FlashRegex: Deducing Anti-ReDoS Regexes from Examples

Yeting Li
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
liyt@ios.ac.cn

Zhiwu Xu
College of Computer Science and Software Engineering, Shenzhen University
Shenzhen, China
xuzhiwu@szu.edu.cn

Jialun Cao
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
Hong Kong, China
jcaoap@cse.ust.hk

Haiming Chen*
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
Beijing, China
chm@ios.ac.cn

Tingjian Ge
University of Massachusetts Lowell, United States
ge@cs.uml.edu

Shing-Chi Cheung
Department of Computer Science and Engineering, The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Haoren Zhao
Shaanxi Normal University
Xi'an, China
zhaohaoren666@gmail.com

ABSTRACT

Regular expressions (regexes) are widely used in different fields of computer science such as programming languages, string processing and databases. However, existing tools for synthesizing or repairing regexes were not designed to be resilient to Regex Denial of Service (ReDoS) attacks. Specifically, if a regex has super-linear (SL) worst-case complexity, an attacker could provide carefully-crafted inputs to launch ReDoS attacks. Therefore, in this paper, we propose a programming-by-example framework, FlashRegex, for generating anti-ReDoS regexes by either synthesizing or repairing from given examples. It is the first framework that integrates regex synthesis and repair with the awareness of ReDoS-vulnerabilities. We present novel algorithms to deduce anti-ReDoS regexes by reducing the ambiguity of these regexes and by using Boolean Satisfiability (SAT) or Neighborhood Search (NS) techniques. We evaluate FlashRegex with five related state-of-the-art tools. The evaluation results show that our work can effectively and efficiently generate anti-ReDoS regexes from given examples, and also reveal that existing synthesis and repair tools have neglected ReDoS-vulnerabilities of regexes. Specifically, the existing synthesis and repair tools generated up to 394 ReDoS-vulnerable regex within few seconds to more than one hour, while FlashRegex generated no SL regex within around five seconds. Furthermore, the evaluation results on ReDoS-vulnerable regex repair also show that FlashRegex has better capability than existing repair tools and even human experts, achieving 4 more ReDoS-invulnerable regex after repair without trimming and resorting, highlighting the usefulness of FlashRegex in terms of the generality, automation and user-friendliness.

KEYWORDS

regular expression (regexes), Anti-ReDoS, program synthesis, program repair

ACM Reference Format:
Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. 2020. FlashRegex: Deducing Anti-ReDoS Regexes from Examples . In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416556

1 INTRODUCTION

Regular expressions (regexes) are a fundamental concept across the fields of computer science, e.g., programming languages, string processing tools, and database query languages [5, 10, 14, 15, 29, 47]. Though popular, regexes are hard for users and even experts to understand and compose [10, 29, 51]. To alleviate this problem, various techniques [3–5, 7, 8, 21, 24, 32, 34] have been proposed. From either positive and/or negative examples, these techniques can synthesize regexes by accepting all positive examples while rejecting all negative ones. This *programming-by-example* (PBE) technique has the salient advantage of allowing users to provide examples to reflect their true intentions. However, existing works [14, 15, 29, 47] do not consider the issue of security vulnerability in regex synthesis. As such, the synthesized regexes are subject to attacks.

To be more specific, if a regex has super-linear worst-case complexity (abbrev. SL regex), an attacker may be able to trigger this

*Haiming Chen is the corresponding author.

complexity by malign input, exhausting the victim's CPU resources and causing Regex Denial of Service (ReDoS) [14]. In other words, SL regexes are ReDoS-vulnerable, since for these regexes, the evaluation could take higher-degree polynomials (*i.e.*, quadratic or worse) or even exponential time in the size of the input. In 2016, ReDoS led to an outage at StackOverflow [19] and rendered vulnerable any websites built with the popular Express.JS framework [2]. Furthermore, a recent study [14] found that SL regexes are rather common in practice—they appear in the core Node.JS (JavaScript) and Python libraries as well as in thousands of modules in the npm [38] and PyPI [23] module registries, including some popular modules with millions of downloads per month. Similar concerns are made by Wstholz [59]: "*Since it is often difficult for humans to reason about the complexity of regular expression matching, we believe there is a real need for techniques that can automatically synthesize equivalent regular expressions with linear complexity*".

We extend this idea to not only **synthesize** ReDos-invulnerable regexes, but also help **repair** incorrect[1] and/or ReDos-vulnerable regexes. Existing techniques [33, 39, 44] that aim at repairing regexes usually focus only on the incorrect ones. There is only two line of work [11, 55] targeting on repairing ReDos-vulnerable regexes; yet the restriction to revisions that would match the exact same *languages* (refer to Section 2 for definition) is actually rarely applicable in practice (for example, cases #5, #7, and #20 in Table 3) according to [14, 47]. To achieve the aforementioned goals , there are three challenges to be addressed.

Huge search space. For both regex synthesis and repair, the search space is extremely large [32, 39] because practical regexes: (i) are large, (ii) operate over very large alphabet size, and (iii) contain various operators such as disjunction, concatenation, quantifiers, and so on.

Difficulty of learning regexes from examples. The problem of synthesis- [25] and repair-from-examples [39] is shown to be an NP-complete problem. We further add the requirement of ReDoS-invulnerabilities, making the problem even harder.

Difficulty of prevention of ReDoS-vulnerabilities. Instead of avoiding certain patterns of regexes as prerequisites of ReDoS attacks [14], developers or users expect to address ReDoS-vulnerability from its root cause—the ambiguity of regexes. Indeed, ambiguity can lead to SL behavior (also known as catastrophic backtracking) that causes ReDoS attacks [14]. How to avoid generating these ambiguous regexes effectively is a distinct merit of our work over existing techniques.

Our key idea is that using *deterministic regexes* (DREs) [9] can prevent generating SL regexes. As its name implies, *determinism* means that when matching a string from left to right against a regex, a symbol in the string can be matched to only one position in the regex without lookahead. Take two equivalent[2] regexes as an example: (i) Non-deterministic regex: `\s*#?\s*` [14] and (ii) DRE: `\s*(#\s*)?`. If we input a whitespace character (*i.e.*, `\s`), the first regex cannot decide which `\s*` (the first or the second) should be matched without looking ahead, while the DRE can efficiently match the first `\s*` without ambiguity. From this example, we can see that one merit of DREs is their efficient matching with malign input avoiding catastrophic backtracking [14], because each matching position in the input word can be uniquely decided, which is in line with the spirit of avoiding ReDoS-vulnerabilities.

This paper proposes FlashRegex with three functionalities: (i) regex synthesis, (ii) incorrect regex repair, and (iii) ReDoS-vulnerable (*i.e.*, SL) regex repair. Given positive and/or negative examples, FlashRegex automatically generates anti-ReDoS regexes that are consistent with the given examples.

We develop novel algorithms that generate anti-ReDoS regexes by reducing the ambiguity of these regexes and by using Boolean Satisfiability (SAT) and Neighborhood Search (NS) techniques. In particular, we design Boolean formulae for the *determinism constraint* and *positive and negative example constraints*, and use a heuristic strategy called *Local Constraints Strengthening* (LCS) to further speed up the process (see Figure 3 in Section 4), enabling us to encode the synthesis problem into SAT. For regex repair, FlashRegex adopts a strategy that a regex after repair is as close to its original regex as possible. To this end, we present a way of slightly changing the edges of the current regex (represented by an automaton) (*i.e.*, the *neighborhood*) and perform neighborhood search starting from the original regex using a *measure function* (see Section 5), until a solution is found. In fact, the synthesis problem and the repair problem are related. The latter can be reduced to the former by disregarding the original regex. However, the regex such repaired may be very dissimilar to the original one.

Although in this paper we only consider the regexes that do not contain non-regular operators (e.g., positive lookahead), our tool is expressive enough to capture most of the regexes appearing in practical applications according to the statistical result [39]. We evaluate FlashRegex by comparing FlashRegex with five state-of-the-art tools in terms of effectiveness (including the correctness and ReDoS-vulnerabilities) and time efficiency on the publicly available benchmarks. The evaluation results reveal that FlashRegex is the only technique that can run on all benchmarks with higher efficiency and generate anti-ReDoS regexes, while the results of other tools can be ReDoS-vulnerable. For example, on benchmark *Multi-Syn-Regex*, FlashRegex reduces the average runtime from more than 1 hour by GP-RegexGolf [4] containing 4 ReDoS-vulnerable results to 4 seconds and without ReDoS-vulnerabilities. The evaluation results on ReDoS-vulnerable regex repair also show that FlashRegex has better capability than existing repair tools and even human experts (see Section 6.4), demonstrating the usefulness of our work.

To summarize, this paper makes three main contributions:

- We develop FlashRegex, a programming-by-example framework, to deduce anti-ReDoS regexes by either synthesizing or repairing from given examples. To the best of our knowledge, it is the first framework that integrates regex synthesis and repair with the awareness of ReDoS-vulnerabilities.
- We present novel algorithms to generate anti-ReDoS regexes by reducing the ambiguity of these regexes. The processes are greatly accelerated by using deterministic automata and optimizations such as the LCS strategy and SAT techniques.
- We conduct a series of comprehensive experiments comparing FlashRegex with the state-of-the-art tools. The evaluation results show that FlashRegex can effectively and efficiently generate anti-ReDoS regexes from given examples, and also

[1]A regex is incorrect if it is not consistent with all the given examples.
[2]Two regexes are equal iff the corresponding languages are equivalent.

reveal that existing synthesis and repair tools have neglected ReDoS-vulnerabilities of regexes.

2 PRELIMINARIES

Let Σ be a finite alphabet of symbols. The set of all words over Σ is denoted by Σ^*. The empty word and the empty set are denoted by ε and $\varnothing$, respectively.

