new equation has a very similar form as the original one. Hence, the exploration and reduction will not terminate (i.e. x_1 in [1] will be further splitted).

$$\boxed{x \cdot Y_1 = Y_2 \cdot y}^{[3]} \qquad \boxed{y \cdot Y_3 = Y_4 \cdot x}^{[4]} \qquad \textbf{(Ex-II)}$$

Such non-termination could also occur in equations that involve multiple variables. Assume the above two clauses. Part of their reductions are shown as below. In the first reduction, y is splitted and in the second reduction, x is splitted. Observe that the equivalence between RHS of [5] and RHS of [8], and the equivalence between RHS of [6] and RHS [7] form a recursion of the form of the original two clauses, causing infinite reduction.

$$[3] \rightarrow \ldots \veebar (\boxed{x = Y_2 \cdot y_1}^{[5]} \wedge Y_1 = y_2 \wedge \boxed{y = y_1 \cdot y_2}^{[6]}) \veebar \ldots$$
$$[4] \rightarrow \ldots \veebar (\boxed{y = Y_4 \cdot x_1}^{[7]} \wedge Y_3 = x_2 \wedge \boxed{x = x_1 \cdot x_2}^{[8]}) \veebar \ldots$$

The root cause of non-termination is that the splitting of a variable directly or indirectly causes further splitting of the same variable. In **Ex-I**, the splitting of x introduces new equivalence such that a subpart of it, x_1, sets out to be splitted. However, we observe that such cases rarely happen in the constraints generated from web applications. Therefore, we define a sub-class of the problem that is decidable and leads to an algorithm that ensures termination.

The Refined Problem. We capture the essence of the non-termination issue using a graphical representation of equation in Fig. 7. Such graphical representation was introduced in various studies of string theories [12]. A solid horizontal segment represents a variable or a constant string. We call it the projection of the variable or string. A compound string consisting of a sequence of variables and constant strings is denoted as a sequence of segments that are slightly misaligned vertically for better visibility. The equivalent compound strings (e.g. the LHS and RHS of an equation) are projected to the same graph. The vertical dotted lines represent the boundaries of segments and the alignment across strings. Note that the positions of the boundaries may vary for different solutions.

For example, Fig. 7 (a) shows the graphical representation of example **Ex-I**. Fig. 7 (b) shows a simplified one for **Ex-II**. It is the union of the two projections regarding equation [3] and [4] by unifying the two y's in [3] and [4] and omitting some compound strings for readability. A split corresponds to that the end of a variable segment falls in the middle of another variable segment. We call it a *cut*.

Definition 1. *We say a varaible x cuts y, denoted as $x \downarrow y$, if there is a reduction from $x \cdot Y_1 = Y_2 \cdot y \cdot Y_3$ to $(x = Y_2 \cdot y_1 \ \wedge \ Y_1 = y_2 \cdot Y_3 \ \wedge \ y = y_1 \cdot y_2)$.*

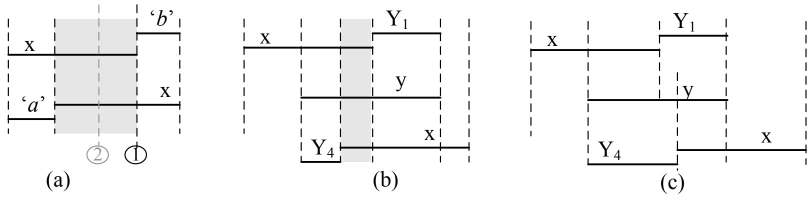


Figure 7: Subgraphs (a) and (b) show the graphical representation of Ex-I and Ex-II, (c) shows a well-formed solution of Ex-II.

Hence, in Fig. 7 (a), the top x (in the LHS of the original equation) cuts the bottom x (in the RHS). In (b), the top x cuts y and y cuts the bottom x.

We observe that non-termination is caused by cases in which the projections from the same variable overlap (but not completely coincide). The shaded regions in Fig. 7 represent the overlapping regions. As such, a cut of a segment will have its projection in other segments of the same variable. These projections may further cut the original segment due to the overlap, resulting in infinite cutting. In Fig. 7 (a), the cut at ① on the bottom x has its projection at ② on the top x and this projection cuts the bottom x again. Figure (b) is similar.

