S3: A Symbolic String Solver for Vulnerability Detection in Web Applications

Minh-Thai Trinh
trinhmt@comp.nus.edu.sg

Duc-Hiep Chu
hiepcd@comp.nus.edu.sg

Joxan Jaffar
joxan@comp.nus.edu.sg

National University of Singapore

ABSTRACT

Motivated by the vulnerability analysis of web programs which work on string inputs, we present S3, a new symbolic string solver. Our solver employs a new algorithm for a constraint language that is expressive enough for widespread applicability. Specifically, our language covers all the main string operations, such as those in JavaScript. The algorithm first makes use of a symbolic representation so that membership in a set defined by a regular expression can be encoded as string equations. Secondly, there is a constraint-based generation of instances from these symbolic expressions so that the total number of instances can be limited. We evaluate S3 on a well-known set of practical benchmarks, demonstrating both its robustness (more definitive answers) and its efficiency (about 20 times faster) against the state-of-the-art.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Security, Reliability, Verification

Keywords

String Analysis; String Constraint Solver; Web Applications

1. INTRODUCTION

Web applications nowadays provide critical services over the Internet and frequently handle sensitive data. Unfortunately, the development is error prone, resulting in applications that are vulnerable to attacks by malicious users. The global accessibility of critical web applications make this an extremely serious problem.

According to the Open Web Application Security Project, or OWASP for short [26], the most serious web application vulnerabilities include: (#1) Injection flaws (such as SQL injection) and (#3) Cross Site Scripting (XSS) flaws. These two vulnerabilities occur mainly due to inadequate sanitization and inappropriate use of input strings provided by users.

How Important is Symbolic String Solving?

To explain why we need string solving, let us look at *dynamic analysis* which involves testing an application as a closed entity with a set of concrete inputs. Its main disadvantage is of course that it is not a complete method. For example, some program paths may only be executed if certain inputs are passed as parameters to the application, but it is very unlikely that a dynamic analyzer can exhaustively test an application with all possible inputs. For web applications, the problem is even more severe since dynamic analysis needs to take into account not only the value space (i.e., how the execution of control flow paths depends on input values), but also an application's event space (i.e., the possible sequences of user-interface actions). As a result, there is in general an impractical number of execution paths to systematically explore, leading to the "low code coverage" issue of dynamic analysis.

A standard approach to have good or complete coverage is static analysis. However, the problem here is the existence of false positives, arising from an over-approximation of the program's behavior. Recent works to avoid false positives, but still preserve high code coverage, are based on *dynamic symbolic execution* (DSE). Some examples are [28, 4, 5, 23, 12, 30, 15, 16, 17, 14, 29, 9, 8, 24, 35, 21]. These approaches employ both concrete and symbolic execution to automatically and systematically generate tests in order to expose vulnerabilities in web applications. DSE for automated test generation involves instrumenting and *concolically* running a program while collecting path constraints on the inputs. Then it attempts to derive new inputs – using an SMT (Satisfiability Modulo Theories) solver – with the hope to steer next executions toward new program paths. For vulnerability detection, DSE combines the derived path constraints with the specifications for attacks, often given by the security experts, to create queries for the SMT solver.

	Dynamic Analysis	DSE
Code Coverage	Potentially Low	High
False Positives	Low	Low
Executable Paths (EPs)	Unlikely to cover all EPs	Likely to cover all EPs

Table 1: DSE as a More Effective Paradigm

In fact, there is a strong connection between an effective vulnerability detection framework and symbolic string solving. As shown in Table 1, DSE achieves higher code coverage. However, because not all path executed by DSE are guaranteed to be executable, to avoid false positives we must be able to *decide* if a (symbolic) path constraint is satisfiable or not. Thus a powerful SMT solver, capable of handling symbolic string variables, is the *key* to achieve efficient analyses with high code coverage and low false positives.

```
...
<html>
...
<script>
function validateEmail(form) {
  var email = form["email"].value;
  var index = email.indexOf("@");
  var local = email.substr(0, index);
  var domain = email.substr(index+1);

  if (domain.equals("nus.edu.sg")){
    var re = new RegExp("^[a-zA-Z][0-9]*$");
    var test1 = re.test(local);
    var test2 = local.length == 8;
    return test1 && test2;
  }
  else if (domain.equals("comp.nus.edu.sg"))
    return local.length >= 4;
  else
    return false;
}
</script>
...
<form name="loginForm" action="/Login"
    onsubmit="return validateEmail(this);">
  Email: <input type="text" name="email"
      size="64" />
  <input type="submit" value="Login" />
</form>
...
</html>
```

Figure 1: An Example of Email Address Validation

To illustrate more clearly how constraint solvers can be helpful in securing web applications, in Fig. 1, we present a JavaScript function which is used to validate input email addresses. The user fills the client-side form, by providing an email address to the HTML input element with name `"email"` (and a password, removed for simplicity). When the `Login` button is clicked, the browser invokes the JavaScript validating function `validateEmail`, which is assigned to the `submit` event of the form. This function first fetches the email address supplied by the user from the corresponding `form` field and then checks if the `email` address is valid. Each student of our department has two email accounts, one from NUS (`nus.edu.sg`), the other from SoC (`comp.nus.edu.sg`). The web page hence accepts both of these two domains. However, these two types of accounts have different formats. While the `local` part of the former is constructed by one alphabetic characters, followed by seven numeric ones, the latter's simply requires at least four characters.

The question is whether this web page is vulnerable to an XSS attack, or to an SQL injection. More specifically, can the following PHP code, with an appropriate instantiation for string variable `$eml`, be executed on the server side, leading to an attack:

```
$eml = $_POST['email'];
$pwd = $_POST['password'];
$stm="SELECT ... where email='$eml' and password='$pwd'";
$result = mysql_query($stm);
```

The answer is *yes* for both of the questions. Now, let us explain the way DSE detects possible vulnerabilities, in comparison with typical dynamic analyses. Since a dynamic analysis is essentially black-box testing, it has no knowledge about the JavaScript code. Thus, it is possible that the dynamic analyzer does not test with email addresses whose domain is `comp.nus.edu.sg`, and subsequently, cannot detect SQL injection and XSS vulnerabilities. In contrast, DSE, which can be seen as white-box testing, enables us to attempt all execution paths by generating three path constraints, corresponding to the three program paths of the `validateEmail` function.

After symbolically executing the program, DSE frameworks such as [28] will combine its results with the specifications for attacks, given by the security experts, to create queries for the constraint solver. The specifications, often come in form of assertions, are some (regular) grammars encoding a set of strings that would constitute an attack against a particular sink. If the constraint solver finds a solution to a query, then this represents an attack that can reach the critical sink and exploit a code injection vulnerability. For example, with the specification to assert if the input email address contains `' OR 1=1--`, we can in fact generate the input

```
' OR 1=1--@comp.nus.edu.sg
```

that leads to an SQL injection. Similarly, a specification for an XSS attack `<script>alert('Test')</script>` would help us to generate the input email address

```
<script>alert('Test')</script>@comp.nus.edu.sg
```

that can be exploited by attackers.

In summary for this subsection, DSE, presently the state-of-the-art in vulnerability detection, is intimately tied to being able to provide definitive answers for the derived constraint queries. In the case of JavaScript and web applications, since the constraints often concern string variables, symbolic string solving is thus the key to detect vulnerabilities in this class of applications. As the encountered string constraints may be in an undecidable class, it is important to have a solver which returns a definitive answer *often* and in a *timely* manner.

We next describe the main contribution of this paper, a new constraint solver S3, which stands for **S**ymbolic **S**tring **S**olver. Our solver makes use of Z3 [11], in order to leverage the recent advances in modern SMT solvers.

What Language Do We Need?

We first argue that a pure string language does not suffice to analyze web applications. This is due to the fact that non-string operations (e.g., boolean, arithmetic constraints) are also widely used in web applications. Moreover, their use is often intertwined with string operations, such as in the case of string length — a string-to-integer constraint. Reasoning about strings and non-strings *simultaneously* is thus necessary. In other words, we need to deal with a *multi-sorted* theory which includes, at least, strings and integers.

To amplify this point, let us now state some statistics from a comprehensive study of practical JavaScript applications [28]. Constraints arising from the applications have an average (per benchmark query) of 63 JavaScript string operations, while the remaining are boolean, logical and arithmetic constraints. The largest fraction are for operations like `indexOf`, `length` (78%). A significant fraction of the operations, including `substring` (5%), `replace` (8%), and `split`, `match` (1%). Of the `match`, `split` and `replace` operations, 31% are based on regular expressions. Operations such as `replace` and `split` give rise to new strings from the original ones, thereby giving rise to constraints involving *multiple* string variables.

To summarize, constraints of interest are either non-strings (e.g., bool-sort, int-sort and particularly length constraints) or strings such as: string equations, membership predicates and high-level string operations, which are over multiple string variables. It is folklore that query with just basic string equations along with length constraints on the string variables is extremely hard to solve (its decidability is open). Therefore, the validation of any approach can only realistically be done *empirically*.