A **regular expression (regex)** over Σ is defined inductively as follows: ε, $\varnothing$, and $a \in \Sigma$ are regular expressions; for regular expressions r_1 and r_2, the disjunction $r_1|r_2$, the concatenation r_1r_2, and the quantifier $r_1\{m,n\}$ where $m \in \mathbb{N}$, $n \in \mathbb{N} \cup \{\infty\}$, and $m \leq n$ are also regular expressions. Besides, $r?$, r^*, r^+ and $r\{i\}$ where $i \in \mathbb{N}$ are abbreviations of $r\{0,1\}$, $r\{0,\infty\}$, $r\{1,\infty\}$ and $r\{i,i\}$, respectively. $r_1\{m,\infty\}$ is often simplified as $r_1\{m,\}$.

The *language* $L(r)$ of a regex r is defined as follows: $L(\varnothing) = \varnothing$; $L(\varepsilon) = \{\varepsilon\}$; $L(a) = \{a\}$; $L(r_1|r_2) = L(r_1) \cup L(r_2)$; $L(r_1r_2) = \{vw \mid v \in L(r_1), w \in L(r_2)\}$; $L(r\{m,n\}) = \bigcup_{m\leqslant i\leqslant n} L(r)^i$.

Practical regexes support special operators , also known as character classes, to denote certain sets of characters. Common character classes include: (i) `\s` , which contains all whitespace characters, (ii) `\d` , which contains digital characters, (iii) intervals $[c_1\text{-}c_2]$ which specifies a range of characters from c_1 to c_2 by using a hyphen, for example, the regex `[a-d]` is the same as `(a|b|c|d)`.

A marked form of a regex r is denoted as $\overline{r}$, which is obtained by marking symbols in r with subscripts, such that each marked symbol occurs only once in $\overline{r}$. For instance, given an expression $r = a(a|b)(ab)^*$, its marked form can be $a_1(a_2|b_1)(a_3b_2)^*$. The same notation will also be used for unmarking, namely, dropping off subscripts from the marked symbols: $\overline{\overline{r}} = r$. We extend this notation for words and sets of symbols in the same way. The set of symbols that occur in a regex r is denoted by $\mathrm{sym}(r)$.

Definition 1. **Deterministic Regular Expression (DRE) [9].** Let $\overline{r}$ be a marked form of the expression r, $\overline{r}$ is deterministic if and only if for all words uxv, $uyw \in L(\overline{r})$ where $x, y \in \mathrm{sym}(\overline{r})$ and $u, v, w \in \mathrm{sym}(\overline{r})^*$, if $\overline{x} = \overline{y}$ then $x = y$. An expression r is deterministic iff $\overline{r}$ is deterministic.

The regex $r_1 = a^*a$ is nondeterministic since $\overline{r}_1 = a_1^*a_2$ is nondeterministic: given two words $a_1a_2, a_2 \in L(\overline{r}_1)$, there is $\overline{a}_1 = \overline{a}_2 = a$ but $a_1 \neq a_2$. Similarly, one can verify that $r_2 = aa^*$ is deterministic.

Notice that DREs are a restricted subclass of regular expressions —i.e., not every regular expression has an equivalent DRE.

Definition 2. A k-Occurrence Regular Expression (k-ORE) [7, 8] is a regular expression in which every alphabet symbol occurs at most k times.

For example, ab^+ is a 1-ORE and $a(ab)^?b^+$ is a 2-ORE. Note that 1-ORE is also known as Single-Occurrence Regular Expression (SORE) [8, 24]. A k-Occurrence Automaton (k-OA) is a specific type of finite state automaton defined in the following (where the states are labeled with symbols while edges are not).

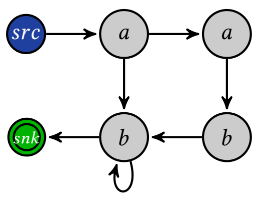
Definition 3. ***k*-Occurrence Automaton (*k*-OA) [7, 34].** A k-OA is a node-labeled graph G=(V, R, lab) where:

- V is a finite set of nodes (also known as states) with a distinguished source src and sink snk.
- R is a set of edge relations representing reachable paths. src has only outgoing edges and snk has only incoming edges. Every $v \in V \backslash \{src, snk\}$ is reachable by a path from src to snk.
- lab is the labeling function $V \setminus \{src, snk\} \rightarrow \Sigma$.
- There are at most k states labeled with the same symbol in Σ.

We use $out(v,\sigma)$ to denote $\{v_1|(v,v_1) \in R \text{ and } \sigma = lab(v_1)\}$, i.e., the set of direct successors of a state v in $\mathcal{A}$ that are labeled σ.

A Single-Occurrence Automaton (SOA) [8, 24] is a special case of k-OA where $k = 1$. A ***marked*** k-OA $\mathcal{A}$ is a k-OA where each node is marked with a subscript such that each node label is unique in $\mathcal{A}$. It is clear that a marked k-OA is an SOA. A ***deterministic*** k-OA is a k-OA in which for each node $v \in V$ and $\sigma \in \Sigma$, $out(v,\sigma)$ contains at most one state.

We can use an adjacency matrix to represent a k-OA. The 2-OA $\mathcal{A}$ for the regex $a(ab)^?b^+$ and its adjacency matrix $\mathcal{A}_G$ are shown in Figure 1.



$$\mathcal{A}_G = \begin{array}{c} \\ src \\ a \\ a \\ b \\ b \end{array} \begin{array}{c} \begin{array}{ccccc} a & a & b & b & snk \end{array} \\ \begin{pmatrix} T & F & F & F & F \\ F & T & F & T & F \\ F & F & T & F & F \\ F & F & F & T & F \\ F & F & F & T & T \end{pmatrix} \end{array}$$

Figure 1: 2-OA $\mathcal{A}$ for $a(ab)^?b^+$ and its adjacency matrix $\mathcal{A}_G$.

3 OVERVIEW

In this section, we present an overview of our approach.

Regex Synthesis. The first problem we target at is to synthesize anti-ReDoS regexes from positive and negative examples. That is, given a positive example set S^+ and a negative example set S^-, the goal is to learn a regex r such that (i) $S^+ \subseteq L(r)$ and $S^- \cap L(r) = \varnothing$; and (ii) r is invulnerable to ReDoS attacks.

The key of our solution to tackle this problem is the use of deterministic regexes. In particular, our solution consists of two steps, namely, *k-OA synthesis* (Section 4.1) and *regex extraction* (Section 4.2). *k-OA synthesis* takes the given positive and negative examples as input and tries to synthesize a deterministic k-OA from the examples via SAT. This task first uses a Boolean variable to represent a possible edge relation between two nodes, wherein T represents an edge exists while F represents no edge exists; and then it encodes the properties of the possible k-OAs into Boolean formulas, which are then fed into a SAT solver. If the formulas are satisfiable, the SAT solver returns a solution, from which *k-OA synthesis* builds a k-OA.

After that, *regex extraction* marks the synthesized deterministic k-OA and extracts a marked regex from the marked k-OA, by calling the procedure *Soa2Sore* used in Freydenberger and Kotzing's work [24]. This procedure *Soa2Sore* builds from a given SOA $\mathcal{A}$ an SORE r that minimally generalizes $L(\mathcal{A})$ (see [24] for the explanation of minimal generalization). Then *regex extraction* unmarks the regex and returns it if it is deterministic.

Generally, regexes with smaller nested quantifiers (*e.g.*, "star height") are less likely to suffer from ReDoS attacks. We also find that a regex extracted from a k-OA with a *smaller* value of k has a *smaller* nested quantifiers. Thus, our solution will perform these two tasks starting from $k = 1$. Once a possible regex is found, our solution returns this regex immediately. Furthermore, a k of 7 is sufficient to capture the intended semantics of 95.76% of the regexes from RegExLib [45], and it is treated as a user parameter controlling how long the synthesizer searches. The bounding of k is helpful in reducing the huge search space.

Regex Repair. The second problem is to repair an incorrect (*i.e.*, rejecting some examples in S^+ or accepting some examples in S^-) or ReDos-vulnerable regex r (*i.e.*, ReDoS-prone) with respect to a positive example set S^+ and a negative example set S^-. The idea is quite similar to regex synthesis: to use deterministic regexes when possible. That is, our solution (Section 5) tries to derive a deterministic regex from r to achieve the same goal of regex synthesis.

In detail, our solution starts with a deterministic k-OA, which is converted from the given regex r. Then it searches for a k-OA which can accept the most positive examples and/or reject the most negative ones among those in the neighborhood (*i.e.*, those with one different value from the current k-OA). Our method keeps on searching, until it finds a deterministic k-OA that accepts all the positive examples and rejects all the negative ones, or the number of iterations exceeds a given number (set to be 200 in this paper).

A Case Study of Regex Synthesis. We illustrate our solution with an example, whose positive example set is $S^+ = \{ab, abbb, aabb\}$ and negative example set is $S^- = \{ba, aab, baba\}$. Assume the automaton to synthesize is a deterministic 2-OA. That is to say, we have at most two nodes labeled with a (resp. b), which are denoted as a_1 and a_2 (resp. b_1 and b_2), respectively. And we represent a possible edge between two nodes u and v as a Boolean variable $A_{u,v}$. Firstly, as the 2-OA is deterministic, there is at most one node that is labelled by any symbol $\sigma \in \{a, b\}$ as well as direct successors of any node $v \in \{a_1, a_2, b_1, b_2, src, snk\}$. That is to say, $out(v, \sigma)$ contains at most one node, which can be encoded as a Boolean formula. For example, the condition for $out(a_1, a)$ can be encoded as the following formula: $(\neg A_{a_1,a_1} \wedge \neg A_{a_1,a_2}) \vee (A_{a_1,a_1} \wedge \neg A_{a_1,a_2}) \vee (\neg A_{a_1,a_1} \wedge A_{a_1,a_2})$. We do the same to other $out(v, \sigma)$'s. Secondly, each word w in S^+ should be accepted by this synthesizd 2-OA; hence there exists a path for w. Take ab as an example. As both a and b have two possible labelled nodes, there are four possible cases for the path of ab, which can be represented as the following formula:

$$(A_{src,a_1} \wedge A_{a_1,b_1} \wedge A_{b_1,snk}) \vee (A_{src,a_1} \wedge A_{a_1,b_2} \wedge A_{b_2,snk}) \vee (A_{src,a_2} \wedge A_{a_2,b_1} \wedge A_{b_1,snk}) \vee (A_{src,a_2} \wedge A_{a_2,b_2} \wedge A_{b_2,snk})$$

It is the same with the other words in S^+. Finally, we also require that each word w in S^- should not be accepted by the 2-OA, which can be converted into Boolean formulas similarly (with a negation on the top). When the formulas are generated[3], we feed the formulas into the SAT solver and get a solution, which is illustrated as the matrix $\mathcal{A}_G$ in Figure 1. Based on this solution, a deterministic 2-OA $\mathcal{A}_2$ can be built easily, which is also given in Figure 1.

