ĐẠI HỌC QUỐC GIA HÀ NỘI
**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**


# BÁO CÁO TOÀN VĂN CÔNG TRÌNH THAM DỰ GIẢI THƯỞNG CÔNG TRÌNH NCKH XUẤT SẮC CỦA SINH VIÊN ĐẠI HỌC QUỐC GIA

## NĂM HỌC 2014 – 2015

**Tên công trình:** Z3-regex: A solver for regular expression
**Người thực hiện:** Trần Đức Mười, Đinh Trung Anh
**Lớp:** K56CA
**Giáo viên hướng dẫn:** PGS. TS. Trương Anh Hoàng.

**Hà Nội, năm 2015**

# Table of Contents

# List of Figures

## List of Tables

# ABBREVIATION

| | | |
|---|---|---|
| bool | : | Boolean |
| int | : | Integer |
| regex | : | regular expression |
| S3 | : | Symbolic String Solver |
| SAT | : | satisfiable |
| SMT | : | Satisfiability Modulo Theories |
| UNSAT | : | unsatisfiable |

# Chapter 1

# INTRODUCTION

## 1.1 Motivation

Nowadays, web application has become very popular. It provides a ton of services for web users every day and sometimes it also processes sensitive data[1]. Those protected information should be safeguarded against unwarranted disclosure. However, it is also the reason that malicious users try to attack web app frequently.

According to the list of top 10 most critical web application security risks of OWASP 2013[2], Injection flaws and XSS (Cross Site Scripting) flaws are placed on the first and the third, respectively. These vulnerabilities are caused primarily by the inappropriate using of input strings. Therefore, web applications are needed to be verified if its string-related constraints are satisfiable. Recently, various string solvers are built to cover these problems. The importance of string solving is explained in [1].

Generally, string solving plays an important role in security verification and bug detection. It implies a significant interest of several research groups about modeling and reasoning about strings. Therefore, a number of powerful string solvers have been introduced, including, e.g., HAMPI [2], Kaluza [3], CVC4 [4], Z3-str [5], Z3-str2 [6] and S3 [1]. These apply different approaches to solve satisfiability problems over string equations and some ([1] [4] [6]) also support the regular expression (RE) membership predicates. In this work, we compare Z3-regex mainly with S3 solver because it is the most powerful string solver by the time we do the research.

When we examined carefully S3[3], we found some points that can be improved so we develop our solver called Z3-regex. First, the S3 team mentioned their *Term* : *regexpr*

[1] `http://help.unc.edu/help/what-is-sensitive-data/`
[2] `https://www.owasp.org/index.php/OWASP_Top_Ten`
[3] `http://www.comp.nus.edu.sg/~trinhmt/S3/`

presenting a regular expression in [1] and several equations around it. Their *Term* : *regexpr* is not a Z3 sort[4] but constructed from string constrains using concatenation $(.)$, union $(+)$ and Kleene star $(*)$. In other hand, regular expressions can be presented in string format (RE). This not only no longer requires a convert step before-hand, but also keeps regex in the original form, which is familiar to the programmers.

Second, S3 theoretically can solve the equation of a string and a regular expression function $Star()$. This is mentioned in [1] as Z3-str-star, a component of S3 and an extension of Z3-str. However, since Z3-str has already supported many high-level string operations, it is necessary to support directly the equalization between a high-level string operation and a regular expression function.

Third, there is not any benchmark for regular expression solving evaluation. As mentioned before, S3 has introduced and solved some regular expression functions, but it is not tested in cases. So it is necessary to provide a reliable set of benchmarks for regular expression solving evaluation.

Hence, we address the above problems in our solver called Z3-regex. Headforemost, we need to construct a string solver that can integrate with Z3 SMT solver and solve the regular expression constraints. We determine if the input constraints are satisfiable (SAT) or not (UNSAT). If SAT, we return a model - values of input variables that make the constraints satisfiable.