S3: A Robust and Incremental String Solver

Although there exist solvers that can reason about both string and non-string constraints (e.g., [28, 6, 27, 39]), they depend on strings being *bounded* in length. Unbounded regular expressions, which can be constructed using Kleene star operation, are not supported. Thus the supported high-level operations are only in bounded forms. For example, instead of fully supporting `replace` function, which could mean replacement of all occurrences, existing tools support an operation to replace a fixed number of occurrences in a string.

It may be argued that certain bounds suffice for a class of applications. There is a *more important* reason why the bound dependency is bad: the algorithms that rely on the bounded reasoning are highly *combinatorial* in approach. In other words, the problem at hand is broken down into cases, the number of which is often a large combinatorial combination arising from some given bounds.

Finally, we mention [1], where there is a real requirement for reasoning about unbounded strings. In verifying client-side input validation functions, a bounded string solver can only find policy violations but it cannot prove the conformance to a given policy. There are certainly some solvers [19, 12, 36, 37] that can reason about unbounded strings. However, their key weakness is that they cannot handle non-string constraints, particularly length constraints. As shown in the statistics above, missing length constraints (whose appearance is frequent) will lead to many false positives. This clearly is not acceptable.

With regard to all the arguments above, we now conclude this Section with three important features of S3.

First, S3 is *expressive* (Section 2). Specifically, it is the first to handle unbounded regular expressions in the presence of length constraints, and express precisely high-level string operations, which ultimately enables a more accurate string analysis.

Second, S3 is *robust*. This means that S3 is able to provide definitive answers to a new level, far beyond the state-of-the-art. This in turn means we can detect more vulnerabilities and more bugs. We demonstrate in Section 6 with two case studies:

- The first compares with Kaluza – the core of Kudzu [28] – a JavaScript symbolic execution framework. We show that S3 is several times faster, and helps detect many more paths that reach the critical sink, that is, paths that are vulnerable.
- The second compares with Z3-str [39]. We show S3 reasons about length constraints much more effectively than Z3-str. This leads to a large increase in applicability to web programs, because these kind of constraints are widely used.

Third, S3 is *efficient*, and one key reason is that it is *incremental*. Our algorithm for string theory is designed in an incremental fashion driven by the try-and-backtrack procedure of the Z3 core (Section 4), so that given a set of input constraints, we perform incremental reduction for string variables until the variables are bounded with constant strings/characters. Another technical challenge is how to reason, effectively and efficiently, about the Kleene star and high-level operations such as `replace` (in its most general usage), of which the semantics are by nature recursively defined. Section 5 introduces the gist of our proposal, the encodings using recursively-defined functions, on which we can incrementally reason: by lazily *unfolding* them.

2. OUR CONSTRAINT LANGUAGE

We introduce the constraint language of our solver in Fig. 2. For simplicity, we only list three primitive types: int, bool and string[1]. The input formula can be of the following forms:

[1]Z3 supports more primitive types [11].

Assertion	::=	**assert** (*(Fml:bool)*)
Fml:bool	::=	*(Term:bool)*
	|	*(Term:bool)* = *(Term:bool)*
	|	*(Term:int)* $\{<, \leq, =, \geq, >\}$ *(Term:int)*
	|	*(Term:str)* = *(Term:str)*
	|	*(Term:str)* $\in$ *(Term:regexpr)*
	|	$\neg$ *(Fml:bool)*
	|	*(Fml:bool)* $\{\wedge, \vee, \Rightarrow\}$ *(Fml:bool)*
Term:bool	::=	*(Var:bool)*
	|	**true**
	|	**false**
	|	**contains**(*(Term:str), (Term:str)*)
Term:int	::=	*(Var:int)*
	|	*Number*
	|	*(Term:int)* $\{+, -, \times, \div\}$ *(Term:int)*
	|	**length**(*(Term:str)*)
	|	**indexOf**(*(Term:str),(Term:str)*)
	|	**search**(*(Term:str), (Term:regexpr)*)
	|	**test**(*(Term:regexpr), (Term:str)*)
Term:str	::=	*ConstString*
	|	*(Var:str)*
	|	*(Term:str)* $\cdot$ *(Term:str)*
	|	**concat**(*(Term:str),(Term:str)*)
	|	**substring**(*(T:str), (T:int), (T:int)*)
	|	**replaceN**(*(T:str),(T:regexpr),(T:str),(T:int)*)
	|	**replaceAll**(*(T:str), (T:regexpr), (T:str)*)
L:str list	::=	**match**(*(Term:str), (Term:regexpr)*)
	|	**split**(*(Term:str), (Term:regexpr)*)
	|	**exec**(*(Term:regexpr), (Term:str)*)
Term:regexpr	::=	*ConstString*
	|	*(Term:regexpr)**
	|	*(Term:regexpr)* $\cdot$ *(Term:regexpr)*
	|	*(Term:regexpr)* + *(Term:regexpr)*

Figure 2: The Grammar of Our Input Constraint Language

- a boolean expression;
- a comparison operation between two integer or boolean expressions;
- an equation between two string expressions. S3 also supports other common string operations. We list here only important ones;
- a membership predicate between a string expression and a regular expression, where an expression can either be a string constant, a variable or their concatenation[2], and regular expressions are constructed from string constants using concatenation ($\cdot$), union ($+$) and Kleene star (*);
- a composite formula constructed using negation and binary connectives, including $\wedge$, $\vee$, $\Rightarrow$.

Z3-str [39] and Kaluza [28] are important existing solvers that can support both string and non-string operations, especially the length constraint. Compared to the constraint syntax of Z3-str, ours can be viewed as an extension with regular expressions, membership predicates, and high-level string operations that often work on regular expressions such as **search**, **replaceAll**[3], **match**, **split**, **test**, **exec**. Our constraint language is also slightly more expressive than Kaluza's since we handle above string operations in its original semantics — unbounded.

[2]We use $x \cdot y$ as a shorter form for **concat**(x, y).

[3]This operation is used to replace all occurrences.

A JavaScript Program

```
function validateFields(p1,p2) {
  var re1 = /^(ab)*$/;
  var re2 = /^(bc)*$/;
  var t1 = re1.test(p1);
  var t2 = re2.test(p2);
  var t3 = p2.length > 0;
  return (t1 && t2 && t3)
}
```

Generated Constraints

```
p1 ∈ ("ab")⋆ ∧
p2 ∈ ("bc")⋆ ∧
length(p2) > 0 ∧

res = p1 · p2 ∧
nM = "ababababababcc" ∧
res = nM
```

Our Internal Representation

```
p1 = star("ab",n1) ∧
p2 = star("bc",n2) ∧
length(p2) > 0 ∧

res = p1 · p2 ∧
nM = "ababababababcc" ∧
res = nM
```

Figure 3: From a JavaScript Program to the Generated Constraints

In addition, we note that our constraint language, which is necessary to reason about high-level string operations in scripting languages, is beyond the class of context free languages. To illustrate, let us look at the following constraints, in which x can be of any string in the context-sensitive language $\{\ a^n{\cdot}b^n{\cdot}c^n \mid n{\geq}0\ \}$:

$$x = y \cdot z \cdot t \wedge y \in a^\star \wedge z \in b^\star \wedge t \in c^\star \wedge \\ \mathbf{length}(y){=}\mathbf{length}(z) \wedge \mathbf{length}(z){=}\mathbf{length}(t)$$

Therefore, existing solvers, which only approximate strings using context free grammars, are not able to reason about the constraints addressed by this paper.

Finally, though it is not shown in Fig. 2, S3 is able to accommodate most regular expression features in JavaScript via a preprocessing step as done in Kudzu [28]. Examples are (possibly negated) character classes, escaped sequences, repetition operators ({n}/?/⋆/+/) and sub-match extraction using capturing parentheses.

3. MOTIVATING EXAMPLES

In this Section, we present two simplified examples to position our work against the state-of-the-art.

In Fig. 3 we start with an example of a regular-expression-based input validation function. The first column is the JavaScript function used to validate the two input fields, namely `p1` and `p2`. This function ensures that `p2` is not an empty string and `p1` and `p2` must belong to the regular expressions `re1` and `re2`, respectively.

Now we want to prove that, given the inputs which have passed the validation function, the output `res`, that is constructed by concatenating `p1` with `p2`, is different from a specified bad string `nM` = "ababababababcc". Ultimately, the above question is reduced to the problem of deciding the satisfiability of the generated constraint formula, presented in the second column of Fig. 3. The proof succeeds if the formula is unsatisfiable[4].

This requires reasoning about string equation `res=p1·p2`, membership predicates `p1` $\in$ ("*ab*")* and `p2` $\in$ ("*bc*")*, and length constraint **length**(`p2`) > 0. In short, it becomes a complicated problem involving strings, non-strings and their combinations (e.g., length constraints). Now, let us discuss how existing solvers would deal with this particular problem.

HAMPI [22], and other solvers [10, 31, 20, 34, 2, 38, 19, 32, 13], which work in the string domain only, cannot handle this example. Since they only support string operations, they are not able to handle non-string constraints, and particularly length constraints that are related to both string and non-string domain and cannot be captured in each individual one.

