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 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.

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 the author(s) 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.

CCS’14, November 3–7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660372.

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.

<table>
<thead>
<tr>
<th>Code Coverage</th>
<th>Dynamic Analysis</th>
<th>DSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>False Positives</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Executable Paths (EPs)</td>
<td>Unlikely to cover</td>
<td>Likely to cover all EPs</td>
</tr>
</tbody>
</table>

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.
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:

```php
<!DOCTYPE html>
<html>
<body>

<form name="loginForm" action="/Login"
         onsubmit="return validateEmail(this);">
  Email: <input type="text" name="email"
             size="64" />
  <br>
  <input type="submit" value="Login" />
</form>

</body>
</html>
```

![Figure 1: An Example of Email Address Validation](image)

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., Z3-str [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 of a regular expression, S3 supports only replacement of the first occurrence.

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 3 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.

The input formula can be of the following forms:

- 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; and regular expressions are constructed from string constants using concatenation (·), union (+) and Kleene star (*);
- a composite formula constructed using negation and binary connectives, including \( \land, \lor, \Rightarrow \).

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, replace, replaceAll, 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.

---

1Z3 supports more primitive types [11].

2We use \( x \cdot y \) as a shorter form for \( \text{concat}(x, y) \).

3This operation is used to replace all occurrences.
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 \( \{ a^n b^n c^n \mid n \geq 0 \} \):

\[
x = y \cdot t \land y \in a^* \land z \in b^* \land t \in c^* \land 
\]

\[
\text{length}(y) = \text{length}(z) \land \text{length}(z) = \text{length}(t)
\]

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 pre-processing step as done in Kuzdu [28]. Examples are (possibly negated) character classes, escaped sequences, repetition operators \((\{n\}|\{n\}^*|\{n\}^+|\{n\}^*)\) 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 \( \text{rel} \) and \( \text{re2} \), respectively.

Now we want to prove that, given the inputs which have passed the validation function, the output \( \text{res} \) that is constructed by concatenating \( p1 \) with \( p2 \), is different from a specified bad string \( \text{nM} \) = “abababababce”. 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. 4. The proof succeeds if the formula is unsatisfiable.

This requires reasoning about string equation \( \text{res} = p1 \cdot p2 \), membership predicates \( p1 \in \{ \text{"ab"} \}^* \) and \( p2 \in \{ \text{"bc"} \}^* \), and length constraint \( \text{length}(p2) > 0 \). 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, 18], 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. 5, Kaluza first needs to come up with a set of satisfying solutions for the lengths of \( p1 \) and \( p2 \), each denoted by a pair \( (t1, t2) \), where \( t1 \) is the length of \( p1 \) and \( t2 \) is the length of \( p2 \). In this case, the set of satisfying solutions for the lengths is \( \{ (0, 12), (2, 10), (4, 8), (6, 6), (8, 4), (10, 2) \} \). 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 6 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
particular, \(p_1 \in \{("ab")^*\}\) is represented as \(p_1 = \text{star}("ab", n_1)\) and similarly \(p_2 \in \{("bc")^*\}\) is represented as \(p_2 = \text{star}("bc", n_2)\). By rewriting, we derive the following equation:

\[
\text{star}("ab", n_1) \cdot \text{star}("bc", n_2) = "ababababababc".
\]

Since the length of \(p_2\) is positive and the RHS is a constant string, this would force the unfolding of expression \(\text{star}("bc", n_2)\) to \(\text{star}("bc", n_2 - 1) \cdot "bc"\). A conflict is then derived since the LHS string ends with "bc" while the RHS string ends with "ce". Our system then can conclude that the input formula is UNSAT.

\[
x = x_1 \cdot x_2 \land z = y \cdot z_1 \land y = z_1 \cdot z_2 \land z_2 = "\ast" \land l_1 = \text{length}(x_1) \land l_2 = \text{length}(z_1) \land l_1 = l_2 + 1 \land x = z \land \text{indexOf}(y, "a") = 3 \land \text{indexOf}(x_1, "n") = 4.
\]

### 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, 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 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.

\[
\begin{align*}
\text{assert}((e_1 \lor e_2) \land e_3 \land e_4) \\
e_1 &: x = "abc" \cdot n \\
e_2 &: x = "efgh" \\
e_3 &: y = "efg" \cdot n \\
e_4 &: y = x
\end{align*}
\]

Consider the assertion above. The core component cannot interpret the string operations; instead it treats them as four independent boolean variables \((e_1, e_2, e_3, e_4)\) 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.