[3]We can also generate the formulas according to the local properties, such as 2-successive-steps, of the given words to synthesize a sound k-OA faster, which are called Local Constraints Strengthening (LCS) heuristic strategy (see Section 4.1 for more detail).

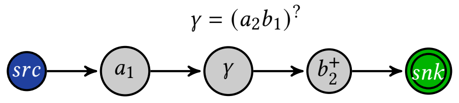


Figure 2: An intermediate SOA in $Soa2Sore(\overline{\mathcal{A}_2})$

Next, we mark $\mathcal{A}_2$, on which we call the procedure *Soa2Sore* to extract a regex. Figure 2 gives an intermediate SOA obtained in $Soa2Sore(\overline{\mathcal{A}_2})$, from which a marked regex $a_1(a_2b_1)^?b_2^+$ is built by *Soa2Sore*. Therefore, the final regex we extract for $\mathcal{A}_2$ is $r_2 = a(ab)^?b^+$.

4 SYNTHESIS ALGORITHM

In this section, we present the details of our synthesis algorithm *SynRegex*.

Our synthesis algorithm is shown in Algorithm 1, which tries to synthesize a possible k-ORE for k ranging in $[1, k_{max}]$. Here, k_{max} is a customized parameter, and in this paper, k_{max} is set to 7 due to the reason discussed in §3. In detail, for a given k, our algorithm first synthesizes a deterministic k-OA from the given examples (line 2), which is introduced in Section 4.1, and then it extracts a regex from the marked version of the synthesized k-OA (line 4), which is presented in Section 4.2, if the k-OA is not *null*. Finally, our algorithm returns the regex with the smallest k if found (lines 5 – 6), or returns *null* otherwise (line 7).

Algorithm 1: SynRegex

```
Input: a positive set $S^+$ and a negative set $S^-$
Output: a deterministic regex $r$ with $S^+ \subseteq L(r)$ and
        $S^- \cap L(r) = \varnothing$ if solvable for $k_{max}$, or null
        otherwise
1 for $k = 1$ to $k_{max}$ do
2     $\mathcal{A} \leftarrow synKOA^{\pm}(S^+, S^-, k)$
3     if $\mathcal{A} \neq null$ then
4         $r \leftarrow GenRegex(\overline{\mathcal{A}})$
5         if $r \neq null$ then
6             return $r$
7 return null
```

4.1 k-OA Synthesis

This section presents the k-OA synthesis algorithm, which synthesize a deterministic k-OA by encoding the synthesis problem into a Boolean satisfiability problem (SAT).

As shown in Figure 1 (Section 2), a k-OA can be represented as a Boolean adjacency matrix. Therefore, we use a Boolean variable to represent a possible edge in a k-OA. Once all the Boolean variables are fixed, a k-OA is obtained meanwhile.

Recall that the k-OA to be synthesized (a) is deterministic and (b) accepts the words in S^+ while rejecting the words in S^-. For the condition (a), $out(v, \sigma)$ contains at most one state for any $v \in V$ and $\sigma \in \Sigma$. Then this deterministic condition on a k-OA can be encoded into a formula via the constraint generator Deter(k), which is shown in Figure 3,where Σ is the alphabet, α_m, α_n and α_t are indexed states for $\alpha \in \Sigma$, and $A_{u,v}$ is a Boolean variable representing a possible edge (*i.e.*, a unit in the Boolean adjacency matrix) for two indexed states u and v.

Let us consider the condition (b). As illustrated in Section 3, both positive examples and negative examples can be entirely dealt with by a general example constraint generator (called global constraint generator). Thus, the longer the example, the more the possible cases (i.e., the conjunctive clauses yielded by the transformation on the corresponding logic formulas). Moreover, for negative examples, a top negation needs to be taken, so that the number of cases for a negative example w would be $(2^{|w|} - 1)$ times that of a positive example of the same length, where $|w|$ denotes the length of w. As a result, with the increase of negative examples, the runtime of the algorithm could grow significantly, especially for the long ones. To speed up the k-OA synthesis algorithm, we present a heuristic algorithm *fast*KOA$^{+}_{-}$, which adopts local constraints, rather than the global constraints on the whole words. We call the strategy as Local Constraints Strengthening (LCS) heuristic one.

The local constraint generators are also given in Figure 3, where $\mathrm{Exam}(w, k)$ denotes the global constraint generator for positive examples. Different from the global constraint generator, the positive example constraint generator $\mathrm{Pos_a}$ does not ensure that there exists a path in the k-OA for each word w in the positive set S^+, but ensures that every step of a possible path for w exists in k-OA. This is clearly a weaker condition. To ensure the soundness, we enhance our constraints with another generator $\mathrm{Pos_b}$. $\mathrm{Pos_b}$ checks that if a step of a possible path for $w \in S^+$ exists in the k-OA, then its next 2-successive-steps exists as well. The condition is a complement of $\mathrm{Pos_a}$, and the composition of $\mathrm{Pos_a}$ and $\mathrm{Pos_b}$ is a stronger condition. Take the word $w = abcade$ for example. $\mathrm{Pos_b}(abcade, k)$ requires that both 2-successive-steps bc and de should be next to every state labelled by a in the k-OA. While the negative example constraint generator $\mathrm{Neg}(w, k)$ requires that for each step of a possible path for $w \in S^-$, none of its next 2-successive-steps in the path exists in the k-OA. The condition is a stronger one as well. Note that only one of the next 2-successive-steps does not exist in the k-OA may be sufficient. For example, if *abcade* is negative, then one of *bc* and *de* not following *a* is sufficient. But this is similar to the global constraint generator: there is a top-level negation. In brief, local constraint generators take 2 steps forward for each step. This step size 2 is set up based on our experience, which can guarantee the solvability of most problems and speed up the solution process.

The algorithm *fast*KOA$^{+}_{-}$ is shown in Algorithm 2, which first encodes our k-OA synthesis into a SAT via the constraint generators in Figure 3 (lines 1-6) and then solves the SAT problem if satisfiable (lines 7-11). If SAT is unsatisfiable, then the algorithm will invoke the exact version *i*KOA$^{+}_{-}$ (line 12), wherein the global constraint generator for examples are used instead. Note that the pruning step aims to delete useless states and edges in an automaton, especially the trap states, which will not affect the acceptances of the examples but could cause the procedure *Soa2Sore* used in *regex extraction* to fail.

We remark that our algorithm is flexible. From the synthesis process, it can be seen that our algorithm not only supports regex synthesis from positive and negative examples, but it can also synthesize regular expressions from positive (or negative) examples only by removing the corresponding constraints of negative (or positive) examples. Moreover, our algorithm is incremental, since one can synthesize a correct regex by feeding more and more examples gradually.

Algorithm 2: *fast*KOA$^{+}_{-}$

```
   Input: a positive set $S^+$, a negative set $S^-$, a value $k$
   Output: a deterministic $k$-OA $\mathcal{A}$ or null
1  initialize the formula set D ← ∅
2  add Deter($k$) to D
3  for $w \in S^+$ do
4      add $\mathrm{Pos_a}(w,k) \wedge \mathrm{Pos_b}(w,k)$ to D
5  for $w \in S^-$ do
6      add $\mathrm{Neg}(w,k)$ to D
7  Put D in a SAT solver
8  if D is satisfiable then
9      convert Boolean variables (matrix) to a $k$-OA $\mathcal{A}$
10     $\mathcal{A}$ ← prune $\mathcal{A}$ w.r.t. $S^+$ and $S^-$
11     return $\mathcal{A}$
12 else return $i$KOA$^{+}_{-}$($S^+$, $S^-$, $k$);
```

4.2 Regex Extraction

In this section, we present our procedure *GenRegex* to extract a regex from a marked k-OA. As mentioned in Section 2, a marked k-OA is also an SOA. Freydenberger and Kotzing [24] have proposed an efficient procedure *Soa2Sore* to convert a given SOA $\mathcal{A}$ to an SORE r that minimally generalizes $L(\mathcal{A})$. This procedure recurses first on all strongly connected looped components and then on the directed acyclic graph obtained by contracting all Strongly Connected Components (SCCs) (i.e., a maximal subgraph in which every node is reachable from every other node) with pluses "+". Here we use this procedure to extract regex from a marked k-OA.

The procedure *GenRegex* proceeds as follows: *GenRegex* first invokes *Soa2Sore*($\mathcal{A}$) to convert a marked k-OA $\mathcal{A}$ into an SORE (*i.e.*, a marked regex) r, then drops off the marks of all symbols in r, and finally returns the unmarked regex if it is deterministic[4], or returns *null* otherwise.

Further, we adopted a *regex rewriting* step to prettify the synthesized regex r in order to get a more concise and practical one by performing on r the rewriting rules in Figure 4 until no more rule is applicable.

Finally, as shown in Algorithm 1, our algorithm may return *null* (*i.e.*, fail), which is mainly due to (i) the parameter k may not be sufficiently large; and (ii) not every regex has an equivalent deterministic version, as deterministic regexes are a restricted subclass of regexes.

5 REPAIR ALGORITHM

In practice, one may write an incorrect or ReDoS-vulnerable regex. So, in the section, we present an algorithm based on Neighborhood

[4]The determinism of a regex can be decided in linear time [26].