Note that *overlapping of multiple projections of the same variable implies recursive self constraining of the same variable.* For instance, in Fig. 7 (a), it implies the prefix and suffix of x are mutually constrained. Based on our observation, such self constraining is very rare in the program analysis context. Hence, we preclude such cases in order to devise a better algorithm that guarantees termination.

The formal definitions regarding the refined problem are presented as follows.

Definition 2. *Given a solution of a set of string constraints and a variable x, the projection of the solution regarding x uses the segment of x as the base and projects all the other variables that overlap with x in some constraint according to the solution.*

Fig. 7 (b) and (c) are sample solution projections regarding y for **Ex-II**.

Definition 3. *We say a solution is* well-formed *if and only if in the projection regarding any variable, there is not a variable x whose multiple projections overlap.*

Fig. 7 (c) is a well-formed solution but (b) is not as the x segments overlap.

Given the definitions, the problem is refined to **finding well-formed solutions of a set of string constraints**.

The Revised Algorithm. We revise the previous algorithm. The revision mainly lies in the split function. We introduce a few new definitions. Mapping VAR identifies the original string variable (of an intermediate variable). For example in **Ex-I**, $VAR(x_1) = x$. We also annotate each intermediate variable x with the variable y that cuts $VAR(x)$ and induces the generation of x. Note that x^y implies that x is a suffix of y according to the definition of a cut. Initially, we annotate all string variables with themselves.

We revise the recursive split function, particularly the part that splits a variable. The function is invoked with the variable x^z that causes the split. For instance, given an equation $x^z \cdot Y_1 = Y_2$, $recSplit(x^z, nil, Y_2)$ is called to start splitting. If it is to split a variable y in the RHS. It checks if $z == VAR(y)$ to avoid self-splitting. Note that since $x == VAR(x)$ for any original variable x, we trivially prevent direct self-splitting. If the cut is admissable, the splitted variable y_1 inherits the annotation z and y_2 retains the annotation of y. If the cut is not admissable, the variable is not splitted.

Example. Lets revisit **Ex-II**. We annotate the original clauses initially as follows.

$$\boxed{x^x \cdot Y_1 = Y_2 \cdot y^y}^{[3]} \qquad \boxed{y^y \cdot Y_3 = Y_4 \cdot x^x}^{[4]}$$

Part of the reduction of [3] is as follows.

$$[3] \rightarrow ... \veebar (\boxed{x^x = Y_2 \cdot y_1^x}^{[5]} \wedge Y_1 = y_2^y \wedge \boxed{y^y = y_1^x \cdot y_2^y}^{[6]}) \veebar ...$$

Part of the reduction of [4] is the following.

$$[4] \rightarrow ... \veebar (\boxed{y^y = Y_4 \cdot x_1^y}^{[7]} \wedge Y_3 = x_2^x \wedge \boxed{x^x = x_1^y \cdot x_2^x}^{[8]}) \veebar ...$$

From [6] and [7], we have $y_1^x \cdot y_2^y = Y_4 \cdot x_1^y$. However, y_1^x will not cause splitting of x_1^y as $VAR(x_1) = x$, which is the annotation of y_1. Hence, the search will focus on splitting Y_4, which corresponds to the well-formed solution as shown in Fig. 7(c).

4. EVALUATION

Our evaluation consists of two experiments. In the first experiment, we compare the performance of Z3-str with Kaluza, which only supports encoding a single path. In the second experiment, we compare Z3-str with our prior work on a solver that integrates HAMPI and STP and supports encoding multiple paths, in terms of both efficiency and effectiveness. We choose to compare with these two solvers as they can solve string and non-string constraints together. All experiments are run on an Intel Core i5-2520M machine with 8GB memory.