<table>
<thead>
<tr>
<th>Fact added</th>
<th>Eq-class</th>
<th>Reduction/Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y = &quot;efg&quot; \cdot n)</td>
<td>(y, &quot;efg&quot; \cdot n)</td>
<td>(\text{Conflict detected})</td>
</tr>
<tr>
<td>(x = y)</td>
<td>({x, y})</td>
<td>({x, y})</td>
</tr>
<tr>
<td>(x = &quot;abc&quot; \cdot n)</td>
<td>({x, &quot;abc&quot; \cdot m, y, &quot;efg&quot; \cdot n})</td>
<td>(\text{backtrack and remove facts})</td>
</tr>
<tr>
<td>(x = &quot;efgh&quot; \cdot n)</td>
<td>({x, &quot;efgh&quot; \cdot n})</td>
<td>(\text{try another option for } e_1)</td>
</tr>
</tbody>
</table>

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

In Table 2, initially there is no fact. The core starts by setting \(e_1\) and \(e_4\) to \(\text{true}\) and reaches step 3. Without loss of generality, assume the core component first tries \(e_1\) to \(\text{true}\). Beware that the core can detect functionally equivalent terms, based on the theory...
of uninterpreted functions. Hence, it puts \( \{ x, y, "abc" \cdot m, "efg" \cdot n \} \) 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 \( \text{length}(x) = 4 \), then Z3-str is not aware of the fact that \( \text{length}(x) \) 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 \( \text{length}(x) \) 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 \( e_3 \land e_4 \rightarrow \neg e_1 \). 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 \( e_6 \) representing \( n = \"h\" \) and an axiom \( "efg\cdot n \Rightarrow e_6 \), which is sent back to the core. From the existing facts and the new axiom, the core component derives \( e_6 \) 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⋆ over Z3-str

Z3-str-star (or Z3-str⋆ 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⋆ (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 \( (=, \prec, \succ, \preceq, \succeq) \) between different length variables?

To answer these questions, we extend Z3 API so that Z3-str⋆ 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.

**Algorithm 1: Top-level Algorithm**

Given an input formula \( F \), S3 recursively reduces \( F \) into new formula \( \text{reduced}_F \), 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 \( \text{reduced}_F \) into disjunctive normal form (DNF). To decide the satisfiability of each disjunct, we extend Z3-str [30] 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⋆. Similar treatment for high-level operations such as replaceAll will be elaborated later. If Z3-str⋆ 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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CONST]</td>
<td>( e \in s \rightarrow e = s )</td>
</tr>
<tr>
<td>[UNION]</td>
<td>( e \in r_1 \lor r_2 \rightarrow e \in r_1 \lor e \in r_2 )</td>
</tr>
<tr>
<td>[CONCAT]</td>
<td>( e \in r_1 \cdot r_2 \rightarrow e \in e_1 \cdot e_2 \land \bigwedge_{i=1}^{n} e_i \in r_i )</td>
</tr>
<tr>
<td>[STAR]</td>
<td>( e \in r^* \rightarrow e = \star(r, n) )</td>
</tr>
</tbody>
</table>

**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⋆ — 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 \[ \text{CONST} \] 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 \[ \text{UNION} \] ensures that the LHS expression \( e \) is a member of one of the RHS sub-expressions (of the union), rule \[ \text{CONCAT} \] splits \( e \) into two fresh string variables, namely \( e_1 \) and \( e_2 \), and checks that they satisfy the condition \( e_i \in r_i \land e_2 \in r_2 \) conjunctively.

The RHS regular expression can also be formed by repeating \( r \) zero or more times (Kleene star). Rule \[ \text{STAR} \] 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 \( \vee \) operator is interpreted as equality operator \( = \). For convenience, we overload \( \equiv \) 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 is then responsible for solving them.

5.3 star Functions

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

\[
\text{Term} : \text{str} \ ::= \text{ConstString} \\
\text{...} \\
\text{star} (\text{Term} : \text{regexpr}, \text{Term} : \text{int})
\]

Like Z3-str, Z3-str* 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, it sends the assignments to our Z3-str plug-in, 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 \( x \in ("ab")^* \), 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 Kalava’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 choose either consume these constants (of course with the capability of backtracking), or find a conflict between unmatchable constants in the two sides.

### Incremental Solving for star Functions

In Table 4 we introduce four auxiliary functions: \( \text{csm}_hld(s, r) \), \( \text{csm}_tl(s, r) \), \( \text{csm}_hd(r_2, r_1) \), and \( \text{csm}_all(s, r) \). 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, \( \text{csm}_tl(s, r) \), consumes the suffix of \( s \) matching \( r \), the third one applies to two regular expressions instead. Lastly, \( \text{csm}_all(s, r) \) 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 \( \text{CON} \cdot \) says about the case when \( \text{star}(r, n) \) equals to some constant string \( s \). As we explained above, method \( \text{csm}_all(s, r) \) is used to decide whether \( s \) can be a member of \( r^* \). If yes, we can update other string expressions that contain \( \text{star}(r, n) \). 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 \( \text{HD} \cdot \cdot \cdot \cdot \cdot \cdot \) and \( \text{TL} \cdot \cdot \cdot \cdot \cdot \cdot \) 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 \( \text{TL} \cdot \cdot \cdot \cdot \cdot \cdot \) for the matching at the end of the LHS.