(1) Determinism Constraint Generator:

$$\mathrm{Deter}(k) = \bigwedge_{\alpha\in\Sigma}\Big\{\Big[\bigvee_{1\le m\le k}\big(A_{src,\alpha_m}\wedge\bigwedge_{1\le n\le k,n\ne m}\neg A_{src,\alpha_n}\big)\Big]\vee\big(\bigwedge_{1\le m\le k}\neg A_{src,\alpha_m}\big)\Big\} \;\wedge\; \bigwedge_{\alpha\in\Sigma,\beta\in\Sigma}\bigwedge_{1\le t\le k}\Big\{\Big[\bigvee_{1\le m\le k}\big(A_{\alpha_t,\beta_m}\wedge\bigwedge_{1\le n\le k,n\ne m}\neg A_{\alpha_t,\beta_n}\big)\Big]\vee\big(\bigwedge_{1\le m\le k}\neg A_{\alpha_t,\beta_m}\big)\Big\}$$

(2) Positive Example Constraint Generators:

$$\mathrm{Pos_a}(w,k) = \begin{cases} A_{src,snk} & |w| = 0\\ \bigvee_{1\le m\le k}\big(A_{src,w[1]_m}\wedge A_{w[1]_m,snk}\big) & |w| = 1\\ \bigvee_{1\le m_1\le k} A_{src,w[1]_{m_1}}\wedge\bigwedge_{1\le i\le|w|-1}\big(\bigvee_{1\le m\le k,1\le n\le k} A_{w[i]_m,w[i+1]_n}\big)\wedge\bigvee_{1\le m_{|w|}\le k} A_{w[|w|]_{m_{|w|}},snk} & |w|\ge 2 \end{cases}$$

$$\mathrm{Pos_b}(w,k) = \begin{cases} \mathrm{T} & |w| < 2\\ \bigwedge_{1\le m_1\le k}\Big[A_{src,w[1]_{m_1}}\rightarrow\bigwedge_{1\le m\le k}\big(A_{w[1]_{m_1},w[2]_m}\wedge A_{w[2]_m,snk}\big)\Big] & |w| = 2\\ \Big[\bigwedge_{1\le m_1\le k}\big(A_{src,w[1]_{m_1}}\rightarrow\bigvee_{1\le m\le k}A_{w[1]_{m_1},w[2]_m}\wedge\big(\bigvee_{1\le n\le k}A_{w[2]_m,w[3]_n}\big)\big)\Big]\wedge\Big[\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[1]_m,w[2]_n}\rightarrow\bigvee_{1\le t\le k}\big(A_{w[2]_n,w[3]_t}\wedge A_{w[3]_t,snk}\big)\big)\Big] & |w| = 3\\ \begin{cases}\Big[\bigwedge_{1\le m_1\le k}\big(A_{src,w[1]_{m_1}}\rightarrow\bigvee_{1\le m\le k}A_{w[1]_{m_1},w[2]_m}\wedge\big(\bigvee_{1\le n\le k}A_{w[2]_m,w[3]_n}\big)\big)\Big]\wedge\Big[\bigwedge_{1\le i\le|w|-3}\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[i]_m,w[i+1]_n}\rightarrow\\ \bigvee_{1\le t\le k}A_{w[i+1]_n,w[i+2]_t}\wedge\big(\bigvee_{1\le o\le k}A_{w[i+2]_t,w[i+3]_o}\big)\big)\Big]\wedge\Big[\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[|w|-2]_m,w[|w|-1]_n}\rightarrow\bigvee_{1\le t\le k}\big(A_{w[|w|-1]_n,w[|w|]_t}\wedge A_{w[|w|]_t,snk}\big)\big)\Big]\end{cases} & |w| > 3 \end{cases}$$

(3) Negative Example Constraint Generators:

$$\mathrm{Neg}(w,k) = \begin{cases} \neg\mathrm{Exam}(w,k) & |w|\le 2\\ \Big[\bigwedge_{1\le m\le k}\big(A_{src,w[1]_m}\rightarrow\neg\big(\bigvee_{1\le n\le k}A_{w[1]_m,w[2]_n}\wedge\big(\bigvee_{1\le t\le k}A_{w[2]_n,w[3]_t}\big)\big)\big)\Big]\wedge\Big[\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[1]_m,w[2]_n}\rightarrow\neg\big(\bigvee_{1\le t\le k}\big(A_{w[2]_n,w[3]_t}\wedge A_{w[3]_t,snk}\big)\big)\big)\Big] & |w| = 3\\ \begin{cases}\Big[\bigwedge_{1\le m\le k}\big(A_{src,w[1]_m}\rightarrow\neg\big(\bigvee_{1\le n\le k}A_{w[1]_m,w[2]_n}\wedge\big(\bigvee_{1\le t\le k}A_{w[2]_n,w[3]_t}\big)\big)\big)\Big]\wedge\Big[\bigwedge_{1\le i\le|w|-3}\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[i]_m,w[i+1]_n}\rightarrow\\ \neg\big(\bigvee_{1\le t\le k}A_{w[i+1]_n,w[i+2]_t}\wedge\big(\bigvee_{1\le o\le k}A_{w[i+2]_t,w[i+3]_o}\big)\big)\big)\Big]\wedge\Big[\bigwedge_{1\le m\le k}\bigwedge_{1\le n\le k}\big(A_{w[|w|-2]_m,w[|w|-1]_n}\rightarrow\neg\big(\bigvee_{1\le t\le k}\big(A_{w[|w|-1]_n,w[|w|]_t}\wedge A_{w[|w|]_t,snk}\big)\big)\big)\Big]\end{cases} & |w| > 3 \end{cases}$$

Figure 3: Determinism Constraint Generator and Local Constraint Generators

$$\begin{array}{rlrl} c_1|\ldots|c_n & \Longrightarrow [c_1\text{-}c_n] & \texttt{[0-9]} & \Longrightarrow \texttt{\textbackslash d}\\ \texttt{[A-Za-z0-9\_]} & \Longrightarrow \texttt{\textbackslash w} & \texttt{[\textbackslash t\textbackslash r\textbackslash n\textbackslash f]} & \Longrightarrow \texttt{\textbackslash s}\\ \underbrace{r_1\ldots r_1}_{k\text{ times}} & \Longrightarrow r_1\{k\} & \underbrace{r_1\ldots r_1}_{k\text{ times}}\, r_1? & \Longrightarrow r_1\{k,k+1\}\\ \underbrace{r_1\ldots r_1}_{k\text{ times}}\, r_1^* & \Longrightarrow r_1\{k,\} & \underbrace{r_1\ldots r_1}_{k\text{ times}}\, r_1^+ & \Longrightarrow r_1\{k+1,\} \end{array}$$

Figure 4: Rewriting Rules for Regex

Search (NS) to repair such an incorrect or ReDos-vulnerable regex to achieve the same goals as our synthesis algorithm. In particular, given an incorrect or ReDos-vulnerable regex r and sets of positive and negative examples, it returns a ReDoS-invulnerable regex r' that is consistent with the examples.

We assume that the given regex is close to a solution. So the key idea of our algorithm is to search for a better solution, i.e., one that accepts more positive samples and/or rejects more negative samples, from the neighborhoods of a candidate k-OA. To start with, we define the *neighborhood* and an evaluation criterion of a k-OA $\mathcal{A}$. Given a k-OA $\mathcal{A}$, i.e., a Boolean matrix, we define its *neighborhood*, denoted as $N(\mathcal{A})$, as the set of k-OAs, which can be obtained by flipping one Boolean value of $\mathcal{A}$. In order to select a k-OA among a set of k-OAs, we define a measure f on k-OA with respect to S^+ and S^- as

$$f(\mathcal{A},S^+,S^-) = \frac{|T_P|-|F_N|+|T_N|-|F_P|}{|S^+|+|S^-|}$$

where we have $T_P = \{w\in S^+ \mid w\in L(\mathcal{A})\}$, $F_N = \{w\in S^+ \mid w\notin L(\mathcal{A})\}$, $T_N = \{w\in S^- \mid w\notin L(\mathcal{A})\}$, $F_P = \{w\in S^- \mid w\in L(\mathcal{A})\}$, and $|S|$ is the number of elements in the set S. Intuitively, the higher the f value, the better the k-OA. Especially, the k-OA with f value 1 will accept all positive samples and reject all negative samples. In addition, if two k-OAs share the same f value, we will select the one with fewer SCCs (or loops). Moreover, similar to regex synthesis, we use deterministic regexes to avoid ReDoS-vulnerabilities.

The repair algorithm is shown in Algorithm 3. It starts with a deterministic k-OA $\mathcal{A}$ that is converted from the marked version of the given regex r (line 1). Then, the algorithm selects the k-OA with the maximum f value, denoted as $\mathcal{A}_{\max}$, from the neighborhoods of $\mathcal{A}$ (line 4). Next, it compares the f values between the current k-OA and the selected k-OA. If the selected k-OA gets a higher f value, then the algorithm replaces the current k-OA by the deterministic version of the selected k-OA (lines 5-6). After that, it checks whether the current k-OA is satisfied, namely, the f value is 1. If it is, the algorithm returns a regex extracted from the pruned version of the current k-OA (lines 7-10). While if the selected k-OA gets a lower f value, then the current k-OA may be a local maximum, so the algorithm returns *null*[5] (line 12). The algorithm repeats the processing above until a satisfactory k-OA is returned or the iteration number exceeds ITER_MAX (lines 3-12). Finally, the algorithm returns *null* if the iteration number exceeds ITER_MAX (line 13).

A k-OA can be obtained easily from the marked form $\overline{r}$ of r: (i) each symbol in $\overline{r}$ forms a state of k-OA; (ii) for each symbol a, there is an edge from a to any symbol that follows a in $\overline{r}$; (iii) there is an edge from *src* to any symbol that may be matched first in $\overline{r}$, similar

[5]Enlarging the neighborhood is a possible but ineffective approach to search a better solution.