On the other hand, the solvers Kaluza [28], [6] and Z3-str [39] are in the same category as ours, and can reason about strings and non-strings simultaneously. Since [6] is similar to Kaluza in many ways, we will just focus on Kaluza here. Kaluza is the string solver used in a JavaScript dynamic test generation framework [28]. To support a wider range of constraint types including integer, boolean and string, it extends both STP [22] and HAMPI.

One major drawback of Kaluza is that it requires the lengths of string variables to be known prior to being able to encode them and query the underlying SMT solvers. In particular, before solving for string constraints, Kaluza finds a set of satisfying solutions for each string length. For each possible length, it encodes each string variable as an array of bits and then queries a bit-vector solver. Kaluza is unable to reuse the encodings and the result of bit-vector solver in previous calls, which induces the overall high cost of repetitive encoding and querying external solvers.

For the example at hand in Fig. 3, Kaluza first needs to come up with a set of satisfying solutions for the lengths of `p1` and `p2`, each denoted by a pair $\langle l_1; l_2\rangle$, where l_1 is the length of `p1` and l_2 is the length of `p2`. In this case, the set of satisfying solutions for the lengths is $\{\langle 0; 14\rangle, \langle 2; 12\rangle, \langle 4; 10\rangle, \langle 6; 8\rangle, \langle 8; 6\rangle, \langle 10; 4\rangle, \langle 12; 2\rangle\}$. For each possible length solution, Kaluza encodes the string variables, and then queries the external bit-vector solver, before finding out that the original set of constraints is unsatisfiable. Overall, Kaluza needs to encode and query bit-vector solver 7 times.

Let us not have the impression that, in general, the number of satisfying solutions for the string lengths should be of this linear complexity. In fact, practical applications involve many string variables, Kaluza approach, i.e., generate-and-test, would easily suffer from a *combinatorial* explosion.

Z3-str [39] cannot handle regular expressions, thus also cannot handle this example. However, it can be considered the first SMT-based string solver. Instead of relying on other theories, it builds a string theory for itself and allows this string theory to be plugged into a modern and powerful solver – Z3 [11]. Thus an important contribution of Z3-str is that string and non-string constraints are now solved simultaneously, in an *incremental* manner.

Inspired by Z3-str's design, our target is to build a string theory that can interact with other theories via Z3. Nevertheless, we want to support a powerful input language, which is especially demanded for testing and analysis of practical web applications.

There are two key technical challenges: (1) how to *incrementally* handle the Kleene star, which is the heart of the issue in reasoning about regular expressions; (2) how to *incrementally* handle high-level string operation such as `replace`, whose semantics is most naturally defined by recursive rules. Our solution therefore is to employ, in our string theory, recursively defined functions whose semantics will be *lazily* unfolded during the process of incremental solving. Such approach resembles the constrain-and-generate technique (to contrast with generate-and-test) in the literature of constraint solving.

We elaborate later with a technical description in Section 5. But now let us give some intuitions on how we approach this example. Internally, we represent membership of regular expression as equation involving a *symbolic representation* of the Kleene star. In

[4]Otherwise, the solver should return satisfying assignments, representing a potential bug/vulnerability of the system.

particular, p1∈(“*ab*”)* is represented as p1=**star**(“*ab*”,n1) and similarly p2∈(“*bc*”)* is represented as p2=**star**(“*bc*”,n2). By rewriting, we would derive the following equation:

$$\boxed{\textbf{star}(\text{“}ab\text{”},\texttt{n1}) \cdot \textbf{star}(\text{“}bc\text{”},\texttt{n2}) = \text{“}ababababababcc\text{”}}$$

Since the length of p2 is positive and the RHS is a constant string, this would force the *unfolding* of expression **star**(“*bc*”,n2) to **star**(“*bc*”,n2-1) · “*bc*”. A conflict is then derived since the LHS string ends with “*bc*” while the RHS string ends with “*cc*”. Our system then can conclude that the input formula is UNSAT.

$$x = x_1 \cdot x_2 \ \wedge \ z = y \cdot z_3 \ \wedge \ y = z_1 \cdot z_2 \ \wedge \ z_2 = \text{“\_”} \ \wedge$$
$$l_1 = \textbf{length}(x_1) \ \wedge \ l_2 = \textbf{length}(z_1) \ \wedge \ l_1 = l_2 + 1 \ \wedge$$
$$x = z \ \wedge \ \textbf{indexOf}(y, \text{“}a\text{”}) = 3 \ \wedge \ \textbf{indexOf}(x1, \text{“}a\text{”}) = 4$$

Figure 4: A Frequent Constraint Pattern

Now let us dissect Z3-str more carefully. Fig. 4 presents an input example for Z3-str, a pattern which is commonly found in many benchmarks extracted from [28]'s comprehensive set of JavaScript applications (e.g. big2). Starting with the fact that z_2 is a constant string of one character, Z3-str is able to deduce that z_2 is of length 1. This constraint will be fed into the arithmetic theory. Similarly, the arithmetic theory would receive the information that y's length is the sum of z_1's length and z_2's length. Since, from the input, the length of x_1 equals to the length of z_1 plus 1, the arithmetic theory can deduce that x_1 and y are of the same length. However, this information will *never* be passed back to the string theory.

As discussed in [39], the current design of Z3 enforces that the plug-in theory, namely Z3-str, to be *disjoint* from Z3's arithmetic theory. Being a plug-in, however, means there is supervisory control over Z3-str which can feed length information to the arithmetic theory so that early conflicts can be detected and exploited. But, importantly, partial information derived by the arithmetic theory will not be fed back to Z3-str. This is the source of Z3-str's inefficiency in many cases.

Returning to the example, if the information that x_1 and y are of the same length is propagated back to the string theory, together with the fact that x_1 and y are prefixes of the two equal strings x and z, our string theory can derive that x_1 and y are equal, therefore proceed the search much more efficiently. In Section 4.2, we discuss our new design in order to overcome this drawback, therefore even when restricted to the same input language as of Z3-str, our tool, S3, does advance the concept of incremental solving to the next level.

4. DESIGN OF S3

Here we present the design of S3. This design is inspired by Z3-str [39], and thus inherits its two main advantages. First, we support the primitive type of string so that there is no need to convert strings to other representations, e.g., bit-vectors. As a result, we can support string variables whose lengths can be *unknown*, especially in the context of static analysis. Second, we leverage the power of Z3 in dealing with multiple theories, and this ultimately leads to the capability of reasoning on string and non-string constraints simultaneously and efficiently. We first give an overview of Z3-str, focusing on how it interacts with the core of Z3. Later we describe our design of S3, along with the improvement of the corresponding component Z3-str-star over Z3-str.

4.1 Overview of Z3-str

Z3-str acts as a plug-in string theory for a SMT solver Z3 [11]. The architecture of Z3 is shown in the shaded box of Fig. 5. Its

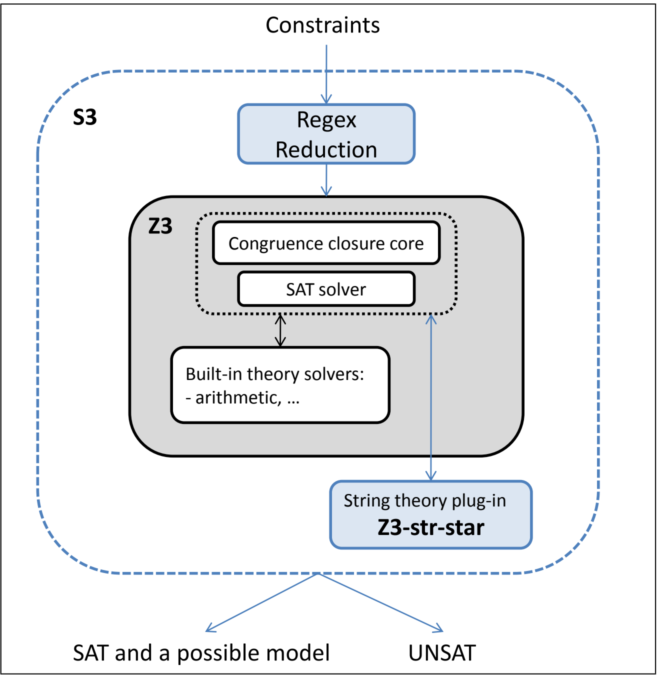

Figure 5: The Design of S3

core component consists of the following modules: the congruence closure engine, a SAT solver-based DPLL layer, and several built-in theory solvers, such as integer linear arithmetic, bit-vectors, etc. The congruence closure engine can detect equivalent terms and then classify them into different equivalence classes, which are shared among all built-in theory solvers. The SAT-based DPLL layer is responsible for handling the boolean structure of the input formula.

assert $((e_1 \vee e_2) \wedge e_3 \wedge e_4)$	
$e_1 : x = \text{“}abc\text{”} \cdot m$	$e_2 : x = \text{“}efgh\text{”}$
$e_3 : y = \text{“}efg\text{”} \cdot n$	$e_4 : x = y$

Consider the assertion above. The core component cannot interpret the string operations; instead it treats them as four independent boolean variables (e_1, e_2, e_3 and e_4) and tries to assign boolean values to them. We now walk through the process of how Z3's core component and the string theory solver interact.