4.1 Comparison with Kaluza

We use the test cases shipped with the Kaluza package to compare performance. Since we currently do not support regular expressions, we remove the constraints related to regular expressions, which account for a small percentage of all the constraints. Then, we run both solvers 100 times for each test case and take average of the execution time. The results are presented in Table 4. We can see Z3-str is faster than Kaluza in 13 out of 14 cases. We also observed 6 solutions provided by Kaluza are partially incorrect. In the last case, Kaluza outperfoms Z3-str. This case has a large set of simple string constraints that are easy to satisfy. We observed a lot of backtrackings due to string length inconsistencies in Z3-str for this case. Further analysis shows that because the version of Z3 we use does not allow us to acquire the concrete values assigned to length variables in the integer domain during the solving process, we cannot leverage the values explicitly to optimize splitting. Although the infeasible splittings are immediately rejected by the integer

DEFINITIONS:

$VAR(x)$ ::= the original string variable of a temporary variable (which is generated by splitting)

Each variable x is annotated with a variable y to denote x was generated by y cutting $VAR(x)$. Initially, we annotate x with x itself.

AUXILIARY FUNCTIONS:

$$\begin{array}{ll}
recSplit(x^z,\ Y_L,\ s \cdot Y_R) & = \bigcup_{\forall s_h \cdot s_t = s} \{\langle Y_L \cdot s_h,\ s_t \cdot Y_R,\ true \rangle\} \bigcup recSplit(x^z,\ Y_L \cdot s,\ Y_R) \\
recSplit(x^z,\ Y_L,\ y^w \cdot Y_R) & = \\
\multicolumn{2}{l}{\quad \begin{cases} \{(Y_L \cdot y_1^z,\ y_2^w \cdot Y_R,\ y_1^z \cdot y_2^w = y^w)\} \bigcup recSplit(x^z,\ Y_L \cdot y^w,\ Y_R) & z! = VAR(y) \\ \{(Y_L \cdot y^w,\ Y_R,\ true)\} \bigcup recSplit(x^z,\ Y_L \cdot y^w,\ Y_R) & otherwise \end{cases}} \\
recSplit(x^z, Y_L, nil) & = \phi
\end{array}$$

Figure 8: New Definitions and Split Function.

theory, Z3-str has to pay the cost of unnecessary splitting and backtracking. In contrast, Kaluza first acquires the concrete length values and then performs encoding. Because these constraints are simple, Kaluza does not need to enumerate multiple solutions for string lengths. Z3 is recently open-sourced. We are looking into if Z3-str can communicate with the integer theory better to improve performance. We also want to point out again that Z3-str is more general in one respect than Kaluza, as Z3-str does support encoding multiple program paths while Kaluza does not.

Table 4: Comparison with Kaluza [22]

	stats		Correct?		Time (s)		
	var	cstr	K	Z	K	Z	K/Z
bettermatch1*	10	8	✓	✓	0.276	0.035	7.9x
bettermatch2*	9	7	✓	✓	0.242	0.036	6.7x
concat	9	9	×	✓	0.216	0.035	6.2x
idxof*	31	40	×	✓	0.632	0.067	9.4x
indexof*	12	14	✓	✓	0.198	0.035	5.7x
match	16	16	×	✓	0.207	0.036	5.8x
replace*	18	19	✓	✓	0.205	0.049	4.2x
search*	8	7	✓	✓	0.193	0.035	5.5x
split*	15	13	×	✓	0.203	0.038	5.3x
streq	11	15	✓	✓	0.187	0.035	5.3x
substr	9	12	×	✓	0.179	0.034	5.3x
substr_idxof*	30	38	×	✓	0.210	0.062	3.4x
big1*	75	91	✓	✓	0.241	0.154	1.6x
big2*	573	713	✓	✓	0.962	2.718	0.4x
Average	–	–	–	–	–	–	5.2x

* Regular expression constraint removed.

4.2 Comparison with Our Prior Work

In our prior work in detecting remote code execution vulnerabilities [33], which allow malicious PHP code to be injected and executed, we developed a solver that supports encoding multiple program paths for static analysis. It combines STP and HAMPI in an *Alternative and Iterative (AI)* fashion. It uses STP to generate sets of feasible paths without considering string behavior. It then uses HAMPI to encode string behavior along those paths. Solving the string constraints determines the true path feasibility. We use the same set of benchmarks to compare the performance. The two solvers are provided the same set of constraints. Each constraint encodes an entire program (after using a static slicer to prune irrelevant parts). The results are listed in Table 5. We use *AI solver* to denote the prior solver. The data suggest that Z3-str performs much better. This is because in *AI solver*, infeasible paths caused by string constraints cannot be detected by STP and have to encoded and passed on to HAMPI, to determine their true feasibility, whereas Z3 can leverage the interal results from Z3-str to avoid them entirely. The frequent queries to STP and HAMPI also cause overhead in process starting and finishing.