## 1.2 Contributions

- We develop our tool, Z3-regex on top of Z3-str that can solve regular expression constraints and provide new contraint language for regular expression in addition to existing rich syntax supporting string and non-string terms of Z3-str.
- We register a standalone Z3 sort of regular expression in Z3-regex. We call it *regex* and its simple form *simpleRegex*. An almost complete regex constraints parser is provided as well.

[4] `http://research.microsoft.com/en-us/um/redmond/projects/z3/ml/z3.html`

- We solve frequently operations in regular expression using that are $Matches()$ and $Star()$ and the equalization between them and high-level string expression such as $Concat()$ in Z3-regex.
- We create a small set of benchmarks for regular expression evaluation
- We publish Z3-regex[5] online for further discussion and development.

The rest of the paper is structured as follows. Chapter 2 explains our design and implementation. Chapter 3 provides experimental results. Chapter 4 discusses related works and concludes.

[5] https://github.com/anhtrung93/Z3-str-Regex

# Chapter 2

# DESIGN AND IMPLEMENTATION

## 2.1 Overall Design

Because we extend Z3-regex from Z3-str, it acts as a plug-in string theory for an SMT solver Z3 [7] and communicates with the Z3-core via Z3 API. There are three main components of Z3-regex as illustrates in Figure 1: Pre-processing, Search processing and Dependence analysis. First, Z3-regex converts the input constraints into basic modules: *Concat*, *Length*() and *Star*(). Next, we apply our reduction rules, interact with Z3 core and reduce formulas into simpler equations. Finally, the plug-in theory analyzes the dependence graph from the current context and assigns free string and integer variables. All the above parts will be discussed in next three sections.

## 2.2 An Example

We illustrate our approach via a simple constraint in SMT-LIB format in Code 1.

Code 1: A sample constraints

```
1 (declare-variable x String)
2 (assert(= true (Matches x 'abc[def]*') ) )
3 (assert(= true (EndsWith x "f") ) )
```

Line 1 declares a string variable `x`. Line 2 requires that `x` shoud Start with `"abc"` and then zero or more occurences of `"d"`, `"e"` and `"f"`. Line 3 states that `x` must end with the string `"f"`. We can see that the constraint is satisfiable (SAT) and `"abcf"` is a model for it.

Table 1 shows the simplified solving steps for the above constraint. Each step will be discussed in detail later. Noted that (.) stands for *Concat*() function.