	Fact added	Eq-class	Reduction/Action
1	y=“efg”·n	{y,“efg”·n}	
2	x=y	{x,y} {y,“efg”·n}	
3	x=“abc”·m	{x,“abc”·m, y,“efg”·n}	• conflict detected • backtrack and remove facts • try another option for e_1
4	x=“$efgh$”	{x,“$efgh$”, y,“efg”·n}	“$efgh$”=“efg”·$n \Rightarrow n$=“h”
SAT solution: x = “$efgh$”, y = “$efgh$”, n = “h”			

Table 2: How Z3-str Interacts with Z3 and Its Backtracking

In Table 2, initially there is no fact. The core starts by setting e_3 and e_4 to true and reaches step 3. Without loss of generality, assume the core component first tries true for e_1. Beware that the core can detect functionally equivalent terms, based on the theory of uninterpreted functions. Hence, it puts $\{x, y, \text{“}abc\text{”} \cdot m, \text{“}efg\text{”} \cdot n\}$ into one equivalence class and notifies the string theory plug-in. We note that the plug-in string theory Z3-str can only know about the equivalent terms that belong to its theory.

As a side remark, if we have an equation $\mathbf{length}(x) = 4$, then Z3-str is not aware of the fact that $\mathbf{length}(x)$ is equal to 4. However, if e_2 were set to `true`, Z3-str would know that x is equivalent to a constant string of length 4. Therefore, it can deduce that $\mathbf{length}(x)$ is equal to 4, thus subsequently passing this information to the arithmetic theory.

Back to the example, with the above equivalence class at step 3, Z3-str detects a conflict and then informs the core component about the new finding through an axiom $e_3 \wedge e_4 \rightarrow \neg e_1$. With this new axiom, the core component backtracks and tries `false` for e_1. When the core component backtracks, it discards the relevant fact and any insertions into equivalence classes as the consequence of the fact. The core then derives that e_2 must be `true` and this assignment is performed in step 4.

Based on the concatenation semantics, Z3-str can infer that n must be “h”. This new finding is formulated by introducing a new boolean variable e_5 representing $n = \text{“}h\text{”}$ and an axiom $\text{“}efgh\text{”} = \text{“}efg\text{”} \cdot n \Rightarrow e_5$, which is sent back to the core. From the existing facts and the new axiom, the core component derives e_5 is `true`. After all boolean expressions have been assigned consistently and Z3-str can find the satisfying values for string variables x, y, and n, the search procedure terminates.

4.2 Improvement of Z3-str-$\star$ over Z3-str

Z3-str-star (or Z3-str-$\star$ for short), a component of our tool, is responsible for solving equations between string expression and recursively-defined functions. It can be viewed as a *significant extension* of Z3-str with the support of recursively-defined functions, introduced to facilitate representing and reasoning about the Kleene star and commonly used high-level string operations.

As mentioned before, in its current implementation, Z3-str does not know about equivalent terms that belong to other theories, especially the arithmetic theory. Another important improvement of Z3-str-$\star$ (over Z3-str) is its direct interactions with the Z3 core, to query about the equivalence classes among multiple theories. More specifically, it asks Z3 core two following questions:

- Is a string length “ground" with a non-negative constant?
- What is the relationship ($=, <, >, \leq, \geq$) between different length variables?

To answer these questions, we extend Z3 API so that Z3-str-$\star$ can interact with the congruence closure core, similarly to other built-in theory solvers. Moreover, the newly introduced API methods also help us to query about other inequality relationship, if necessary. Answers to these questions ultimately allow us to propagate the information of string lengths to string theory solver so that string and non-string constraints can be simultaneously reasoned about. In short, this gives us a truly incremental solver for strings and non-strings. We will revisit this side contribution in our experimental evaluation – Section 6.

5. ALGORITHM

5.1 Top-level Algorithm

S3 finds a list of string assignments that satisfies the input formula or decides that no satisfying assignment exists. Algorithm 1 summarizes its top level algorithm.

```
Input: F : Formula
Output: (IsSat : bool, Solutions : (variable, string) list)
reduced_F ← reduce(F);
$\bigvee_i^n$ disjunct$_i$ ← normalize_to_DNF(reduced_F);
for $i = 1$ to $n$ do
    (Res, Sols) ← Z3-str-$\star$(disjunct$_i$);
    if Res = SAT then
        return (true, Sols);
    end
end
return (false, []);
```

Algorithm 1: Top-level Algorithm

Given an input formula `F`, S3 recursively reduces `F` into new formula `reduced_F`, which may contain equations (among string expressions and recursive functions such as **star**) and length constraints. Here we only take into consideration the string and length constraints, non-string constraints will be unchanged unless otherwise stated. Reduction rules may result in a disjunctive formula. Thus, the next step is to normalize `reduced_F` into disjunctive normal form (DNF). To decide the satisfiability of each disjunct, we extend Z3-str [39] to support recursive functions. In particular, we use the recursive function **star** to represent the Kleene star. For presentation purpose, we first only discuss how to handle the **star** function, calling our extended component Z3-str-$\star$. Similar treatment for high-level operations such as **replaceAll** will be elaborated later. If Z3-str-$\star$ finds a satisfiable disjunct, it stops and returns the corresponding satisfying assignments. Otherwise, it decides that no such assignment exists.

5.2 Reduction of Regular Expressions

Rule	Reduction
[CONST]	$e \in s \quad \rightarrow \quad e{=}s$
[UNION]	$e \in r_1{+}r_2 \quad \rightarrow \quad e \in r_1 \vee e \in r_2$
[CONCAT]	$e \in r_1{\cdot}r_2 \quad \rightarrow \quad e{=}e_1{\cdot}e_2 \wedge \bigwedge_{i=1}^{2} e_i \in r_i$
[STAR]	$e \in r^\star \quad \rightarrow \quad e \overset{\vee}{=} \mathbf{star}(r,\ n)$

Table 3: Reduction Rules

Given an input constraint formula, we first reduce membership predicates into equations among string expressions and **star** function. The reduction rules are summarized in Table 3. Our aim is to obtain a list of new constraints of the form that can be solved *incrementally* by Z3-str-$\star$ — equations among string expressions and recursively-defined functions, along with length constraints.

These rules deal with constraints checking if a string expression e (LHS) is in a regular expression (RHS). If the RHS is merely a string constant, rule [CONST] will convert such membership constraint into an equality. The next two rules handle the case when the RHS is constructed by union and concatenation operations. While rule [UNION] ensures that the LHS expression e is a member of one of the RHS sub-expressions (of the union), rule [CONCAT] splits e into two fresh string variables, namely e_1 and e_2, and checks that they satisfy the condition $e_1 \in r_1 \wedge e_2 \in r_2$ conjunctively.

The RHS regular expression can also be formed by repeating r zero or more times (Kleene star). Rule [STAR] encodes such constraint as an equation, where the LHS is a string expression and the RHS is a symbolic representation for a family of strings generated by the Kleene star. The fresh (symbolic) integer variable n indicates the frequency where r is repeated. This symbolic variable is used to:

- Distinguish different **star** functions, which have the same base regular expression (e.g. r).
- Guide the on-demand unfolding in the recursively-defined functions such as **star** or **replaceAll** (that will be discussed later).
- Interact with the Arithmetic Solver module in Z3.

When r is a constant string and n is a concrete value, the $\overset{\vee}{=}$ operator is interpreted as equality operator $=$. For convenience, we overload $\overset{\vee}{=}$ with the $=$ notation.

In short, after the reduction for regular expressions, we have equations among string expressions and recursively-defined **star** functions, along with length constraints. Z3-str-$\star$ is then responsible for solving them.

5.3 star Functions

Z3-str-$\star$ extends Z3-str [39] with the support for handling **star** functions. The internal language is extended with the following:

$$\begin{array}{rcl} \textit{Term:str} & ::= & \textit{ConstString} \\ & | & \ldots \\ & | & \mathbf{star}(\textit{Term:regexpr}, \textit{Term:int}) \end{array}$$

Like Z3-str, Z3-str-$\star$ also works as a plug-in of Z3. It is notified by the Z3 core component when a string equation is asserted as part of the try-and-backtrack process. In particular, the core component invokes a callback function in the plug-in, providing the abstract syntax tree (AST) of the equation as an input parameter. The callback function inspects the AST, and if it involves string operations, the function tries to reduce AST to a simpler abstract syntax tree, say AST'. The reduction is conveyed to the core component by adding an axiom with the form of $AST \Rightarrow AST'$. Recall that since the core component does not understand the string domain, it treats both AST and AST' as independent boolean variables. Because AST has been assigned a `true` value, with the new axiom, the core will assign `true` to AST' as well, which is a new fact, and in turn triggers further plug-in processing. Thus, to act as a plug-in, we need to provide reduction rules for each callback from Z3.

We list selected reduction rules in Table 4. There are 3 cases of interest related to **star** functions:

- when **star** appears in one side of an equation,
- when **star** appears in both side of an equation and
- when **star** can be used to concretize other concatenations based on its concrete string value.