Table 5: Comparison with AI Solving [33]

	stats		AI solver		Z3-str	AI solver
application	var	cstr	iteration	real(s)	real(s)	/ Z3-str
aidiCMS v3.55	446	464	0	3.950	0.421	9.4x
phpMyFAQ v2.7.0	296	307	5	6.228	0.187	33.3x
zingiri webshop v2.2.2	300	315	13	47.380	0.109	434.7x
phpMyAdmin v3.4.3	164	150	0	1.053	0.119	8.8x
phpLDAPadmin v1.2.1.1	628	627	39	84.938	0.693	122.6x
phpScheduleIt v1.2.10	514	597	9	6.812	0.228	29.9x
FreeWebshop v2.2.9 R2	1099	1147	62	95.995	0.255	376.5x
ignition v1.3	260	256	1	0.748	0.227	3.3x
monalbum v0.8.7	175	174	0	0.348	0.211	1.6x
webportal v0.7.4	12	9	0	0.359	0.031	11.6x

```
@ fws/adminedit.php
51   $name = explode(".", $_GET['filename']);
67   if ($_POST['action'] == "write_changes") {
68     if ($name[1] == "txt" || $name[1] == "sql") {
69       if ($name[1] == "txt" && ... ) {
70         $fp=fopen($_GET['filename'],"w");
71         fwrite($fp,$_POST['text2edit']);
73       } else {...}   } }
```

Figure 9: FP type 1 in Zingiri webshop

Eliminating False Positives in [33] . Next, we will show that Z3-str allows us to remove all the false positives (FP) in the prior work as Z3-str supports the commonly used string operations while the prior solver does not.

In web applications, clients often can upload files or save user input in a server-side file. Sometimes, the file name can be provided by the client too. If arbitrary user input can be saved in a user specified file ending with ".php", a RCE vulnerability is introduced since the client may inject and execute arbitrary PHP code by writing and manipulating that file. Many web applications have proper protection by having checks on file names before file writes. The reason for the 6 FPs we had before is because we cannot model the file name checking logic due to the string operators used.

Two of the FPs have file name checking similar to that in Fig. 9. Function explode is used to separate a string to substrings. Previously, due to the lack of support of **split**, we cannot model the function and have to use free variables to denote $name. As a result, its relation with $_GET['filename'] is missed. Therefore, the assertions of $_GET['filename'] being ended with ".php" and the reachability conditions at line 71 are SAT. With Z3-str, we can model explode() precisely with **split** and correctly determine that writing to ".php" file is infeasible.

The remaining four FPs are similar to Fig. 10, in which getFileExt() cannot be properly modeled. With Z3-str, the function can be modeled using **substring** and **indexof**. Hence, we can correctly decide that the file write at line 40 is not reachable when the file name extension is ".php".

5. RELATED WORK

There is a vast literature on the problem of solving equations of all manner through unification [16, 15] and term rewriting [4]. Schulz uses a combination of tehcniques from

```
@ modul/tinymce/plugins/ajaxfilemanager/ajax_save_text.php
$path = $not_important . $_POST['name'];
if ( getFileExt($_POST['name']) == "php" )  { ... }
else {
  if( file_exists($path) )  { ...}
  else {
    $fp = fopen($path , "w+");
    fwrite($fp, $_POST['text']);   } }
```

Figure 10: FP type 2 in adidCMS

string-unification and Makanin's algorithms to solve the problem of terminating minimal and complete word unification [23].

Based on the underlining representation, existing string analyses can be roughly categorized into two kinds: automata-based [9, 24, 13, 14, 27, 31, 26, 2, 21] and bit-vector-based [17, 22, 8, 21]. We have made comparison with a number of these existing works [17, 22, 33, 26] in Sections 2 and 4. Hence in this section, we focus on the other works.