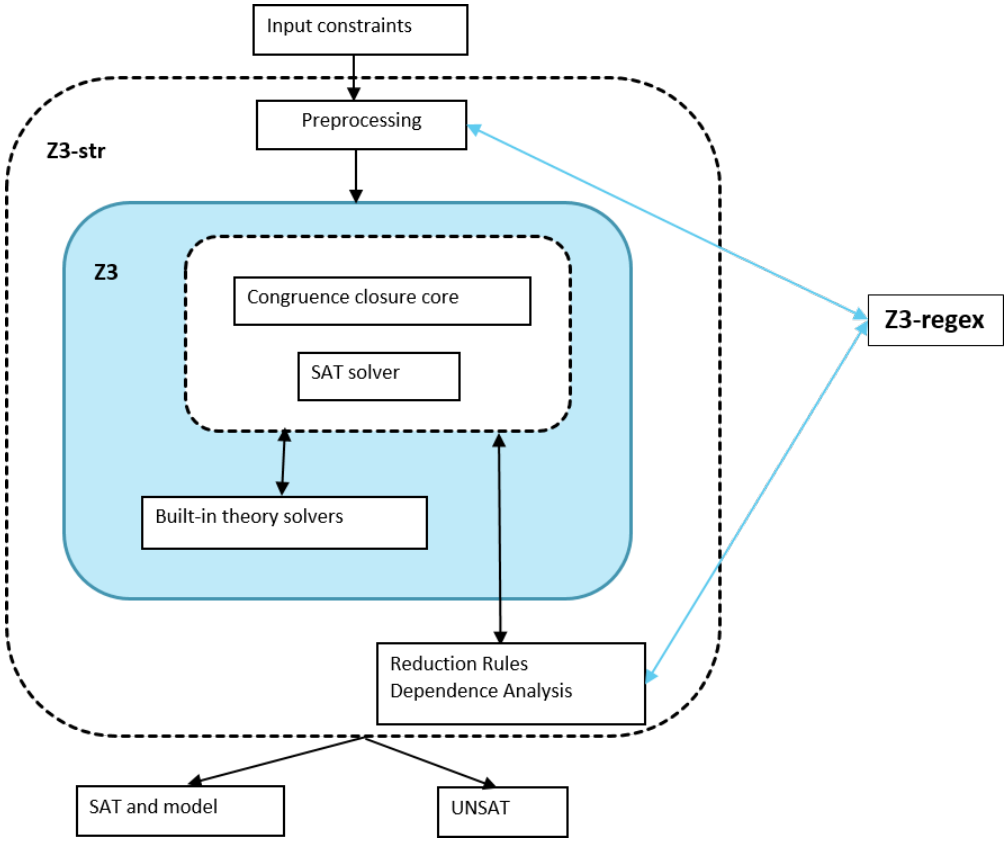


**Figure 1:** The design of Z3-regex

| Step | Eq-class | Operations |
|---|---|---|
| 1 | $\{x, "abc".Star([def], n)\}$ | $\Rightarrow x = "abc".Star([def], n) \wedge n \geq 0$ |
| 2 | $x = "abc".Star([def], n) = s_1."f"$ | $\Rightarrow (s_1 = "abc".s_2 \wedge Star([def], n) = s_2."f"$ $\wedge length(s_1) \geq 3)$ $\vee (s_1 = "abc" \wedge Star([def], n) = "f")$ |
| 3 | $\{x, "abc".Star([def], n), s_1."f"\}$ $\{s_1, "abc".s_2\}$ | Identify simple equations from eq-class: $x = "abc".Star([def], n)$ $x = s_1."f"$ $s_1 = "abc".s_2$ $Star([def], n] = s_2."f"$ |
| 4 | $\{x, "abc".Star([def], n), s_1."f"\}$ $\{s_1, "abc".s_2\}$ | Try to assign $n = 0$ $\Rightarrow x = "abc" \Rightarrow$ conflicts with $x = s_1."f"$ Try to assign $n = 1$ $\Rightarrow x = "abcf" \vee x = "abce" \vee x = "abcd"$ With $x = "abcf" \Rightarrow s_1 = "abc" \Rightarrow s_2 = "f"$ |

**Table 1:** Solving steps for Code 1

## 2.3 Constraint Language

In comparison with Z3-str, Z3-regex supports regular expression and two frequently functions $Matches()$ and $Star()$. Therefore, the input language in Z3-regex was also expanded from Z3-str to accept regular expression constraints.

Z3-regex provides new boolean function $Matches()$ and a new function $Star()$ equal to a string. We also introduce $Term : regex$ that is declared to equal to $\langle RE \rangle$ (the regular expression in string format). The detail new grammar of regular expression is listed in the Table 2.

The input formula of Z3-regex can be one of the following forms:

- Boolean expression
- A comparison operation between two integers or boolean expressions an equation between two string expressions
- A composite formula constructed using negation and binary connectives.
- A membership predicate between a high-level string operations and a regular expression function like $Matches()$ and $Star()$.