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

The last rule \( \text{REP} \cdot \cdot \cdot \) 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. Z3 core continually sends the assignments to our Z3-str (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 \( \text{REP} \cdot \cdot \cdot \) 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 \( \text{HT} \cdot \cdot \cdot \cdot \) 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 \( \text{star}(\text{bc}^n \cdot \text{cn}_2) \) so that we can find a conflict between “\text{bc}” and “\text{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

<table>
<thead>
<tr>
<th>Operations</th>
<th>Reduction Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>I=\text{search}(S, r)</td>
<td>((1 &lt; 0 \land -(B \in E \cdot r \cdot )) \lor (1 \geq 0 \land S = U \cdot M_1 \cdot H_2 \cdot R \cdot M_1 \cdot H_2 \in r \land</td>
</tr>
<tr>
<td>R=\text{replaceAll}(S, r, T)</td>
<td>I=\text{search}(S, r) \land ((1 &lt; 0 \land R = S) \lor (1 \geq 0 \land S = U \cdot M_1 \cdot S_1 \cdot R = U \cdot T \cdot R_2 \cdot M \in r \land</td>
</tr>
</tbody>
</table>

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
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 the `generate-and-test` approach.

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 the `generate-and-test` approach.

$$\text{Rule} \quad \text{Reduction} \quad \text{Condition}$$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Reduction</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CON → s</code></td>
<td><code>star(r, n)=s</code> ⇒ <code>¬star(r, n)=s</code></td>
<td><code>¬csm_all(s, r)</code>, <code>s:ConstString</code></td>
</tr>
<tr>
<td><code>CON → s</code></td>
<td><code>star(r, n)=s \land (E_1:star(r, n) \land E_2=Es)</code> ⇒ <code>E_1 \star s \land E_2=Es</code></td>
<td><code>csm_all(s, r)</code>, <code>s:ConstString</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
<tr>
<td><code>HT → s</code></td>
<td><code>star(r_1, n_1) \land E_1:star(r_2, n_2)</code> ⇒ <code>E_1 \star s \land E_2\star s</code></td>
<td><code>[s]=csm_hl(s_1, r_1)</code></td>
</tr>
</tbody>
</table>

Table 4: Selected Reduction Rules for star Functions

- the base case, when we cannot find any substring that satisfies such condition. The resulting string $\emptyset$ is then the same as the input string $\emptyset$.

The `replaceAll` function will use `search` function to find the index of substring $s=\emptyset$, $\emptyset$ in $\emptyset$. Specifically, this auxiliary function takes as input a symbolic string input $\emptyset$, a regular expression $\emptyset$, and returns the starting index $I$ of a substring in $\emptyset$ that matches $\emptyset$. If there exists no such substring, it returns a negative number. Otherwise, it returns the index of the substring $\emptyset$ 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 functions `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 4` lists only two reduction rules for the case when the prefix of the first parameter $S$ is known.\footnote{Other rules related to the second, the third parameter, the result and their combinations are constructed similarly.}

The `replaceAll` function will use `search` function to find the index of substring $s=\emptyset$, $\emptyset$ in $\emptyset$. Specifically, this auxiliary function takes as input a symbolic string input $\emptyset$, a regular expression $\emptyset$, and returns the starting index $I$ of a substring in $\emptyset$ that matches $\emptyset$. If there exists no such substring, it returns a negative number. Otherwise, it returns the index of the substring $\emptyset$ that satisfies the condition.

By remarking 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 functions `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 4` lists only two reduction rules for the case when the prefix of the first parameter $S$ is known.\footnote{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 | \( nM = "abababababec" \) | \{"abababababec\}, res, \( nM, p1 \cdot p2 \) | \{\{\text{REP - i}\}: res = "abababababec"\}
2 | \( p1 = \text{star}("ab",n1) \) | \{\[\text{p1, star}("ab",n1)\]\} | \{\{\text{REP - i}\}: res = star("ab",n1) \cdot p2\}
3 | \( p2 = \text{star}("bc",n2) \) | \{\[\text{p2, star}("bc",n2)\]\} | \{\{\text{REP - i}\}: res = star("bc",n1) \cdot star("bc",n2)\}
4 | \( \text{res= star}("ab",n1) \cdot \text{star}("bc",n2) \) | \{"abababababec"\}, res, \( nM, \text{star}("ab",n1) \cdot \text{star}("bc",n2), p1 \cdot p2 \) | \{\{\text{REP - i}\}: star("ab",n1) \cdot star("bc",n2) = "abababababec"\}
5 | \( \text{star}("ab",n1) \cdot \text{star}("bc",n2) = "abababababec" \) | \{"abababababec"\}, res, \( nM, \text{star}("ab",n1) \cdot \text{star}("bc",n2), p1 \cdot p2 \) | \{\{\text{REP - i}\}: star("ab",n1) \cdot star("bc",n2) = "abababababec"\}