Automata/regular-expressions are a natural form to represent strings so that many works are based on them. Java String Analyzer(JSA) [9] applies static analysis to model flow graphs of Java programs. These graphs capture dependencies among string variables. Finite automata can be computed from the graphs to reflect possible string values. Shannon et al. [24] used finite state machines (FSMs) to model strings. String computation is modeled by FSM refinement. They have ad-hoc rules for integer relations but do not support integer constraints in general. Hooimeijer and Weimer [14] designed an heuristics based approach to find a SAT solution. They search lazily to avoid building full automata. As a result, the performance is improved greatly compared to their previous work [13]. Rex[27, 28] uses symbolic automata where labels are represented by predicates. It uses symbolic language acceptor and explores various optimizations of symbolic automata, like minimization, to leverage the underlying SMT solver and eliminate inconsistencies. Also, the trade-offs between the language acceptor based encoding and automata-specific algorithms are discussed [28]. Symbolic automata are implemented in the symbolic automata library [29]. Stranger[31] developed by Yu et al. analyzes strings for PHP based on automata. ViewPoints [2] uses DFA to model client- and server-side input validation functions. The inconsistency between validation functions can be found by comparing their DFAs. [7] surveyed a large set of existing string solvers and compared them using a table.

Using automata/FSM/regular-expressions allows the above techniques nicely support infinite strings and regular expression related operations. However, many of them have difficulties in handling string constraints related to integers like *length* and *substring* (with variable indices). More importantly, many of them do not provide native support for other types of constraints such as integer, which is needed in reasoning about both string and non-string behavior together.

In [30], string automata are extended with arithmetic automata to support integers and length constraints in addition to string constraints. Prefix/suffix operations and Presburger arithmetic constraints on integer variables (i.e., linear arithmetic constraints + Boolean connectives + quantification) are also supported. In comparison, Z3-str is based on term rewriting and leverages Z3, which may allow better scalability and support a richer set of theories.

The *STeP* system[6] had a different rewriting style string solver. It was complete for a specific fragment of strings called "queues", where a concatenation is restricted to having at most one variable. It can solve equations and inequalities. It also used extensions for general concatenation, length and reverse. Z3-str supports multiple variables.

Bjørner et al. [8] proposed a string constraint solving technique to reason about feasibility of a concrete execution path. It works in a way similar to Kaluza, encoding strings into bit-vectors. Hence it needs to enumerate concrete length values. It supports common integer related string operations expect *replace*. Regular expressions are not supported. Redelinghuys et al. [21] developed a constraint solving engine that can handle multiple types of constraints for Java PathFinder. It first ignores string constraints and gets a satisfiable solution for numeric and boolean variables. These concrete values are then used to encode string constraints to bit-vectors (with fixed lengths). If the string part is UNSAT, it tries a different SAT solution from the non-string domains. They can handle many operators. They have limited support for *replace*, requiring the result and arguments must be concrete. They do not handle regular expressions. As discussed in Section 2, the above techniques have similar limitations as Kaluza as they only allow encoding one path and hence not ideal for static analysis. The path exploration engine can hardly leverage the string solving results to prune search space (see Fig. 2).

There have been a lot of theoretical research in general string theory [12, 18, 11, 20]. Some proposed algorithms to solve string equations. However, these algorithms usually do not consider important string operations such as *length*. Z3-str shares some similarity to [18] in splitting variables. In contrast, we refined the problem scope to make the problem decidable and yet sufficient for web program analysis and the algorithm is incremental, driven by the try-and-backtrack procedure of Z3.

6. CONCLUSION

We develop a general purpose string solver, Z3-str, as an extension of Z3. Z3-str treats strings as a primitive type, avoiding the inherent limitations observed in many existing solvers that encode strings in terms of other primitives. It supports string constants and variables of arbitrary length, and commonly used string operations. It allows encoding single or multiple program paths such that it can be used in both dynamic and static analysis. The underlying algorithm based on constant string and variable splitting is sound and guarantees termination for a restricted theory that is sufficient in practice. Our system is complete for positive equations and length. We do handle dis-equations, but we haven't established completeness for it. Our experiments show that Z3-str outperforms other state-of-the-art solvers and its support of various string operations allows us to eliminate all false positives in remote code execution vulnerability detection.