$$
\begin{array}{lll}
\langle Term : bool \rangle & ::= & \textbf{Matches}(\langle Term : string \rangle, \langle Term : regex \rangle) \\
\langle Term : string \rangle & ::= & \textbf{Star}(\langle Term : regex \rangle, \langle Term : int \rangle) \\
\langle Term : regex \rangle & ::= & \langle RE \rangle \\
\langle RE \rangle & ::= & \langle union \rangle | \langle s-RE \rangle \\
\langle union \rangle & ::= & \langle RE \rangle \langle RE \rangle \\
\langle s-RE \rangle & ::= & \langle concatenation \rangle | \langle b-RE \rangle \\
\langle concatenation \rangle & ::= & \langle s-RE \rangle \langle b-RE \rangle \\
\langle b-RE \rangle & ::= & \langle star \rangle | \langle plus \rangle | \langle question \rangle | \langle counter \rangle | \langle e-RE \rangle \\
\langle e-RE \rangle & ::= & \langle group \rangle | \langle any \rangle | \langle char \rangle | \langle set \rangle \\
\langle star \rangle & ::= & \langle e-RE \rangle * \\
\langle question \rangle & ::= & \langle e-RE \rangle ? \\
\langle plus \rangle & ::= & \langle e-RE \rangle + \\
\langle counter \rangle & ::= & \langle e-RE \rangle \langle number \rangle \\
\langle group \rangle & ::= & (\langle RE \rangle)
\end{array}
$$

**Table 2:** New grammar of the input constraints

## 2.4 Pre-processing

In this step, Z3-regex reduces constraints and equations to simpler equivalence. This is in order to improve the following processes. To simplify constraints, we apply some reduction rules and transform all the equations to a set of core functions that are $Length()$, $Concat()$ and $Star()$. Noted that, Z3-str also has some pre-processing rules to convert high-level string operations into set of $Length()$ and $Concat()$ equations [5]. The reduction rules are shown in Table 3.

Come back to our example at Table 1, step 1 is in pre-processing stage. At step 1, since variable $x$ matches with '$abd[def]*$', we have a new formula based on our reduction rules: $x = "abc".Star([def], n) \wedge n \geq 0$. Similarly, a new formula is generated in Line 2 from Code 1: $x = s_1."f" \wedge Length(s_1) \geq 0$. Thereafter, we put the equivalent elements into a class (Eq-class): $\{x, "abc".Star([def], n), s_1."f"\}$.

| Condition | Action |
|---|---|
| $s \in \langle char \rangle$ | $\Rightarrow s = \langle char \rangle$ |
| $s \in [\langle set - item \rangle]$ | $\Rightarrow Length(s) = 1 \wedge (\forall c \in \langle set - item \rangle, s = c)$ |
| $s \in \langle RE_1 \rangle \vert \langle RE_2 \rangle$ | $\Rightarrow s \in \langle RE_1 \rangle \vee s \in \langle RE_2 \rangle$ |
| $s \in \langle s - RE \rangle \langle b - RE \rangle$ | $\Rightarrow s_1 \in \langle s - RE \rangle \wedge s_2 \in \langle b - RE \rangle \wedge s_1.s_2 = s$ |
| $s \in \langle e - RE \rangle *$ | $\Rightarrow s = Star(\langle e - RE \rangle, n) \wedge n \geq 0$ |
| $s \in \langle e - RE \rangle +$ | $\Rightarrow s = Star(\langle e - RE \rangle, n) \wedge n \geq 1$ |
| $s \in \langle e - RE \rangle ?$ | $\Rightarrow s =$"" $\vee s \in \langle RE \rangle$ |
| $s \in \langle e - RE \rangle \ \{\langle n_1 \rangle, \langle n_2 \rangle\}$ | $\Rightarrow s = Star(\langle e - RE \rangle, n) \wedge n \geq n_1 \wedge n \geq n_2$ |
| $s \in \langle e - RE \rangle \langle n \rangle$ | $\Rightarrow s = Star(\langle e - RE \rangle, n) \wedge n \geq n$ |

**Table 3:** Preprocessing rules

## 2.5 Search Processing

In this step, Z3-regex applies some reduction rules to different string, integer and regular expression constraints in order to interact the search process in the Z3 core. We provide a new bundle of rules for $Star()$ functions because after pre-processing steps, we have all equations in form of $Star()$, $Length()$ and $Concat()$. The complete rules are shown in Table 4.