Algorithm 3: RepairingRE

```
**Input:** a regex $r$, a positive set $S^+$, a negative set $S^-$
**Output:** a deterministic regex $r$ with $S^+ \subseteq L(r)$ and
        $S^- \cap L(r) = \varnothing$ if solvable or *null* otherwise
$\mathcal{A} \leftarrow$ DISAMBIGUATE(RE2KOA($\overline{r}$), $S^+$, $S^-$);
$i \leftarrow 0$;
while $i < ITER\_MAX$ do
    $\mathcal{A}_{max} \leftarrow \arg\max_{\mathcal{A}' \in N(\mathcal{A})} f(\mathcal{A}', S^+, S^-)$;
    if $f(\mathcal{A}_{max}, S^+, S^-) > f(\mathcal{A}, S^+, S^-)$ then
        $\mathcal{A} \leftarrow$ DISAMBIGUATE($\mathcal{A}_{max}$, $S^+$, $S^-$);
        if $f(\mathcal{A}, S^+, S^-) == 1$ then
            $\mathcal{A} \leftarrow$ prune $\mathcal{A}$ w.r.t. $S^+$ and $S^-$;
            $r \leftarrow GenRegex(\overline{\mathcal{A}})$;
            return $r$;
        $i \leftarrow i + 1$;
    else return *null*;
return *null*;
```

to *sink*. Then a deterministic version can be obtained by removing the superfluous edges via the procedure DISAMBIGUATE given in Algorithm 4.

Algorithm 4: DISAMBIGUATE

```
**Input:** a $k$-OA $\mathcal{A}$, a positive set $S^+$, a negative set $S^-$
**Output:** a deterministic $k$-OA $\mathcal{A}$
while $\exists s \in \mathcal{A}.V$ and $\sigma \in \Sigma.\ |out(v, \sigma)| > 1$ do
    $C \leftarrow \emptyset$;
    for $t \in out(v, \sigma)$ do
        $\mathcal{A}' \leftarrow \mathcal{A}$;
        delete $(s, t')$ in $\mathcal{A}'$ for all $t' \in out(v, \sigma) \setminus \{t\}$;
        add $\mathcal{A}'$ in $C$;
    $\mathcal{A} \leftarrow \arg\max_{\mathcal{A}' \in C} f(\mathcal{A}', S^+, S^-)$;
return $\mathcal{A}$;
```

Note that character classes can be treated as special symbols such that all (special) symbols are pairwise disjoint.

In addition, the repair algorithm is incremental as well. The user may think that if she keeps on repairing the regex with some more interesting positive or negative examples, then she could finally get the equivalent one that she wants. However, the fact is that our neighborhood search may not find such a solution in a limited number of iterations, if the given examples are too many or the given regex is far away from a solution. Nevertheless, in that case, we would strongly suggest the user to use the synthesis algorithm instead.

6 EVALUATION

We implemented FlashRegex based on Z3 [16] SMT solver in Python, and conducted experiments on a machine with 16 cores Intel Xeon CPU E5620 @ 2.40GHz with 12MB Cache, 24GB RAM, running Windows 10 operating system. Our open-source implementation and the datasets of the experiments are available online[6]. The experiments were designed to study three sets of research questions concerning FlashRegex's functionalities (*i.e.*, synthesizing regexes, repairing incorrect regexes, and repairing SL regexes).

Table 1: Benchmarks

Task	Benchmark	Number	Sources	Description
Synthesis	Bin-Syn-Regex	50	AlphaRegexPublic [32] (25) AutoTutor [13] (25)	- Alphabet size is binary
	Multi-Syn-Regex	50	Regex Golf [17] (17) RegExLib [45] (33)	- Alphabet size is large
Repair (Incorrect)	Pos-Rep-Regex	50	Rebele et al. [44] (50)	- Incorrect regex - Positive examples only
	Pos-Neg-Rep-Regex	2,129	RegExLib [45] (25) AutoTutor [13] (2,104)	- Incorrect regex - Positive and negative examples
Repair (SL)	SL-Regex	20	OWASP (3), StackOverflow (1) snyk (1), RegExLib [45] (3) Davis et al. [14] (3), CVE (9)	- ReDoS-vulnerable regex - Positive and negative examples

RQ1. Evaluation of regex synthesis. Can FlashRegex synthesize regexes efficiently? (§6.2.1) Can FlashRegex synthesize regexes from examples correctly? (§6.2.2) Can FlashRegex synthesize safe regexes that are free from ReDoS-vulnerabilities? (§6.2.3) Can the local constraints speed up the solution process? (§6.2.4)

RQ2. Evaluation of incorrect regex repair. Can FlashRegex repair incorrect regexes efficiently? (§6.3.1) Can FlashRegex repair incorrect regexes from examples correctly? (§6.3.2) Can FlashRegex repair incorrect regexes so that they are free from ReDoS-vulnerabilities? (§6.3.3)

RQ3. Evaluation of SL regex repair. Can FlashRegex repair SL regexes efficiently? (§6.4.1) Can FlashRegex repair SL regexes from examples correctly? (§6.4.2) Can FlashRegex repair SL regexes so that they are free from ReDoS-vulnerabilities after repair? (§6.4.3)

6.1 Benchmarks and Existing Tools

To evaluate FlashRegex in three application scenarios under different prerequisites, we constructed ten separate benchmarks accordingly, collected from widely-used sources: (i) Regex Golf [17], (ii) AlphaRegexPublic [32], (iii) AutoTutor dataset [13], (iv) the RegExLib library [45], (v) Rebele et al. [44], (vi) OWASP[7], (vii) StackOverflow[8] (viii) snyk[9], (ix) Davis et al. [14], and (x) Common Vulnerabilities and Exposures (CVE)[10]. Some information of them is given in Table 1.

We compared FlashRegex with five state-of-the-art tools, *GP-RegexGolf* [4], *AlphaRegex* [32], *RegexGenerator++* [3, 5], *RFixer* [39], and the one recently proposed by *Rebele et al.* [44]. Among them, *GP-RegexGolf*, *AlphaRegex*, and *RegexGenerator++* were developed for regex synthesis, while the remaining two for regex repair. Furthermore, we used three tools to detect ReDoS-vulnerabilities, ReScue [47], Rexploiter [59] and SDLFuzzer [53] and manually checked all inconsistent results concluded by these tools.

[6]https://github.com/EasyRegex/FlashRegex
[7]https://www.owasp.org/
[8]https://stackoverflow.com/
[9]https://snyk.io/blog/redos-and-catastrophic-backtracking/
[10]https://cve.mitre.org/

Table 2: The effectiveness and efficiency of regex synthesis

Benchmarks	Bin-Syn-Regex				Multi-Syn-Regex			
Technique	#Sol (%)	#CSol (%)	#Vul	Avg. Time (s)	#Sol (%	#CSol (%)	#Vul	Avg. Time (s)
RegexGenerator++	-	-	-	-	50 (100%)	3 (6%)	0	198.0
GP-RegexGolf	-	-	-	-	50 (100%)	7 (14%)	4	3889.6
AlphaRegex	50 (100%)	50 (100%)	21	7.6	-	-	-	-
FlashRegex-Exact	50 (100%)	50 (100%)	0	3.3	38 (76%)	38 (100%)	0	5.3
FlashRegex-LCS	36 (72%)	36 (100%)	0	1.1	29 (58%)	29 (100%)	0	3.4
FlashRegex	50 (100%)	50 (100%)	0	1.9	38 (76%)	38 (100%)	0	4.0

6.2 Evaluation of Regex Synthesis

In this experiment, we evaluated the effectiveness and efficiency of regex synthesis, as well as its comparison with the exact-directed (FlashRegex-Exact) and local-directed (FlashRegex-LCS) constraint encodings. The evaluation results are shown in Table 2. The experiments were conducted on two benchmarks, *Bin-Syn-Regex* and *Multi-Syn-Regex*, measuring the performance on benchmarks with different alphabet sizes. Columns in each benchmark represent the number of synthesized regexes by each tool (#Sol), the number of correctly synthesized ones by each tool (i.e., the ones that are consistent with all the examples) (#CSol), the number of ReDoS-vulnerable ones in the synthesized results (#Vul) and the average time of synthesizing one regex in seconds (Avg Time), respectively. The symbol "-" indicates that the tool does not support the corresponding benchmark functionally.

6.2.1 *Efficiency of Regex Synthesis.*

Table 2 compares the runtime efficiency of FlashRegex with three synthesis tools by their average time (in seconds) taken to synthesize a regex on the corresponding benchmark. On benchmark *Bin-Syn-Regex* with binary alphabet size, though the differences in average time are small (ranging from 1.1 to 7.6 seconds), the average runtime of FlashRegex (1.9 seconds) is still less than half of that of AlphaRegex (7.6 seconds). On benchmark *Mul-Syn-Regex* with multiple alphabet sizes, the average time varies greatly, from one hour to few seconds. Note that the time in Table 2 refers to average time, so the total running time taken by RegexGenerator++ and GP-RegexGolf on this benchmark are more than 2 hours and 50 hours, respectively, which is considered unaffordable in practice. In contrast, FlashRegex (using either of the synthesis algorithms) only takes an average 4 seconds, which is significantly more efficient than the other tools.

6.2.2 *Correctness of Regex Synthesis.*

The results on correctness are summarized by column #CSol in Table 2. The correctness of synthesized regex is crucial. It guarantees the quality of regexes after synthesis. The experiments show that the correctness of existing tools is unsatisfactory. We can see that though RegexGenerator++ and GP-RegexGolf can synthesize all the 50 regexes on the second benchmark, only 3 and 7 of them are correct. It is mainly because they require the synthesized regex to be consistent with as many given examples as possible, instead of all the examples. By contrast, FlashRegex and AlphaRegex achieve 100% correctness ratio, because they aim to find a regex that accepts all the positive examples while rejecting all the negative examples.

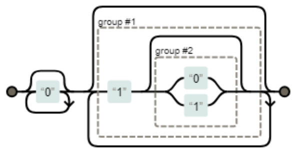


Figure 5: `0*(1(0|1)?)*`

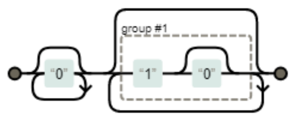


Figure 6: `0*(10?)*`

6.2.3 *ReDoS-vulnerability of Regex Synthesis.*

The two #Vul columns in Table 2 show that 4 and 21 regexes synthesized by GP-RegexGolf and AlphaRegex, respectively, are ReDoS-vulnerable. In other words, the consistency with given examples does not necessarily guarantee the synthesized regexes free from ReDoS-vulnerability. In contrast, all the regexes synthesized by FlashRegex are ReDoS-invulnerable.