7. ACKNOWLEDGMENTS

We would like to thank Leonardo De Moura and Nikolaj Bjørner for their help and comments. We thank the reviewers for their substantial efforts. This research is supported, in part, by the National Science Foundation (NSF) under grants 0845870, 0917007, 1218993 and Canadian NSERC Discovery Grant 2013. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

8. REFERENCES

[1] `http://www.cs.purdue.edu/homes/zheng16/str`

[2] M. Alkhalaf, S. Choudhary, M. Fazzini, T. Bultan, A. Orso and C. Kruegel. ViewPoints: Differential String Analysis for Discovering Client- and Server-Side Input Validation Inconsistencies. In *ISSTA'12*.

[3] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar and M. Ernst. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. In *TSE, vol.36, no.4, pp.474-494, 2010*.

[4] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz and V. Venkatakrishnan. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *CCS'10*

[6] N. Bjørner. Integrating decision procedures for temporal verification. *Ph.D. thesis, Stanford University, 1999*

[7] N. Bjørner, V. Ganesh, R. Michel and M. Veanes. An SMT-LIB Format for Sequences and Regular Expressions. In *SMT workshop 2012*.

[8] N. Bjørner, N. Tillmann and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *TACAS'09*.

[9] A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *SAS'03*.

[10] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07*.

[11] V. Ganesh, M. Minnes, A. Solar-Lezama and M. Rinard. Word equations with length constraints: what's decidable? In *HVC'12*.

[12] C. Gutiérrez. Solving Equations in Strings: On Makanin's Algorithm. In *LATIN'98*

[13] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI'09*.

[14] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *ASE'10*.

[15] J. Jaffar. Minimal and complete word unification. In *Journal of the ACM 37(1), 47-85, 1990*.

[16] Y. Khmelevskii. Equation in free semigroups. In *Trudy Math. Inst. Steklov. 107 (1971); English Transl., Proc. Steklov Inst. Math. 107 (1971)*.

[17] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer and M. Ernst. HAMPI: a solver for string constraints. In *ISSTA'09*.

[18] G. Makanin. The problem of solvability of equations in a free semigroup. In *Mathematics of the USSR-Sbornik, 1977, 32, 129*.

[19] L. Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*.

[20] W. Plandowski. An efficient algorithm for solving word equations. In *STOC'06*.

[21] G. Redelinghuys, W. Visser and J. Geldenhuys. Symbolic execution of programs with strings. In *SAICSIT'12*.

[22] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant and D. Song. A Symbolic Execution Framework for JavaScript. In *SP'10*.

[23] K. Schulz. Word unification and transformation of generalized equations. In *J. Autom. Reason. 11(2):149-184, 1993*.

[24] D. Shannon, I. Ghosh, S. Rajan and S. Khurshid. Efficient symbolic execution of strings for validating web applications In *DEFECTS'09*.

[25] F. Sun, L. Xu and Z. Su. Static detection of access control vulnerabilities in web applications. In *USENIX Security'11*.

[26] T. Tateishi, M. Pistoia and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA'11*.

[27] M. Veanes, P. Halleux and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*.

[28] M. Veanes, N. Bjørner and L. Moura. Symbolic automata constraint solving. In *LPAR-17*.

[29] M. Veanes and N. Bjørner. Symbolic Automata: The Toolkit. In *TACAS'12*.

[30] F. Yu, T. Bultan and O. Ibarra. Symbolic String Verification: Combining String Analysis and Size Analysis In *TACAS'09*.

[31] F. Yu, M. Alkhalaf and T. Bultan. Stranger: An Automata-based String Analysis Tool for PHP. In *TACAS'10*.

[32] Y. Zheng and X. Zhang. Static Detection of Resource Contention Problems in Server-Side Scripts. In *ICSE'12*.

[33] Y. Zheng and X. Zhang. Path Sensitive Static Analysis of Web Applications for Remote Code Execution Vulnerability Detection. In *ICSE'13*.