Moreover, we also divide $Term$ : *regex* into two types that are *simple regex* and *complex regex* to enhance the performance. We define simple regex as a regular expression with only one possible string solution; a complex regex, on the other hand, has more than one satisfiable string. In other words, after applying our regex reduction rules in pre-processing stage, simple regex returns only one string constant with a constant length. With a constant length, solving for the number of occurrences of that simple regex is much faster when dealing with constant strings or their concatenation functions. For example, the regular expression $'abc'$ is a simple regex because there is only "$abc$" matches with it.

Now we consider our example in Table 1 again with step 2. From the equivalent class, we have an equation that is: "$abc$"$.Star([def], n) = s_1$."$f$". It applies some reduction rules, then implies: $(s_1 =$ "$abc$"$.s_2 \wedge Star([def], n) = s_2$."$f$"$\wedge length(s_1) \geq 3)$ (1)
$\vee (s_1 =$ "$abc$"$\wedge Star([def], n) =$ "$f$"$)$ (2)

From this new formula, the Z3 SMT treats it as two separated branches (1) and (2). Theoretically, Z3 will "push" into one branch, then do a backtrack if the core found any

| | | | |
|---|---|---|---|
| $\mathbf{Star}(r_1,n_1)$ $x_1.s_1$ | $=$ | $\Rightarrow$ | $(\mathbf{Star}(r_1,n_1-1)=x_1.s_2 \wedge s_2.s_3=s_1 \wedge s_3.\mathbf{Matches}(r_1) \wedge n_1 \geq 1)$ $\vee(\mathbf{Star}(r_1,n_1-1).x_2=x_1 \wedge (x_2.s_1).\mathbf{Matches}(r_1) \wedge n_1 \geq 1)$ |
| $\mathbf{Star}(r_1,n_1)$ $s_1.x_1$ | $=$ | $\Rightarrow$ | $(\mathbf{Star}(r_1,n_1-1)=s_2.x_1 \wedge s_3.s_2=s_1 \wedge s_3.\mathbf{Matches}(r_1) \wedge n_1 \geq 1)$ $\vee(x_2.\mathbf{Star}(r_1,n_1-1)=x_1 \wedge (s_1.x_2).\mathbf{Matches}(r_1) \wedge n_1 \geq 1)$ |
| $\mathbf{Star}(r_1,n_1)=$ $\mathbf{Star}(r_2,n_2).x_1$ | | $\Rightarrow$ | $(n_1=n_2=0 \wedge x_1=\text{“”})$ $\vee(n_2=0 \wedge x_1=\mathbf{Star}(r_1,n_1))$ $\vee(n_1 \geq 1 \wedge n_2 \geq 1 \wedge x_2.\mathbf{Star}(r_1,n_1-1)=x_3.\mathbf{Star}(r_2,n_2-1).x_1 \wedge$ $x_2.\mathbf{Matches}(r_1) \wedge x_3.\mathbf{Matches}(r_2))$ |
| $\mathbf{Star}(r_1,n_1)=$ $x_1.\mathbf{Star}(r_2,n_2)$ | | $\Rightarrow$ | $(n_1=n_2=0 \wedge x_1=\text{“”})$ $\vee(n_2=0 \wedge x_1=\mathbf{Star}(r_1,n_1))$ $\vee(n_1 \geq 1 \wedge n_2 \geq 1 \wedge \mathbf{Star}(r_1,n_1-1).x_2=x_1.\mathbf{Star}(r_2,n_2-1).x_3 \wedge$ $x_2.\mathbf{Matches}(r_1) \wedge x_3.\mathbf{Matches}(r_2))$ |
| $\mathbf{Star}(r_1,n_1)$ $x_1.x_2$ | $=$ | $\Rightarrow$ | $(x_1=\text{“”} \wedge x_2=\text{“”} \wedge n_1=0)$ $\vee(x_1=\mathbf{Star}(r_1,n_2) \wedge x_2=\mathbf{Star}(r_1,n_1-n_2) \wedge n_2 \geq 0 \wedge n_1 \geq n_2)$ $\vee(x_1=\mathbf{Star}(r_1,n_2).x_3 \wedge x_2=x_4.\mathbf{Star}(r_1,n_1-n_2-1) \wedge$ $x_3.x_4.\mathbf{Matches}(r_1) \wedge n_2 \geq 0 \wedge n_1-1 \geq n_2)$ |
| $\mathbf{Star}(sr_1,n_1)=$ $\mathbf{Star}(sr_2,n_2)$ | | $\Rightarrow$ | $n_1 * \mathbf{length}(sr_1)=n_2 * \mathbf{length}(sr_2)$ |
| $\mathbf{Star}(sr,n_1)=$ $\mathbf{Star}(r,n_2)$ | | $\Rightarrow$ | $n_1=n_2=0$ $\vee(x_1.\mathbf{Star}(sr,n_1-1)=x_2.\mathbf{Star}(r,n_2-1) \wedge x_2.\mathbf{Matches}(r) \wedge$ $x_1.\mathbf{Matches}(sr) \wedge n_1 \geq 1 \wedge n_2 \geq 1)$ |