The gist of our reduction rules is to make use of the semantics of **star** functions (or their previous forms – regular expressions with Kleene star). In fact, with a membership constraint such as $\mathtt{x} \in (\text{“}ab\text{”})^\star$, we can directly make use Z3-str to generate the possible string assignments for x, then checking membership is straightforward since x is already ground. However, this naive approach is likely to be inefficient. Sometimes, it may be worse than Kaluza's approach, where the lengths are used to refine the string constraints. To deal with **star** functions effectively and efficiently, we propose to reduce it *lazily* and only *on demand*. We call that technique *“unfold and consume"*. The basic principle is to lazily unfold its semantics, until we find a matching between constant string segments in the two sides of an equation. At that time, we can easily to choose either consume these constants (of course with the capability of backtracking), or to find a conflict between *unmatchable* constants in the two sides.

Incremental Solving for star Functions

In Table 4 we introduce four auxiliary functions: $csm\_hd(s, r)$, $csm\_tl(s, r)$, $\overset{r}{csm}\_hd(r_2, r_1)$, and $csm\_all(s, r)$. The first one takes a constant string and a regular expression, and returns a list of strings s_i such that: $s \in r{\cdot}s_i$. Intuitively, this function aims to consume the prefix of s matching r. Similarly, while the second, $csm\_tl(s, r)$, consumes the suffix of s matching r, the third one applies to two regular expressions instead. Lastly, $csm\_all(s, r)$ checks if s can be consumed completely by matching it with r.

Now, let us have a look at reductions rules in Table 4. The rule [CON$-\star$] says about the case when $\mathbf{star}(r, n)$ equals to some constant string s. As we explained above, method $csm\_all(s, r)$ is used to decide whether s can be a member of $r^\star$. If yes, we can update other string expressions that contain $\mathbf{star}(r, n)$. Otherwise, it is a conflict and Z3 core component will need to backtrack. Note that, in Table 4, all E_1, E_2 and E_3 are concatenations among string expressions and **star** functions.

The rules [HT$-\star$] and [HD$-\star$] are to handle the case when there is a matching between **star** and a constant string. In the latter, the matching is at the beginning of the LHS; while the former is a special case of it, where the matchings occur at both ends. These two rules will be elaborated more in the next example. Similarly, we have the rule [TL$-\star$] for the matching at the end of the LHS.

The rule [HD$-\star-\star$] ([TL$-\star-\star$], [HT$-\star-\star$]) is applied when there are two **star** function at the beginning (end or both) of each side of the equation. In the rule [HD$-\star-\star$], we assume that r_1 cannot be consumed by r_2 so that we only need the auxiliary function $\overset{r}{csm}\_hd(r_2, r_1)$.

The last rule [REP$-\star$] aims to replace all string variables by their aliases, which are a concatenation among constant strings and **star** functions.

To illustrate how these rules are applied, in Table 5, we present running steps for solving the example in Fig. 3. Z3 core continually sends the assignments to our Z3-str-$\star$ (via its call back function) from step 1 to step 5. At the same time, Z3 also maintains functionally equivalent terms in their equivalence classes. From step 1 to step 4, we apply the rule [REP$-\star$] repetitively to replace a string variable by a constant string, a **star** function or their concatenation (shown in column 4, step 1-4). In step 5, we apply a specialized version of rule [HT$-\star$], where we also make use of constraints on variable n_1 and n_2. More specifically, for this running example, we are able to force the unfolding of $\mathbf{star}(\text{“}bc\text{”}, \mathtt{n2})$ so that we can find a conflict between “*bc*” and “*cc*”. Finally, we give back the new axiom (in column 4, step 5) to Z3 so that Z3 can conclude the input formula is UNSAT.

5.4 String Operations

Operations	Reduction Rules
$\mathtt{I}{=}\mathbf{search}(\mathtt{S}, \mathtt{r})$	$(\mathtt{I}{<}0 \wedge \neg(\mathtt{S} \in (.^\star){\cdot}\mathtt{r}{\cdot}(.^\star))) \vee (\mathtt{I}{\geq}0 \wedge \mathtt{S}{=}\mathtt{U}{\cdot}\mathtt{M1}{\cdot}\mathtt{M2}{\cdot}\mathtt{R} \wedge \mathtt{M1}{\cdot}\mathtt{M2} \in \mathtt{r} \wedge \mathbf{length}(\mathtt{U}){=}\mathtt{I} \wedge \mathbf{length}(\mathtt{M2}){=}1 \wedge \neg(\mathtt{U}{\cdot}\mathtt{M1} \in (.^\star){\cdot}\mathtt{r}{\cdot}(.^\star)))$
$\mathtt{R}{=}\mathbf{replaceAll}(\mathtt{S}, \mathtt{r}, \mathtt{T})$	$\mathtt{I}{=}\mathbf{search}(\mathtt{S}, \mathtt{r}) \wedge ((\mathtt{I}{<}0 \wedge \mathtt{R}{=}\mathtt{S}) \vee (\mathtt{I}{\geq}0 \wedge \mathtt{S}{=}\mathtt{U}{\cdot}\mathtt{M}{\cdot}\mathtt{S}_1 \wedge \mathtt{R}{=}\mathtt{U}{\cdot}\mathtt{T}{\cdot}\mathtt{R}_1 \wedge \mathtt{M} \in \mathtt{r} \wedge \mathbf{length}(\mathtt{U}){=}\mathtt{I} \wedge \mathtt{R}_1{=}\mathbf{replaceAll}(\mathtt{S}_1, \mathtt{r}, \mathtt{T})))$

Table 7: Reduction Rules for search and replaceAll

Typically, the semantics of string operations such as **replaceAll**, **match**, **split**, **test**, **exec**, are recursively defined. As such, it is natural for us to interpret them as recursively-defined functions, similarly to our handling of **star** functions. In this Subsection, we only

Rule	Reduction			Condition
$[\textbf{CON}-\star]$	$\textbf{star}(r,n)=s$	$\Rightarrow$	$\neg\textbf{star}(r,n)=s$	$\neg csm\_all(s,r)$, s: *ConstString*
	$\textbf{star}(r,n)=s\wedge$ $(E_1\cdot\textbf{star}(r,n)\cdot E_2=E_3)$	$\Rightarrow$	$E_1\cdot s\cdot E_2=E_3$	$csm\_all(s,r)$, s: *ConstString*
$[\textbf{HT}-\star]$	$\textbf{star}(r_1,n_1)\cdot E_1\cdot\textbf{star}(r_2,n_2)$ $=s_1\cdot E_2\cdot s_2$	$\Rightarrow$	$(E_1=s_1\cdot E_2\cdot s_2\wedge n_1=0\wedge n_2=0)\vee$ $(\bigvee_{i=1}^{k}\textbf{star}(r_1,n_1-1)\cdot E_1=s_i\cdot E_2\cdot s_2\wedge n_2=0)\vee$ $(\bigvee_{j=1}^{l}E_1\cdot\textbf{star}(r_2,n_2-1)=s_1\cdot E_2\cdot s_j\wedge n_1=0)\vee$ $\bigvee_{i,j}^{k,l}\textbf{star}(r_1,n_1-1)\cdot E_1\cdot\textbf{star}(r_2,n_2-1)=s_i\cdot E_2\cdot s_j$	$[s_i]=csm\_hd(s_1,r_1)$, $[s_j]=csm\_tl(s_2,r_2)$
$[\textbf{HD}-\star]$	$\textbf{star}(r,n)\cdot E_1=s\cdot E_2$	$\Rightarrow$	$(E_1=s\cdot E_2\wedge n=0)\vee\ \bigvee_{i=1}^{k}\textbf{star}(r,n-1)\cdot E_1=s_i\cdot E_2$	$[s_i]=csm\_hd(s,r)$
$[\textbf{TL}-\star]$	$E_1\cdot\textbf{star}(r,n)=E_2\cdot s$	$\Rightarrow$	$(E_1=E_2\cdot s\wedge n=0)\vee\ \bigvee_{i=1}^{k}E_1\cdot\textbf{star}(r,n-1)=E_2\cdot s_i$	$[s_i]=csm\_tl(s,r)$
$[\textbf{HT}-\star-\star]$	$\textbf{star}(r_1,n_1)\cdot E_1\cdot\textbf{star}(r_3,n_3)=$ $\textbf{star}(r_2,n_2)\cdot E_2\cdot\textbf{star}(r_4,n_4)$	$\Rightarrow$	$(n_2=0\wedge n_4=0\wedge\textbf{star}(r_1,n_1)\cdot E_1\cdot\textbf{star}(r_3,n_3)=E_2)\vee$ $(n_2=0\wedge\textbf{star}(r_1,n_1)\cdot E_1\cdot\textbf{star}(r_3,n_3)=E_2\cdot\textbf{star}(r_4,n_4))\vee$ $(n_4=0\wedge\textbf{star}(r_1,n_1)\cdot E_1\cdot\textbf{star}(r_3,n_3)=\textbf{star}(r_2,n_2)\cdot E_2)\vee$ $\bigvee_{i,j}^{k,l}\textbf{star}(r_1,n_1-1)\cdot E_1\cdot\textbf{star}(r_3,n_3-1)=$ $s_i\cdot\textbf{star}(r_2,n_2-1)\cdot E_2\cdot\textbf{star}(r_4,n_4-1)\cdot s_j$	$[s_i]=\overset{r}{csm}\_hd(r_3,r_1)$, $[s_j]=\overset{r}{csm}\_tl(r_4,r_2)$
$[\textbf{HD}-\star-\star]$	$\textbf{star}(r_1,n_1)\cdot E_1=$ $\textbf{star}(r_2,n_2)\cdot E_2$	$\Rightarrow$	$(E_1=E_2\wedge n_1=0\wedge n_2=0)\vee\ (\textbf{star}(r_1,n_1)\cdot E_1=E_2\wedge n_2=0)\vee$ $(E_1=\textbf{star}(r_2,n_2)\cdot E_2\wedge n_1=0)\vee$ $\bigvee_{i=1}^{k}\textbf{star}(r_1,n_1-1)\cdot E_1=s_i\cdot\textbf{star}(r_2,n_2-1)\cdot E_2$	$[s_i]=\overset{r}{csm}\_hd(r_2,r_1)$
$[\textbf{TL}-\star-\star]$	$E_1\cdot\textbf{star}(r_1,n_1)=$ $E_2\cdot\textbf{star}(r_2,n_2)$	$\Rightarrow$	$(E_1=E_2\wedge n_1=0\wedge n_2=0)\vee\ (E_1\cdot\textbf{star}(r_1,n_1)=E_2\wedge n_2=0)\vee$ $(E_1=E_2\cdot\textbf{star}(r_2,n_2)\wedge n_1=0)\vee$ $\bigvee_{i=1}^{k}E_1\cdot\textbf{star}(r_1,n_1-1)=E_2\cdot\textbf{star}(r_2,n_2-1)\cdot s_i$	$[s_i]=\overset{r}{csm}\_tl(r_2,r_1)$
$[\textbf{REP}-\star]$	$x=E\wedge\ (E_1\cdot x\cdot E_2)$	$\Rightarrow$	$E_1\cdot E\cdot E_2$	E is a concatenation among constant strings and **star** functions