Let us give an example from *Bin-Syn-Regex* to illustrate the difference between ReDoS-vulnerable and ReDoS-invulnerable regexes. Given the same set of positive and negative examples, AlphaRegex synthesizes `0*(1(0|1)?)*`, while FlashRegex synthesizes `0*(10?)*`. Although the two regexes are equivalent, their evaluations take exponential and linear time in the length of the input, respectively. Let us consider the railroad diagrams of these two regexes, which are given in Figures 5 and 6. Suppose the given string is "11"; there are two possible paths in Figure 5 to generate this string, while there is only one path in Figure 6. When the given string is "111", there are four possible paths in Figure 5, whereas there is still one possible path in Figure 6. This illustrates how the two search spaces differ as the length of the input increases.

6.2.4 *Evaluation of Different Constraint Encodings.*

Table 2 shows that FlashRegex-Exact is more effective than FlashRegex-LCS in synthesizing significantly more regexes. On the other hand, FlashRegex-LCS is more efficient by leveraging the LCS heuristic strategy. There are 14 (9) benchmarks on *Bin-Syn-Regex* (*Multi-Syn-Regex*) that FlashRegex-Exact can solve while FlashRegex-LCS cannot. FlashRegex-Exact is, on average, 2.0X slower than FlashRegex-LCS on *Bin-Syn-Regex*, and 0.6X slower on *Multi-Syn-Regex*. The two constraint encodings are complementary on effectiveness and efficiency.

Summary to RQ1: FlashRegex can synthesize regex efficiently, correctly and safely. The advantage of high efficiency of FlashRegex becomes more obvious with the increase of the alphabet size. Also, the two constraint encodings of FlashRegex are complementary on effectiveness and efficiency. The results also confirmed the lack of focus on ReDoS-vulnerability in previous works, thus making further repair a necessity.

6.3 Evaluation of Incorrect Regex Repair

Table 5 shows the evaluation results of repairing incorrect regexes in terms of efficiency, correctness and ReDoS-vulnerability. It uses the same columns as Table 2. The experiments were conducted on *Pos-Rep-Regex* and *Pos-Neg-Rep-Regex* benchmarks. For comparison, the tools proposed by Rebele et al [44] and RFixer were evaluated. Note that the regexes under repair in the evaluation are incorrect.

Table 3: ReDoS-vulnerable Regexes Repaired by RFixer and FlashRegex. RV denotes ReDoS-vulnerable.

No.	Source	SL (Sub-)Regex	RFixer			FlashRegex		
			Repaired (Sub-)Regex	Time (s)	RV	Repaired (Sub-)Regex	Time (s)	RV
#1	OWASP	(a|aa)+	(a|aa)+	0.098	V	a+	0.596	I
#2	OWASP	(a|a?)+	(a?)+	0.133	I	a*	0.028	I
#3	OWASP	([a-zA-Z]+)*	([a-zA-Z]+)*	0.057	V	([a-zA-Z])*	0.059	I
#4	StackOverflow	(x+x+)+y	(x+)+y	10.289	V	xx+y	0.183	I
#5	snyk	(\w+\d+)+C	(\w+\d+)+C	0.176	V	([A-Za-z_]*\d)+C	0.058	I
#6	RegExLib	(\d+(,\d+)*)+	(\d+(,\d+)*)+	0.196	V	\d+(,\d+)*	0.427	I
#7	RegExLib	([0-9a-f]+\d+)*	([0-9a-f]+\d+)*	0.204	V	(([a-f]+\d)|\d)*	0.574	I
#8	RegExLib	(\d+|(\d*\.\d+))+	(\d+|(\d*\.\d+))+	0.158	V	(\.?\d)+	0.040	I
#9	Davis et al. [14]	\s*#?\s*	\s*#?\s*	0.139	V	\s*(#\s*)?	0.249	I
#10	Davis et al. [14]	(\n\s*)+	(\n\s*)+	0.004	V	\n\s*	0.052	I
#11	Davis et al. [14]	[$_a-z]+[$_a-z0-9-]*	[$_a-z]+[$_a-z0-9-]*	0.003	V	[$_a-z][$_a-z0-9-]*	0.061	I
#12	CVE-2009-3277	((a{1,2}){1,2}){1,10}	((a{1,2}){1,2}){1,10}	15.763	V	a{1,40}	8.555	I
#13	CVE-2016-4055	A(B|C+)+D	A(B+|C+)+D	0.162	V	A(B|C)+D	0.063	I
#14	CVE-2017-15010	([^=;]+)\s*=\s*([^\n\r\0]*)	([^=;\s]+)\s*=\s*([^\s\0]*)	31.534	I	([^=;\s]+)\s*=\s*([^\s\0]*)	5.484	I
#15	CVE-2017-16098	\s*=\s*['"]? *([\w\-]+)	\s*=\s*['"]? *([\w\-]+)	3.218	V	\s*=\s*(['"] *)?([\w\-]+)	20.125	I
#16	CVE-2017-16137	\s*\n\s*	[\f\r\t\v]*\n\s*	0.368	I	[\f\r\t\v]*\n\s*	0.543	I
#17	CVE-2017-18214	(\s*?[\u0600-\u06FF]+){1,2}	(\s*?[\u0600-\u06FF]+){1,2}	3.270	V	\s*[\u0600-\u06FF]+(\s+[\u0600-\u06FF]+)?	23.074	I
#18	CVE-2018-3737	([\n \t]+([^\n]+))?	([\n \t]+([^\n]+))?	183.469	V	([\n \t]+([^\n \t]+))?	0.375	I
#19	CVE-2019-17592	(\-|\+)?([1-9]+[0-9]*)	(\-|\+)?([1-9]+[0-9]*)	15.936	V	(\-|\+)?[1-9]\d*	4.305	I
#20	CVE-2020-5243	*([^;]+) *	*([^;]+) *	1.406	I	*([^;]+) *	2.102	I

Table 4: ReDoS-vulnerable Regexes Repaired by Experts and FlashRegex. RV denotes ReDoS-vulnerable.

Source	SL Regex	Expert			FlashRegex		
		Strategy	Fix Solution	RV	Strategy	Fix Solution	RV
CVE-2016-4055	A(B|C+)+D	Revise	A(?=(B|C+))\1+D	I	Revise	A(B|C)+D	I
CVE-2017-15010	([^=;]+)\s*=\s*([^\n\r\0]*)	Revise	(([^=;]+))\s{0,512}=\s*([^\n\r\0]*)	V	Revise	([^=;\s]+)\s*=\s*([^\s\0]*)	I
CVE-2017-16098	\s*=\s*['"]? *([\w\-]+)	Trim	limit match string, only allow max 10 spaces and 100 charset string	-	Revise	\s*=\s*(['"] *)?([\w\-]+)	I
CVE-2017-16137	\s*\n\s*	Resort	(i) split the line by '\n', and (ii) trim each line, (iii) then join each line by ' '	-	Revise	[\f\r\t\v]*\n\s*	I
CVE-2017-18214	(\s*?[\u0600-\u06FF]+){1,2}	Revise	(\s*?[\u0600-\u06FF]{1,256}){1,2}	V	Revise	\s*[\u0600-\u06FF]+(\s+[\u0600-\u06FF]+)?	I
CVE-2018-3737	([\n \t]+([^\n]+))?	Revise	([\n \t]+([^\n \t][^\n]*))?	I	Revise	([\n \t]+([^\n \t]+))?	I
CVE-2019-17592	(\-|\+)?([1-9]+[0-9]*)	Revise	(\-|\+)?[1-9][0-9]*	I	Revise	(\-|\+)?[1-9]\d*	I
CVE-2020-5243	*([^;]+) *	Revise	{0,2}([^;]+) {0,2}	I	Revise	*([^;]+) *	I

They either reject some positive examples or accept some negative ones.

Table 5: The effectiveness and efficiency of incorrect regex repair

Benchmarks	Pos-Rep-Regex				Pos-Neg-Rep-Regex			
Technique	#Sol (%)	#CSol (%)	#Vul	Avg. Time (s)	#Sol (%	#CSol (%)	#Vul	Avg. Time (s)
Rebele et al [44]	50 (100%)	50 (100%)	0	0.2	-	-	-	-
RFixer	35 (70%)	35 (100%)	3	2.4	1,611 (75.67%)	1,611 (100%)	349	9.3
FlashRegex	35 (70%)	35 (100%)	0	1.5	1,948 (91.50%)	1,948 (100%)	0	1.6

6.3.1 ***Efficiency of Incorrect Regex Repair.*** The tool proposed by Rebele et al. [44] ran the fastest (0.2 seconds) on the first benchmark (*i.e.*, the one from the tool), but it cannot handle negative examples. RFixer took the longest average time on both benchmarks. It took much more average time on the given negative examples. In constrast, the efficiency of FlashRegex was mildly affected by the negative samples. The average time it took increased mildly from 1.5 to 1.6 seconds.

6.3.2 ***Correctness of Incorrect Regex Repair.*** Table 5 shows that the tool proposed by Rebele et al. [44] can repair the most number of regexes on the first benchmark, but it is not able to process any negative examples on the second benchmark. FlashRegex can repair 91.5% of the incorrect regexes, 337 (15.83%) more regexes than those repaired by RFixer, on the second benchmark.

6.3.3 *ReDoS-vulnerability of Incorrect Regex Repair.*

According to the #Vul coulmn in Table 5, three (8.57%) ReDoS-vulnerable regexes were generated by RFixer on the first benchmark. The number increases significantly on the second benchmark. There were 349 (21.66%) regexes repaired by RFixer suffered from ReDoS-vulnerability. In contrast, FlashRegex can repair 337 more regexes than RFixer with no ReDoS-vulnerable regexes generated.