<table>
<thead>
<tr>
<th>Rule</th>
<th>Reduction</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{RED - 1}]</td>
<td>replaceAll(( s \cdot R, r, T )) = U &amp; \Rightarrow V \cdot replaceAll(( t \cdot R, r, T )) = U</td>
<td>(V, t)=rep(s, r, T)</td>
</tr>
<tr>
<td>[\text{RED - 2}]</td>
<td>replaceAll(( \text{star}(s, n) \cdot R, r, T )) = U &amp; \Rightarrow V \cdot replaceAll(( \text{star}(s, n) \cdot R, r, T )) = U</td>
<td>(V, t)=rep(( \text{star}(s, n) \cdot R, r, T ))</td>
</tr>
</tbody>
</table>

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

Table 6: Reduction Rules for replaceAll Functions

ing the regular expression \( r \), by \( T \). It then returns the pair \( (V, t) \) such that 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 replaceAll(\( t \cdot R, r, T \)). The application of rep for the case \( \text{star}(s, n) \) 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:

\[
\text{replaceAll}(\text{"abcd"}, R, \text{"ab"}, T) = U
\]

the \( \text{rep}(\text{"abcd"}, \text{"ab"}, T) \) method will return \( (T, \text{"cd"}, \text{"e"}) \). In the second one:

\[
\text{replaceAll}(\text{"abcd"}, R, (\text{"ab"} \cdot \text{"cd"}), T) = U
\]

it will return \( (T, \text{"ce"}, \text{"d"}) \) 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 \( X = Y \rightarrow \text{length}(X) = \text{length}(Y) \). Due to space limit, we do not include them in our paper. Importantly, the unfolding of recursive functions (\text{star}, \text{replaceAll}, etc.) would incrementally expose more concrete (sub)strings and therefore the interactions from Z3-str-\* to the Arithmetic Solver module in Z3 also happen incrementally.

In addition, as stated in Sec. [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-\*. For example, when the Arithmetic Solver module can deduce concrete values for length variables, Z3-str-\* 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 a 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 straightforward 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.

<table>
<thead>
<tr>
<th>#Files</th>
<th>Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>S3</td>
</tr>
<tr>
<td>SAT/Small</td>
<td>19984</td>
</tr>
<tr>
<td>SAT/Big</td>
<td>1835</td>
</tr>
<tr>
<td>UNSAT/Small</td>
<td>11761</td>
</tr>
</tbody>
</table>

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’. 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.

### Table 9: S3 vs. Kaluza on UNSAT Category

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Model produced?</th>
<th>Time(m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>YES</td>
<td>7124</td>
</tr>
<tr>
<td>Kaluza</td>
<td>14877</td>
<td>7124</td>
</tr>
<tr>
<td>YES</td>
<td>18210</td>
<td>2894</td>
</tr>
<tr>
<td>YES</td>
<td>0</td>
<td>22653</td>
</tr>
<tr>
<td>MAYBE</td>
<td>0</td>
<td>559</td>
</tr>
<tr>
<td>TIMEOUT (1 min)</td>
<td>0</td>
<td>143</td>
</tr>
</tbody>
</table>

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).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Model produced?</th>
<th>Time(m/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z3-str</td>
<td>YES</td>
<td>58</td>
</tr>
<tr>
<td>S3</td>
<td>YES</td>
<td>3225</td>
</tr>
<tr>
<td>timeouts</td>
<td>YES</td>
<td>27x</td>
</tr>
<tr>
<td>timeouts</td>
<td>YES</td>
<td>129</td>
</tr>
<tr>
<td>timeouts</td>
<td>YES</td>
<td>3496x</td>
</tr>
<tr>
<td>timeouts</td>
<td>YES</td>
<td>21</td>
</tr>
</tbody>
</table>

### 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-⋆ 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-⋆ 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 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, \( \{a^n \cdot b^n | n \geq 0\} \) can be easily modeled as \( \text{star}(a, n) \cdot \text{star}(b, n) \) \& \( n \geq 0 \). 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