Table 4: Selected Reduction Rules for star Functions

give the details of reduction for **replaceAll**. Other operations can be treated in a similar manner.

As stated earlier, we aim to support the most general usage of `replace` function – replacing *all* occurrences. In practice, there is also another version (e.g. in PHP) which allows users to specify the maximum number of occurrences to be replaced. We call it **replaceN**, to distinguish the two versions. In fact, **replaceN** is already supported by existing solvers, e.g., Kaluza. The typical treatment is to model the input parameter as a concatenation of N parts, and then apply one replacement to each part. However, this technique cannot be generalized to address **replaceAll**, since we do not know such an N beforehand. Here we propose to model both **replaceAll** and **replaceN**, again, using recursively-defined functions. In fact, restricting to **replaceN** alone, our approach will be more efficient than Kaluza's. This efficiency comes from the superiority of incremental solving (via constrain-and-generate approach) over generate-and-test approach.

Since **replaceN** is a special case of **replaceAll**, we focus on discussing only the latter. Table 7 shows that `R`=**replaceAll**(`S`, `r`, `T`) belongs to one of two possible cases:

- the recursive case, when we find a substring `M`, that matches regular expression `r`, at an index `I`. We then can replace `M` by `T` and continue to apply **replaceAll** function on the remaining part $\texttt{S}_1$ until we reach the base case.
- the base case, when we cannot find any substring that satisfies such condition. The resulting string `R` is then the same as the input string `S`.

The **replaceAll** function will use **search** function to find the index of substring $\texttt{M}=\texttt{M}_1\cdot\texttt{M}_2$ in `S`. Specifically, this auxiliary function takes as input a symbolic string input `S`, a regular expression `r`, and returns the starting index `I` of a substring in `S` that matches `r`. If there exists no such substring, it returns a negative number. Otherwise, it returns the index of the substring `M` that satisfies the condition.

We remark that the second parameter of **replaceAll** function cannot be a variable since in such case, the behavior of this function is undefined. Naively, we can keep unfolding recursively-defined function **replaceAll**, until we can decide if the current formula is satisfiable or not. However, we provide reduction rules (unfolding on demand) for them instead. For presentation purpose, Table 6 lists only two reduction rules for the case when the prefix of the first parameter S is known[5]. In rule $[\textbf{RED}-\textbf{1}]$, the prefix of S is a constant string s, while it is **star**(s,n) in rule $[\textbf{RED}-\textbf{2}]$. In both cases, since the prefix is already known, we are able to apply the replacement on the part s (**star**(s,n) in the other case) via auxiliary function *rep*. In rule $[\textbf{RED}-\textbf{1}]$, suppose that S is composed by s and R, function $rep(s,r,T)$ replaces all occurrences in s, match-

[5]Other rules related to the second, the third parameter, the result and their combinations are constructed similarly.

Step	Fact added	Eq-class	Reduction/Action
1	`nM` = "*abababababcc*"	{"*abababababcc*", `res`, `nM`, `p1` $\cdot$ `p2`}	$[\mathbf{REP}{-}\star]$: `res` = "*abababababcc*"
2	`p1` = **star**("*ab*",`n1`)	{`p1`, **star**("*ab*",`n1`)}	$[\mathbf{REP}{-}\star]$: `res` = **star**("*ab*",`n1`) $\cdot$ `p2`
3	`p2` = **star**("*bc*",`n2`)	{`p2`, **star**("*bc*",`n2`)}	$[\mathbf{REP}{-}\star]$: `res` = **star**("*ab*",`n1`) $\cdot$ **star**("*bc*",`n2`)
4	`res`= **star**("*ab*",`n1`)$\cdot$**star**("*bc*",`n2`)	{"*abababababcc*", `res`, `nM`, **star**("*ab*",`n1`)$\cdot$ **star**("*bc*",`n2`), `p1` $\cdot$ `p2`}	$[\mathbf{REP}{-}\star]$:**star**("*ab*",`n1`)$\cdot$**star**("*bc*",`n2`)= "*abababababcc*"
5	**star**("*ab*",`n1`) $\cdot$ **star**("*bc*",`n2`) = "*abababababcc*"	{"*abababababcc*", `res`, `nM`, **star**("*ab*",`n1`)$\cdot$ **star**("*bc*",`n2`), `p1` $\cdot$ `p2`}	**star**("*ab*",`n1`)$\cdot$**star**("*bc*",`n2`)="*abababababcc*"$\Rightarrow$ $\neg$**star**("*ab*",`n1`)$\cdot$**star**("*bc*",`n2`)="*abababababcc*"
UNSAT			

Table 5: A Solving Procedure for the Motivating Example in Fig. 3

Rule	Reduction	Condition
$[\mathbf{RED}{-}1]$	$\mathbf{replaceAll}(s{\cdot}R, r, T){=}U \quad \Rightarrow \quad V{\cdot}\mathbf{replaceAll}(t{\cdot}R, r, T){=}U$	$(V, t){=}rep(s, r, T)$
$[\mathbf{RED}{-}2]$	$\mathbf{replaceAll}(\mathbf{star}(s, n){\cdot}R, r, T){=}U \quad \Rightarrow \quad V{\cdot}\mathbf{replaceAll}(t{\cdot}R, r, T){=}U$	$(V, t){=}rep(\mathbf{star}(s, n), r, T)$

Table 6: Reduction Rules for replaceAll Functions

ing the regular expression r, by T. It then returns the pair (V, t) such that $\mathbf{replaceAll}(s, r, T){=}V{\cdot}t$, where t is guaranteed to be the longest suffix of s that must be examined together with R in the next step $\mathbf{replaceAll}(t{\cdot}R, r, T)$. The application of rep for the case **star**(s, n) is similar to s except that V is parameterized by n. Now, we illustrate how this auxiliary function can be applied via two examples. In the first example:

$$\mathbf{replaceAll}(\text{“}abcd\text{”}{\cdot}R, \text{“}ab\text{”}, T) = U$$

the $rep(\text{“}abcd\text{”}, \text{“}ab\text{”}, T)$ method will return $(T{\cdot}\text{“}cd\text{”}, \text{“”})$. In the second one:

$$\mathbf{replaceAll}(\text{“}abcd\text{”}{\cdot}R, (\text{“}ab\text{”} + \text{“}de\text{”}), T) = U$$

it will return $(T{\cdot}\text{“}c\text{”}, \text{“}d\text{”})$ since it is possible that R starts with character 'e'.

Length constraints. We have inherited rules from Z3-str, to infer length constraints such as $X{=}Y \rightarrow \mathbf{length}(X){=}\mathbf{length}(Y)$. Due to space limit, we do not include them in our paper. Importantly, the unfolding of recursive functions (**star**, **replaceAll**, etc.) would incrementally expose more concrete (sub)strings and therefore the interactions from Z3-str-$\star$ to the Arithmetic Solver module in Z3 also happen incrementally.