**Table 4:** Regular expression reduction rules

conflicts in it. We assume Z3 tries branch (1) first. Therefore, beside the existed equivalent class in the previous step, now we have $\{s_1,\text{“}abc\text{”}.s_2\}$ in another equivalent class as well.

## 2.6 Dependence Analysis

When no more reduction can be applied, the core reaches the form of *simple equation*. If each equivalence class has a constant in one side, the solver will terminate with a SAT solution. Otherwise, it builds a dependence graph of variables involved in simple equations. From the dependence graph, Z3-regex identifies free variables, the variables that do not depend on others and do not have constants in their equivalence class. It then tries to assign concrete values to free variables by adding new axioms.

A free variable $x$ in the string domain may be constrained by length assertions (in the

integer domain) on the variable itself or on other variables that are dependent on $x$, following the free variable rule in [5].

As the $Star()$ function cannot be reduced in previous steps, we now will determine the dependence of its variables. Normally, $Star()$ function has an integer variable in this step and Z3-regex will assign various incremental concrete values for integer variables, then add the new $Star(term : regex, term : int)$ with a known integer as an axiom to Z3.

To have a clearer view at this stage, we will consider step 3 and 4 from the example in Table 1. From previous steps, we can identify a set of simple equations:

1. $x = "abc".Star([def], n).n)$
2. $x = s_1."f"$
3. $s_1 = "abc".s_2$
4. $Star([def], n) = s_2."f"$

Firstly, we try to assign $n = 0$, it implies that $Star([def], n) = Star([def], 0) = ""$. From (1.), we can find $x = "abc"$. However, this value conflicts with $x = s_1."f"$ in (2.). Therefore, Z3 do a backtrack and try an incremental value of $n$.

Secondly, we try to assign $n = 1$, it implies that $Star([def], n) = Star([def], 1)$. This new $Star()$ is parsed by the parser into: $x = "abcd" \vee x = "abce" \vee x = "abcf"$. The Z3 core tries each branch, checks if there is any conflict. After "push" and "pop" several times, Z3 found $x = "abcf"$ satisfying all 4 simple equations above. That value of $x$ implies $s_1 = "abc"$ and $s_2 = "f"$.

Finally, Z3-regex returns a SAT solution: $x = "abcf"$.

# Chapter 3

# EXPERIMENTAL RESULTS

## 3.1 Experimental benchmarks

In this section, we will compare the regular expression solving between Z3-regex and S3 in Table. 5 and 6.

To compare Z3-regex and S3, we use 20 tests obtained from S3 website [1], AppScan Source [2] projects, which is also introduced in [6] and from Z3-regex benchmarks itself. All of these inputs have regular expression constraints in it.