Let us present a few incorrect regexes to illustrate the vulnerable and invulnerable repairs. From *Pos-Neg-Rep-Regex*, for the incorrect regex `(((0+1*)*0+)*|((1+0*)*1+)*)*`, a repaired regex produced by RFixer is `(0(0*|1+0)*)|(1(1*|0+1)*)`, which is ReDos-vulnerable. We can see that although this regex is consistent with the given examples, it still causes ambiguity because its star height (*i.e.*, nested quantifiers) equals to two. For example, the string "0" can be either consumed by the inner quantifier (*) or the outer one (*), leading to incorrect behavior with worst-case exponential costs on a mismatch. In contrast, FlashRegex deduced the regex `((01?)+|(10*)+)`, which successfully avoids the security issue. Similar incorrect ones repaired by RFixer have also been found from *Pos-Rep-Regex*, they are `(([1234567,][1234568,]|[79,]))*\d` and `(([a-z]|[A-Z])?[AegikLloPr3tuVwX ])+(\d+\.?)+`. Both are ReDoS-vulnerable.

Summary to RQ2: FlashRegex can repair incorrect regex efficiently, correctly and safely. The efficiency is not affected significantly by negative examples, and the regex after repair is free from ReDoS-vulnerability.

6.4 Evaluation of SL Regex Repair

The evaluation results of SL regex repair are shown in Table 3. We compared FlashRegex with RFixer, which is designed to repair regexes from both positive and negative examples. The benchmark contains 20 SL regexes (listed in Table 3) with positive and negative examples. Due to the space constraint, we only show the problematic sub-regexes for some regexes. Examples corresponding to these (sub-)regexes are generated manually or based on the Brics automaton library [37]. Specifically, the positive examples are enumerated by randomly traversing the deterministic finite automaton (DFA) of the given regex (resp. the negative examples are synthesized by stochastically traversing the DFA of the negation of the given regex). The columns in the table represent the sources of each regex (Source), the ReDoS-vulnerable regex (SL (Sub)-Regex); columns in RFixer and FlashRegex are the regex repaired by the corresponding tools (Repaired (Sub-)Regex), the running time in seconds (Time) and whether the repaired regexes are ReDoS-vulnerable (RV) (*V* for vulnerable, *I* for invulnerable).

6.4.1 *Efficiency of SL Regex Repair.*

The average running time taken for repairing varies across regexes. RFixer took from less than 0.057 to 183 seconds, and 13.329 seconds on average. FlashRegex offers a more stable and better efficiency, ranging from 0.04 to 23.074 seconds, and 3.348 seconds on average.

6.4.2 *Correctness of SL Regex Repair.*

In Table 3, all the repaired regexes are consistent with the given examples. This suggests the correctness of both tools. The results are also in line with the results of repairing incorrect regexes in Table 5.

6.4.3 *ReDoS-vulnerability of SL Regex Repair.*

According to Table 3, 16/20 (80%) regexes repaired by RFixer are still vulnerable to ReDoS attacks. In constrast, all the SL regexes repaired by FlashRegex are free from ReDoS-vulnerabilities.

We further compared the results repaired by FlashRegex with the results achieved by developer experts, which are given in Table 4[11]. We found that experts repaired SL regexes using one of three strategies introduced by Davis et al. [14]: revising the regex, trimming the input, or resorting it to alternate logic. The first one is motivated but not easy to do it manually, as demonstrated by the results, 2/6 (33%) regexes repaired by experts are still vulnerable to ReDoS attacks, and the invulnerable ones may be difficult for users to follow (*e.g.*, the first one using back-reference and the last one using bounded iteration). In contrast, all the SL regexes repaired by FlashRegex are free from ReDoS-vulnerabilities. The second one works well but is not friendly to users, while the last one is only effective for some special regexes (*i.e.*, it cannot be generalized).

Summary to RQ3: FlashRegex can repair ReDoS-vulnerable regex efficiently and correctly. The experiment also indicates the incapability of existing work for repairing ReDoS-vulnerable regex. Further, comparing with the manual repair, FlashRegex works in an automatic and user-friendly manner, keeping users' intention meanwhile getting rid of the ReDoS-vulnerability.

7 THREATS TO FLASHREGEX'S VALIDITY

Considering that FlashRegex is a programming-by-example (PBE) algorithm, the quality of regex synthesized by FlashRegex highly depends on the quality of examples. In other words, if users can not provide sufficient characteristic examples, the synthesized regexes will be unsatisfactory (i.e., over-fitting or under-fitting). To alleviate this problem, we can adopt the following three strategies:

Generalized examples. The users provide some abstract generalized examples rather than concrete examples. The generalized examples can reduce the amount of required concrete (characteristic) examples meanwhile lessening the workload from users, thereby improving the quality of regex synthesis by generalizing concrete examples. For instance, We can use a generalized example `<NUM>dog` instead of concrete examples (e.g., `1dog`, `2dog`, `3dog`).

Integration of PBE and programming by natural language (PBNL). We can leverage PBNL techniques to overcome the drawbacks of PBE techniques. Specifically, incorporating natural language can greatly improve the generalization of PBE techniques. For example, we can use the natural language description "*lines with vowels after lower-case letters*" as a major resource, and some concrete example (e.g., `biiii` and `cee`) instead of a large number of concrete examples.

Interaction with users. A more user-friendly strategy is that whenever the generated regex is out of expectation of users, they can add/delete/update examples interactively so that the resulting regex can be adjusted dynamically until meet users' requirements.

[11] Due to space constraint, only solutions from CVE are given.

8 RELATED WORK

Programming-by-Example (PBE). PBE techniques have been the subject of research in the past few decades [46] and successful paradigms for program synthesis, allowing end-users to construct and run new programs by providing examples of the intended program behavior [18]. Recently, PBE techniques have been successfully used for string transformations [27, 48, 49], data filtering [57], data structure manipulations [22, 60], table transformations [20, 28], SQL queries [56, 62], MapReduce programs [1, 50], and also regex synthesis [3–5, 7, 8, 21, 24, 32, 34]. In this paper, we take advantage of PBE techniques to enhance user-friendliness by allowing users to provide examples to reflect their true intentions.

ReDoS Detection and Prevention. Various techniques have been proposed [30, 41–43, 47, 52, 53, 58, 59] to identify ReDoS-vulnerabilities, which can be mainly classified into two paradigms: static analysis [30, 42, 43, 52, 58, 59] and dynamic fuzzing [41, 47, 53].

To prevent ReDoS attacks, Davis et al. [14] identified three anti-patterns of regexes as prerequisites of ReDoS attacks and recommended avoiding using them. These anti-patterns include: (i) regexes with *nested quantifiers* (e.g., `A(B|C+)+D`); (ii) Regexes with *Quantified Overlapping Disjunction (QOD)* (e.g., $(\backslash w|\backslash d)+$); and (iii) Regexes with *Quantified Overlapping Adjacency (QOA)* (e.g., `\s*#?\s*`). In our work, we adopt deterministic k-OA and deterministic regex constraints to avoid the ambiguity caused by QOD or QOA in FlashRegex algorithms, and searching for solutions starting from $k = 1$ can effectively avoid nested quantifiers.

ReDoS attacks can also be alleviated by regex matching speedup, which is possible in some special cases, e.g., by parallel algorithms [35], GPU-based algorithms [61], state-merging algorithms [6], and Thompson's Non-deterministic Finite Automaton algorithm [12, 54]. However, these can only alleviate the ReDoS-vulnerability problem, since the regexes themselves are still subject to ReDoS attacks. In our work, we address this issue from the regexes side.

Regexes Synthesis. The problem of automatic regex synthesis from examples has been explored in many domains [3–5, 7, 8, 21, 24, 31, 32, 34, 36, 40, 63]. AlphaRegex [32] is an enumeration algorithm for synthesizing simple regexes over binary alphabets from examples. However, all the synthesized expressions are over alphabets of size 2. *RegexGenerator++* [3, 5] is a state-of-the-art approach for the synthesis of regexes from examples. The fact that *RegexGenerator++* utilizes genetic programming means that it is not guaranteed to generate a correct solution–*i.e.*, accepting all the positive examples while rejecting all the negative examples. Lots of existing works focus on XML schemas inference [7, 8, 21, 24, 34], via resorting to infer regexes from examples. These approaches usually aim to tackle restricted forms of *deterministic* regexes [9] from positive examples only. *GP-RegexGolf* [4] is an approach based on genetic programming for playing *regex golf* [17] automatically, *i.e.*, for writing the shortest regex that matches all positive strings and does not match any negative string. Unlike many of these efforts which aim to generalize beyond the provided examples, *GP-RegexGolf* focuses on binary classifying input strings and it does not require a form of generalization, *i.e.*, the ability of inducing a general pattern from the provided examples. Several works from the Natural Language Processing community address the problem of generating regexes from natural language specifications based on sequence-to-sequence (seq2seq) model [31, 36, 40, 63].

All of the synthesis techniques above barely pay attention to preventing ReDoS during the synthesis process, making the synthesized regexes (hyper-)vulnerable to ReDoS attacks.

Regexes Repair. There are several works targeting at repairing or modifying regexes from examples, specifically the incorrect regexes. We discuss two main paradigms of them. In the first paradigm, works only consider either positive or negative examples. Li et al. [33] proposed ReLIE, which can modify complex regexes by rejecting the newly-input negative examples. By contrast, Rebele et al. [44] proposed a novel way to generalize a given regex so that it accepts the given positive examples. On the other hand, works in the second paradigm take both positive and negative examples into consideration. Pan et al. [39] designed RFixer, a tool for repairing incorrect regexes using both examples. It took advantage of skeletons of regexes to effectively prune out the search space, and it employed SMT solvers to efficiently explore the sets of possible character classes and numerical quantifiers. Our work is similar to their work, yet differs in the effectiveness and quality of repaired regexes — we consider not only the correctness, but also the ReDoS-invulnerability of the regexes.

There is only two work exploring how to repair ReDoS-vulnerable regexes [11, 55], which considers revisions that match the exact same languages of the original regexes. However, the (exact) equivalence is too strong to use in practice [14, 47].