In addition, as stated in Sec. 4.2, the length constraints, in the feedback from the Arithmetic Solver module, can also be used to prune the search space in string theory component, Z3-str-$\star$. For example, when the Arithmetic Solver module can deduce concrete values for length variables, Z3-str-$\star$ will be able to make use of such information.

6. EVALUATION

In our experimental evaluation, we conduct two case studies to compare S3 with state-of-the-art string solvers. All experiments are run on an 3.2GHz machine with 8GB memory.

In Section 3, we stated that constraint solvers, which work only on string domain or only on non-string domain, are not effective for analyzing web applications. Thus, it is sufficient for us to compare S3 only with Kaluza and Z3-str.

6.1 Comparison with Kaluza

In this case study, we use the set of (50,000+) benchmarks that is shipped with Kaluza, which can be downloaded at:

`http://webblaze.cs.berkeley.edu/2010/kaluza`

They were generated using Kudzu [28], a symbolic execution framework for JavaScript. Since our solver can parse these generated constraints directly, it is straight-forward to plug S3 into Kudzu. The Kaluza benchmarks are classified into two:

- SAT, where Kaluza finds a satisfying solution and returns YES.
- UNSAT, where Kaluza cannot find any solutions and returns NO or it ends up with errors.

For each category, the benchmarks are further divided into two groups: *small* and *big*, based on the size of the files. We note that the classification is done by [28]; such classification is not necessarily accurate in reality. In fact, after rectifying totally 1 file in SAT category and 4057 files in UNSAT category, those having parsing errors (due to incorrect syntax or types), Kaluza can now answer YES (means satisfiable) for 2894 out of 4057 previously UNSAT cases. We will elaborate on this later.

	#Files	Time(s)		
		K	S3	K/S3
SAT/Small	19984	5190	267	19.4x
SAT/Big	1835	3165	166	19.0x
UNSAT/Small	11761	4532	173	26.2x

Table 8: Timing Comparison: S3 vs. Kaluza

We first consider a timing comparison. We ran both Kaluza (column K) and S3 on all the benchmarks in the SAT category, as well the small benchmarks in the UNSAT category. The reason for omitting the large benchmarks in the UNSAT category is that often Kaluza fails to find a definitive answer here (due to crashing or timing out, after 1 minute), and therefore it is not meaningful to compare with its timing. The results, which are summarized in Table 8, clearly show that S3 is much faster, by a factor 19 or more.

In the next experiment, we consider something of perhaps greater importance: *robustness*. Roughly speaking, this measures how often a solver is able to provide a *definitive answer*. This, in turn, means that if the solver returns YES, then it should produce a particular *model* which demonstrates the executability of the path in question. If the solver returns NO, then it should mean that the path in question is in fact not executable. There is of course a third possible answer, MAYBE, which is not definitive (and which is the cause of false alarms). A robust system therefore is one which returns definitive answers often.

As mentioned above, the Kaluza benchmarks are categorized into SAT and UNSAT.

In the SAT category (of $19984 + 1835 = 21819$ benchmarks), S3 finds 6 of them are unsatisfiable. By a careful investigation, it is in fact the case. Moreover, S3 successfully reports YES on all the benchmarks (excluding the 6 which are wrongly classified), and further, the complete models, returned by S3, are *cross-checked* by running them using Kaluza.

Next, we use S3 to cross-check the models produced by Kaluza. Since in Kaluza, each query must specify a variable, for which they will generate the model, in our setting we tested with the variable that starts with '`var`'[6]. As a result, Kaluza has errors with 11 benchmarks that do not have any variable starting with '`var`'. For the other 21808 benchmarks, it reports a YES answer. However, in 695 out of 21813 indeed *satisfiable* benchmarks, the model returned by Kaluza is incomplete. Because Kaluza only returns the model for one variable, it is possible that the return model for the chosen variable may not be extensible to become a complete model which includes other variables. In short, these 695 models are in fact not really models that are useful to reproduce attacks. (We note that [39] has previously remarked this "semi-soundness" issue of Kaluza.)

Table 9 shows statistics for 33230 $(11761 + 21469)$ benchmarks in the UNSAT category. Kaluza reports a YES answer for 2894 out of 4057 rectified benchmarks. For other files (including the other rectified files), Kaluza is not reporting a NO answer to all benchmarks therein; rather, the answer is MAYBE most of the times. In fact, more than half (18210) of the benchmarks in this category was determined by S3 as *SATISFIABLE*! Again, the return models are confirmed by running them using Kaluza. This means that S3 has much more potential for vulnerability detection than Kaluza does.

	S3	Kaluza
NO	14877	7124
YES	18210	2894
ERROR	0	22653
TIMEOUT (1 min)	0	559
MAYBE	143	0

Table 9: S3 vs. Kaluza on UNSAT Category

Note that in Table 9 we no longer distinguish between big and small programs. The frequency of Kaluza crashing (ERROR) or timing out (TIMEOUT, after 1 minute), as opposed to saying NO, is extremely high. Also note that, 2894 files that Kaluza reports SAT (YES) only belong to rectified ones. Moreover, Kaluza often (about 70% of the time) returns a non-definitive answer, either by crashing or timing out. In contrast, S3 returns a definitive answer much more often.

In summary for this study, we have first shown that S3 is far more efficient than Kaluza using its own impressive set of $(50,000+)$ benchmarks, by a significant margin. Perhaps as importantly, we have also shown that for a large number of cases where Kaluza provides no conclusion, S3 can actually provide a definitive conclusion (about 99.6% on the UNSAT benchmarks). Thus S3 is far more efficient and robust than Kaluza.

6.2 Comparison with Z3-str

Recall that Z3-str deals with a *smaller* class of constraints than S3 (since Z3-str cannot handle regular expressions). The purpose of this study is to answer the question: w.r.t constraints that can be handled by *both* of the two solvers, are the performances the same? We now demonstrate that the answer is *no*, via defining the classes that show S3's improvement (esp. our enhanced design).

[6]There is usually one such variable in each benchmark.

Benchmark	Model produced?		Time(ms)		
	Z3-str	S3	Z3-str	S3	Z3-str/S3
ID_3482	NO	YES	_	58	_
ID_3468	NO	YES	_	23	_
ID_1543(*)	NO	YES	_	36	_
ID_3464	NO	YES	_	35	_
ID_3487	NO	YES	_	31	_
ID_new.23484(*)	NO	YES	_	21	_
sat_bnd	YES	YES	3225	120	27x
sat_unbnd	YES	YES	451s	129	3496x
unsat_bnd	_	NO	TO	30	_
unsat_unbnd	_	NO	TO	46	_

Timeout (TO) is at 2h. '*': regular expressions are removed.

Table 10: S3 vs. Z3-str

To demonstrate that S3 is better, we first use six test cases from the SAT benchmarks of Kaluza. We follow the setting of Z3-str [39] and remove all the constraints related to regular expressions. This way we can run Z3-str on the resulting constraints. These six benchmarks are presented in the first part of Table 10. For each of them, while S3 returns YES with a solution model, Z3-str instead returns NO. We note that the models S3 provides are validated as correct by using Z3-str itself.

We now briefly discuss why we have this difference. One reason is that Z3-str cannot acquire the concrete values assigned to length variables. In contrast, our design, presented in Section 4.2, enables the direct interactions between the string solver plug-in Z3-str-$\star$ and Z3 core, to query if the lengths of some string variables have been deduced or constrained in the arithmetic theory. This helps Z3-str-$\star$ avoid repetitive case analysis.

More specifically, the six we use in Table 10, have the following (frequent) pattern: there exists at least one variable that is only constrained by its length. Basically, with the constraint $\mathbf{length}(x){=}i$, the solution for x can be any string of length i, i.e. "@..@", where each @ is an arbitrary character. However, Z3-str cannot make use of this length constraint and keeps trying to assign string value for x, starting from the empty string. Given that x is constrained by its length, Z3-str must try-and-test many times until there is no more conflict with that length constraint. Thus, the total number of values to be tested by Z3-str will be blown up, preventing it from finding a solution.

We next consider another set of benchmarks, representing another pattern (which is also frequent in Kaluza's benchmarks): there exists a relationship between the lengths of different string variables. Indeed the example presented in Fig. 4 resembles such pattern. See the second part of Table 10, where statistics for 4 benchmarks are shown. We purposely make two benchmarks satisfiable – names start with 'sat', whereas the other two are unsatisfiable – names start with 'unsat'. In the two whose names end with 'bnd', the lengths of the string variables are bounded by 10, while in the other two (the names end with 'unbnd'), there is no such bound. For each satisfiable benchmarks, both Z3-str and S3 can find a correct solution model. However, S3 outperforms Z3-str significantly by an order of magnitude. For the unsatisfiable cases, while S3 returns NO within a second, Z3-str runs for more than 2 hours without producing an answer.

In summary, our design allows the full interaction between string theory and arithmetic theory, enabling S3 to handle length constraints more effectively. Thus, even discounting the fact that S3 solves a more general class of constraints than Z3-str, its performance is much better in the common class of constraints. This ensures its applicability in web programs, where length constraints are ubiquitous.