String solvers such S3, CVC4 or Z3-str2 normally use Kaluza Benchmark Suite[3] which has the constraints generated by a JavaScript symbolic execution engine. However, this suite is only publicly available string constraint set, exclude finite regular expression membership. Therefore, we do not include Kaluza benchmark in our experiments.

## 3.2 Performace Results

| | S3 | | Z3-regex | |
|---|---|---|---|---|
| | ✓ | ✗ | ✓ | ✗ |
| SAT | 8 | 2 | 16 | 0 |
| UNSAT | 4 | 5 | 4 | 0 |
| Timeout | | 1 | | |
| Total time(s) | 891(**1x**) | | 1116(**1.25x**) | |

**Table 5:** Overall results in comparison

The results we obtained are summarized in Table 5 and listed in details in Table 6. In Table 5, in "SAT" and "UNSAT" rows, (✗) denotes for the number of incorrect results,

[1] `http://www.comp.nus.edu.sg/~trinhmt/S3/`
[2] `http://www-03.ibm.com/software/products/en/appscan-source`

which is either an "UNSAT" response where the inputs have a solution or an "SAT" response with a wrong model, while (✓) is the rest. The total times count only the tests that both solvers return feasible solution.

According to Table 5, Z3-regex is (✓) in all the benchmarks while S3 solver only return 12/20 true answers. Overall, the total times of S3 is little less than Z3-regex (1.25x). However, it has a timeout test, even we set the timeout equal to 1000 seconds.

| Input | S3 | | | Z3-regex | | |
|---|---|---|---|---|---|---|
| | Results | Models | Time(s) | Results | Models | Time(s) |
| regex-01 | SAT | (✓) | 0.160 | SAT | (✓) | 0.390 |
| regex-02 | SAT | (✓) | 0.070 | SAT | (✓) | 0.058 |
| regex-03 | SAT | (✓) | 0.054 | SAT | (✓) | 0.049 |
| regex-04 | UNSAT | (✓) | 0.053 | UNSAT | (✓) | 0.073 |
| regex-05 | UNSAT | (✓) | 0.057 | UNSAT | (✓) | 0.064 |
| regex-06 | SAT | (✓) | 0.066 | SAT | (✓) | 0.059 |
| regex-07 | UNSAT | (✓) | 0.037 | UNSAT | (✓) | 0.039 |
| regex-08 | SAT | (✓) | 0.050 | SAT | (✓) | 0.044 |
| regex-09 | UNSAT | (✓) | 0.039 | UNSAT | (✓) | 0.045 |
| regex-10 | UNSAT | (✗) | 0.054 | SAT | (✓) | 0.057 |
| regex-11 | SAT | (✓) | 0.064 | SAT | (✓) | 0.046 |
| regex-12 | SAT | (✗) | 0.094 | SAT | (✓) | 0.066 |
| regex-13 | UNSAT | (✗) | 0.063 | SAT | (✓) | 220.172 |
| regex-14 | Time out | | 999.99 | SAT | (✓) | 0.129 |
| regex-15 | SAT | (✗) | 0.062 | SAT | (✓) | 0.068 |
| regex-16 | UNSAT | (✗) | 0.058 | SAT | (✓) | 0.341 |
| regex-17 | UNSAT | (✗) | 0.059 | SAT | (✓) | 1.175 |
| regex-18 | UNSAT | (✗) | 0.059 | SAT | (✓) | 5.496 |
| regex-19 | SAT | (✓) | 0.148 | SAT | (✓) | 0.165 |
| regex-20 | SAT | (✓) | 0.094 | SAT | (✓) | 0.084 |

**Table 6:** Details comparison performance of S3 and Z3-regex

# Chapter 4

# CONCLUSION AND FUTURE WORKS

Since solving string constraints is currently a hot topic and the existing solvers still have some deficiencies in solving regular expression, we develop and introduce our string solver, Z3-regex, which can solve more classes of constraints that the state-of-the-art solvers cannot. Experimental results confirm that Z3-regex can solve well these cases.