9 CONCLUSION

Many techniques for synthesizing or repairing regexes have been proposed. However, the lack of attention to ReDoS-vulnerabilities affects the security of existing tools. We propose a PBE framework, FlashRegex, which provides three core functionalities including regex synthesis, incorrect regex repair, and ReDoS-vulnerable (i.e., SL) regex repair. This is achieved by devising novel algorithms to deduce deterministic regexes from both positive and negative examples based on SAT or NS. Ours is the first framework that integrates the synthesis and repair of regexes with the awareness of ReDoS-vulnerabilities. The evaluation results show that our work can effectively and efficiently generate anti-ReDoS regexes from given examples, and has better capability than existing repair tools and even human experts on ReDoS-vulnerable regex repair, demonstrating the usefulness of our work. The results also reveal that existing synthesis and repair tools have neglected ReDoS-vulnerabilities of regexes. Although our experiments have shown the effectiveness and efficiency of FlahsRegex, we plan to conduct a larger scale of evaluation with more complex subjects in our future work.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work is supported in part by National Natural Science Foundation of China (Grants #61872339, #61472405, #61932021, #61972260, #61772347, #61836005), NSF grant IIS-1633271, Guangdong Basic and Applied Basic Research Foundation under Grant #2019A1515011577, and Huawei PhD Fellowship, MSRA Collaborative Research Grant.

REFERENCES

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2016. Leveraging Parallel Data Processing Frameworks With Verified Lifting. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.* 67–83.

[2] Adam Baldwin. 2016. Regular Expression Denial Of Service Affecting Express.js. https://medium.com/node-security/

[3] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. 2014. Automatic Synthesis Of Regular Expressions From Examples. *IEEE Computer* 47, 12 (2014), 72–80.

[4] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2014. Playing Regex Golf With Genetic Programming. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014.* 1063–1070.

[5] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2016. Inference Of Regular Expressions For Text Extraction From Examples. *IEEE Trans. Knowl. Data Eng.* 28, 5 (2016), 1217–1230.

[6] Michela Becchi and Srihari Cadambi. 2007. Memory-Efficient Regular Expression Search Using State Merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA.* IEEE, 1064–1072.

[7] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. 2010. Learning Deterministic Regular Expressions For The Inference Of Schemas From XML Data. *TWEB* 4, 4 (2010), 14:1–14:32.

[8] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Stijn Vansummeren. 2010. Inference Of Concise Regular Expressions And DTDs. *ACM Trans. Database Syst.* 35, 2 (2010), 11:1–11:47.

[9] Anne Brüggemann-Klein. 1993. Unambiguity Of Extended Regular Expressions In SGML Document Grammars. In *Algorithms - ESA '93, First Annual European Symposium, Bad Honnef, Germany, September 30 - October 2, 1993, Proceedings.* 73–84.

[10] Carl Chapman, Peipei Wang, and Kathryn T. Stolee. 2017. Exploring Regular Expression Comprehension. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017.* 405–416.

[11] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. 2017. A Search For Improved Performance In Regular Expressions. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2017, Berlin, Germany, July 15-19, 2017.* 1280–1287.

[12] Russ Cox. 2007. Regular Expression Matching Can Be Simple And Fast (But Is Slow In Java, Perl, PHP, Python, Ruby, ...). https://swtch.com/~rsc/regexp/regexp1.html

[13] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs From Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017.* 582–592.

[14] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The Impact Of Regular Expression Denial Of Service (ReDoS) In Practice: An Empirical Study At The Ecosystem Scale. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* 246–256.

[15] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions A Lingua Franca? An Empirical Study On The Re-Use And Portability Of Regular Expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.* 443–454.

[16] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 337–340.

[17] Erling Ellingsen. 2019. Regex Golf. https://alf.nu/RegexGolf

[18] Kevin Ellis and Sumit Gulwani. 2017. Learning To Learn Programs From Examples: Going Beyond Program Structure. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017.* 1638–1645.

[19] Stack Exchange. 2016. Outage Postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016

[20] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis Of Table Consolidation And Transformation Tasks From Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017.* 422–436.

[21] Henning Fernau. 2009. Algorithms For Learning Regular Expressions From Positive Data. *Inf. Comput.* 207, 4 (2009), 521–541.

[22] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations From Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015.* 229–239.

[23] Python Software Foundation. 2018. PyPI–The Python Package Index. https://pypi.org/

[24] Dominik D. Freydenberger and Timo Kötzing. 2015. Fast Learning Of Restricted Regular Expressions And DTDs. *Theory Comput. Syst.* 57, 4 (2015), 1114–1158.

[25] E. Mark Gold. 1978. Complexity Of Automaton Identification From Given Data. *Information and Control* 37, 3 (1978), 302–320.

[26] Benoît Groz and Sebastian Maneth. 2017. Efficient Testing And Matching Of Deterministic Regular Expressions. *J. Comput. Syst. Sci.* 89 (2017), 372–399.

[27] Sumit Gulwani. 2011. Automating String Processing In Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011.* 317–330.

[28] William R. Harris and Sumit Gulwani. 2011. Spreadsheet Table Transformations From Examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* 317–328.

[29] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2019. Regexes Are Hard: Decision-Making, Difficulties, And Risks In Programming Regular Expressions. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* 415–426.

[30] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis For Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings.* 135–148.

[31] Nate Kushman and Regina Barzilay. 2013. Using Semantic Unification To Generate Regular Expressions From Natural Language. In *Human Language Technologies: Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, June 9-14, 2013, Westin Peachtree Plaza Hotel, Atlanta, Georgia, USA.* 826–836.

[32] Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions From Examples For Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2016, Amsterdam, The Netherlands, October 31 - November 1, 2016.* 70–80.

[33] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and H. V. Jagadish. 2008. Regular Expression Learning For Information Extraction. In *2008 Conference on Empirical Methods in Natural Language Processing, EMNLP 2008, Proceedings of the Conference, 25-27 October 2008, Honolulu, Hawaii, USA, A meeting of SIGDAT, a Special Interest Group of the ACL.* 21–30.

[34] Yeting Li, Xiaolan Zhang, Jialun Cao, Haiming Chen, and Chong Gao. 2019. Learning K-Occurrence Regular Expressions With Interleaving. In *Database Systems for Advanced Applications - 24th International Conference, DASFAA 2019, Chiang Mai, Thailand, April 22-25, 2019, Proceedings, Part II.* 70–85.

[35] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. 2011. Accelerating Regular Expression Matching Using Hierarchical Parallel Machines On GPU. In *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA.* IEEE, 1–5.

[36] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural Generation Of Regular Expressions From Natural Language With Minimal Domain Knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016.* 1918–1923.

[37] Anders Møller. 2017. dk.brics.automaton – Finite-State Automata and Regular Expressions for Java. https://www.brics.dk/automaton/

[38] Inc. NPM. 2018. NPM. https://www.npmjs.com/

[39] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. 2019. Automatic Repair Of Regular Expressions. *PACMPL* 3, OOPSLA (2019), 139:1–139:29.

[40] Jun-U. Park, Sang-Ki Ko, Marco Cognetta, and Yo-Sub Han. 2019. SoftRegex: Generating Regex From Natural Language Descriptions Using Softened Regex Equivalence. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019.* 6424–6430.

[41] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection Of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 2155–2168.

[42] Asiri Rathnayake. 2015. *Semantics, Analysis And Security Of Backtracking Regular Expression Matchers.* Ph.D. Dissertation. University of Birmingham, UK.

[43] Asiri Rathnayake and Hayo Thielecke. 2014. Static Analysis For Regular Expression Exponential Runtime Via Substructural Logics. *CoRR* abs/1405.7058

(2014).

[44] Thomas Rebele, Katerina Tzompanaki, and Fabian M. Suchanek. 2018. Adding Missing Words To Regular Expressions. In *Advances in Knowledge Discovery and Data Mining - 22nd Pacific-Asia Conference, PAKDD 2018, Melbourne, VIC, Australia, June 3-6, 2018, Proceedings, Part II*. 67–79.

[45] RegExLib. 2019. Regular Expression Library. http://regexlib.com/

[46] David E. Shaw, William R. Swartout, and C. Cordell Green. 1975. Inferring LISP Programs From Examples. In *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, September 3-8, 1975*. 260–267.

[47] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: Crafting Regular Expression DoS Attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 225–235.

[48] Rishabh Singh. 2016. BlinkFill: Semi-Supervised Programming By Example For Syntactic String Transformations. *PVLDB* 9, 10 (2016), 816–827.

[49] Rishabh Singh and Sumit Gulwani. 2012. Learning Semantic String Transformations From Examples. *PVLDB* 5, 8 (2012), 740–751.

[50] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 326–340.

[51] Eric Spishak, Werner Dietl, and Michael D. Ernst. 2012. A Type System For Regular Expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs, FTfJP 2012, Beijing, China, June 12, 2012*. 20–26.

[52] Satoshi Sugiyama and Yasuhiko Minamide. 2014. Checking Time Linearity Of Regular Expression Matching Based On Backtracking. *Information and Media Technologies* 9, 3 (2014), 222–232.

[53] Bryan Sullivan. 2010. New Tool: SDL Regex Fuzzer. https://cloudblogs.microsoft.com/microsoftsecure/2010/10/12/new-tool-sdl-regex-fuzzer

[54] Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[55] Brink van der Merwe, Nicolaas Weideman, and Martin Berglund. 2017. Turning Evil Regexes Harmless. In *Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017*. 38:1–38:10.

[56] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing Highly Expressive SQL Queries From Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466.

[57] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: Filtering Spreadsheet Data Using Examples. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. 195–213.

[58] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. 2016. Analyzing Matching Time Behavior Of Backtracking Regular Expression Matchers By Using Ambiguity Of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings*. 322–334.

[59] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection Of DoS Vulnerabilities In Programs That Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*. 3–20.

[60] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing Transformations On Hierarchically Structured Data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. 508–521.

[61] Xiaodong Yu and Michela Becchi. 2013. GPU Acceleration Of Regular Expression Matching For Large Datasets: Exploring The Implementation Space. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, Hubertus Franke, Alexander Heinecke, Krishna V. Palem, and Eli Upfal (Eds.). ACM, 18:1–18:10.

[62] Sai Zhang and Yuyin Sun. 2013. Automatically Synthesizing SQL Queries From Input-Output Examples. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 224–234.

[63] Zexuan Zhong, Jiaqi Guo, Wei Yang, Jian Peng, Tao Xie, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2018. SemRegex: A Semantics-Based Approach For Generating Regular Expressions From Natural Language Specifications. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. 1608–1618.