7. RELATED WORK

Symbolic execution has recently been exploited to address a wide range of security problems. Some notable examples are: automated fingerprint generation [7], protocol replay [25], automated code transformation to eliminate SQL injection attacks in legacy web applications [4].

Motivated by the problem of analyzing JavaScript code for the purpose of detecting security flaws, [28] proposed a framework, Kudzu, which leverages the benefits of both concrete and symbolic evaluation. This work effectively reduced the analysis problem of web applications to the problem of solving string constraints. In order to be widely applicable, it is important to have a string solver which is able to reason about both string and non-string constraints. Importantly, the solver must also support constraints involving regular expressions and with multiple variables.

There is a vast literature on the problem of string solving. In previous Sections, we have carefully positioned our work against Kaluza and Z3-str. We now focus on other closely related work.

Practical methods for solving string equations can loosely be divided into bounded and unbounded methods. Bounded methods (e.g., HAMPI [22], CFGAnalyzer [3], and [18]) often assume fixed length string variables, then treat the problem as a normal constraint satisfaction problem (CSP). These methods can be quite efficient in finding satisfying assignments and often can express a wider range of constraints than the unbounded methods. However, as also identified in [28], there is still a big gap in order to apply them to constraints arising from the analysis of web applications.

In the spirit of Kaluza, [6] proposed to reason about feasibility of a symbolic execution path from high-level programs, of which string constraints are involved. In principle, the approach is similar to Kaluza: it proceeds by first enumerating concrete length values, before encoding strings into bit-vectors. It supports common integer related string operations, taken from the basic **.NET** string library, except for `replace`. Unlike Kaluza, however, regular expressions are not supported here. In a similar manner, [27] addresses multiple types of constraints for Java PathFinder. Though this approach can handle many operators, it provides limited support for `replace`, requiring the result and arguments to be concrete. Furthermore, it does not handle regular expressions. In summary, the above methods are less powerful than S3 in terms of the expressiveness of the input language. Importantly, they have similar limitations as Kaluza, which we have carefully discussed.

PISA [33] is the first path- and index-sensitive string solver that targets static analysis of web applications. The verification is conducted by encoding the program in Monadic Second-Order Logic (M2L). It supports regular expressions as well as Java's `replace` method. However, it does not support binary operations between two variables, i.e., PISA requires at least one of them to be constant. Also importantly, its expressiveness for arithmetic operations is restricted due to the limitations of M2L. For example, it does not support numeric multiplications and divisions.

Other unbounded methods are often built upon the theory of automata or regular languages. We will be brief and mention a few notable works. Java String Analyzer (JSA) [10] applies static analysis to model flow graphs of Java programs in order to capture dependencies among string variables. A finite automata is then derived to constrain possible string values. The work [31] used finite state machines (FSMs) for abstracting strings during symbolic execution of Java programs. They handle a few core methods in the `java.lang.String` class, and some other related classes. They partially integrate a numeric constraint solver. For instance, string operations which return integers, such as `indexOf`, trigger case-splits over all possible return values.

In short, using automata and/or regular language representations potentially enables the reasoning of infinite strings and regular expressions. However, most of existing approaches have difficulties in handling string operations related to integers such as `length` and `indexOf`, let alone other high-level operations addressed in this paper. More importantly, to assist web application analysis, it is necessary to reason about both string and non-string behavior together. It is not clear how to adapt such techniques for the purpose, given that they do not provide native support for constraints of the type integer.

Since our method does not rely on the length bounds in enumerating solutions, and our particular treatment of (possibly unbounded) recursive operations is lazy, it is possible that S3 can handle query of unbounded length variables as well as unbounded regular expression. However, to guarantee termination, we do rely on the fact that the lengths are bounded. In fact, our work *targets* the input constraints arising from realistic web applications. Therefore, even when the lengths are not precisely known – in the case of static analysis – it is reasonable to assume that the lengths of input string variables are indeed bounded, as many modern practical string solvers do.

8. CONCLUDING REMARKS

This paper presents a new algorithm for solving string constraints. The class of constraints is practically expressive, for its intended purpose of analyzing web programs which manipulate string inputs. Experimental evaluations show that our solver S3, despite being more expressive than other solvers, is much more robust and efficient.

We remark that in lieu of presenting an end-to-end system, we show that our proposed solver is indeed a *modular* contribution to any hypothetical dynamic symbolic execution end-to-end system. That is, the superior performance of our solver can be used, *without significant engineering* of integrating it, to obtain an improvement in the hypothetical system.

We believe, based on its symbolic representation of string constraints, S3 can also be extended to be more efficient in the context of static analysis, where even regular expressions can also be symbolically constructed.

Astute readers might already notice that our underlying symbolic representation goes well beyond regular languages. As an example, $\{a^n \cdot b^n \mid n \geq 0\}$ can be easily modeled as $\mathbf{star}(a, n) \cdot \mathbf{star}(b, n) \wedge n \geq 0$. While this paper focuses on the practical impact of S3, investigating the theoretical impact of such symbolic representation is left as our future work.

9. REFERENCES

[1] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying Client-side Input Validation Functions Using String Analysis. In *ICSE*, pages 947–957, 2012.

[2] M. Alkhalaf, S. R. 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*, pages 56–66, 2012.

[3] R. Axelsson, K. Heljanko, and M. Lange. Analyzing Context-Free Grammars Using an Incremental SAT Solver. In *ICALP*, pages 410–422, 2008.

[4] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *CCS*, pages 607–618, 2010.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *CCS*, pages 575–586, 2011.

[6] N. Bjørner, N. Tillmann, and A. Voronkov. Path Feasibility Analysis for String-Manipulating Programs. In *TACAS*, pages 307–321, 2009.

[7] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium*, pages 15:1–15:16, 2007.

[8] S. Bucur, J. Kinder, and G. Candea. Prototyping Symbolic Execution Engines for Interpreted Languages. In *ASPLOS*, pages 239–254, 2014.

[9] A. Chaudhuri and J. S. Foster. Symbolic Security Analysis of Ruby-on-rails Web Applications. In *CCS*, pages 585–594, 2010.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, pages 1–18, 2003.

[11] L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[12] M. Emmi, R. Majumdar, and K. Sen. Dynamic Test Input Generation for Database Applications. In *ISSTA*, pages 151–162, 2007.

[13] G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and P. Schachte. Unbounded Model-Checking with Interpolation for Regular Language Constraints. In *TACAS*, pages 277–291, 2013.

[14] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *ICSE*, pages 992–1001, 2013.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[16] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *NDSS*, pages 151–166, 2008.

[17] W. G. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *ISSTA*, pages 285–296, 2009.

[18] J. He, P. Flener, J. Pearson, and W. Zhang. Solving String Constraints: The Case for Constraint Programming. In *CP*, pages 381–397, 2013.

[19] P. Hooimeijer and W. Weimer. A Decision Procedure for Subset Constraints over Regular Languages. In *PLDI*, pages 188–198, 2009.

[20] P. Hooimeijer and W. Weimer. Solving String Constraints Lazily. In *ASE*, pages 377–386, 2010.

[21] C. S. Jensen, M. R. Prasad, and A. Møller. Automated Testing with Targeted Event Sequence Generation. In *ISSTA*, pages 67–77, 2013.

[22] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A Solver for String Constraints. In *ISSTA*, pages 105–116, 2009.

[23] A. Kiezun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic Creation of SQL Injection and Cross-site Scripting Attacks. In *ICSE*, pages 199–209, 2009.

[24] J. Maras, M. Štula, and J. Carlson. Generating Feature Usage Scenarios in Client-side Web Applications. In *ICWE*, pages 186–200, 2013.

[25] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic Protocol Replay by Binary Analysis. In *CCS*, pages 311–321, 2006.

[26] OWASP. Top ten project, May 2013. `http://www.owasp.org/`.

[27] G. Redelinghuys, W. Visser, and J. Geldenhuys. Symbolic Execution of Programs with Strings. In *SAICSIT*, pages 139–148, 2012.

[28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *SP*, pages 513–528, 2010.

[29] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript. In *ESEC/FSE*, pages 488–498, 2013.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *ESEC/FSE*, pages 263–272, 2005.

[31] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid. Efficient Symbolic Execution of Strings for Validating Web Applications. In *DEFECTS*, pages 22–26, 2009.

[32] T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis based on Monadic Second-order Logic. In *ISSTA*, pages 166–176, 2011.

[33] T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[34] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST*, pages 498–507, 2010.

[35] R. Wang, P. Ning, T. Xie, and Q. Chen. MetaSymploit: Day-one Defense Against Script-based Attacks with Security-enhanced Symbolic Analysis. In *SEC*, pages 65–80, 2013.

[36] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *PLDI*, pages 32–41, 2007.

[37] G. Wassermann and Z. Su. Static Detection of Cross-site Scripting Vulnerabilities. In *ICSE*, pages 171–180, 2008.

[38] F. Yu, M. Alkhalaf, and T. Bultan. STRANGER: An Automata-based String Analysis Tool for PHP. In *TACAS*, pages 154–157, 2010.

[39] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A Z3-based String Solver for Web Application Analysis. In *ESEC/FSE*, pages 114–124, 2013.