Our tool is open source so our implementation can be incorporated to the existing solvers such as S3 and Z3-str2 to increase their capabilities in handling regular expression constraints. In Z3-regex, a new Z3 sort is registered for regular expression. Therefore, Z3-regex can precisely solve equalities between regular expression functions and other expressions in different domains (e.g, string). At the moment, it supports frequently equations over $Matches()$ and $Star()$.

To improve our tool, we plan to re-base our project on Z3-str2 instead of Z3-str as it is now. Z3-str2 experiment results also show that there are many unsolved constraints in the benchmarks. We also plan to study these cases to find problems and solutions for them. Next, we continue our work on Z3-regex to optimize the dependence analysis process when the input constraints is a complex combination between multiples theories such as integer, string and regular expression. Finally, we will build a front-end web for users to use and evaluate our tool. From this, we will verify real-world applications, then make the regular expression benchmarks become richer.

## References


[1] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," *CCS '14 Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1232–1243, 2014.

[2] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, , and M. D. Ernst, "Hampi: A solver for string constraints," *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA '09*, pp. 105–116, 2009.

[3] P. Saxena, D. Akhawe, F. M. Steve Hanna, S. McCamant, , and D. Song, "A symbolic execution framework for javascript," *Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP '10*, pp. 513–528, 2010.

[4] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "A dpll(t) theory solver for a theory of strings and regular expressions," *Proceedings of the 26th International Conference on Computer Aided Verification, CAV'14*, pp. 646–662, 2014.

[5] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: a z3-based string solver for web application analysis," *The 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, pp. 114–124, 2013.

[6] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang, "Effective search-space pruning for solvers of string equations, regular expressions and length constraints," *The 27th International Conference on Computer Aided Verification (CAV 2015)*, 2015.

[7] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," *Technical Report, Microsoft*, 2008.

[8] G. Redelinghuys, "Symbolic string execution," *Master of Science in Computer Science at the University of Stellenbosch*, 2012.

[9] G. Li and I. Ghosh, "Pass: String solving with parameterized array and interval automaton," *9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pp. 15–31, 2013.

[10] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: a theorem prover for program checking," *Journal of the ACM (JACM)*, vol. 52, pp. 365–473, 2005.

[11] P. Hooimeijer and W. Weimer, "A decision procedure for subset constraints over regular languages," *PLDI '09 Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pp. 188–198, 2009.

[12] M. Emm, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," *ISSTA '07 Proceedings of the 2007 international symposium on Software testing and analysis*, pp. 151–162, 2007.

[13] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," *PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 32–41, 2007.

[14] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," *ICSE '08 Proceedings of the 30th international conference on Software engineering*, pp. 171–180, 2008.

[15] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," *Proceeding SAS'03 Proceedings of the 10th international conference on Static analysis*, pp. 1–18, 2003.

[16] D. Shannon, I. Ghosh, S. Rajan, and S. Khurshid, "Efficient symbolic execution of strings for validating web applications," *DEFECTS '09: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pp. 22–26, 2009.

[17] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select–a formal system for testing and debugging programs by symbolic execution," *Proceedings of the International Conference on Reliable Software*, pp. 234–245, 1975.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, pp. 385–394, 1976.

[19] W. E. Howden, "Experiments with a symbolic evaluation system," *Proceedings, National Computer Conference*, 1976.

[20] L. A. Clarke, "A program testing system," *ACM 76: Proceedings of the Annual Conference*, pp. 488–491, 1976.

[21] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t)," *Journal of the ACM (JACM)*, vol. 53, pp. 937–977, 2006.

[22] S. Ranise and C. Tinelli, "The satisfiability modulo theories library (smt-lib)." www.SMT-LIB.org, 2006.