Repairing DoS Vulnerability of Real-World Regexes

Nariyoshi Chida
NTT Corporation / Waseda University
nariyoshichidamm@gmail.com

Tachio Terauchi
Waseda University
terauchi@waseda.jp

***Abstract*—There has been much work on synthesizing and repairing regular expressions (*regexes* for short) from examples. These *programming-by-example* (PBE) methods help the users write regexes by letting them reflect their intention by examples. However, the existing methods may generate regexes whose matching may take super-linear time and are vulnerable to regex denial of service (ReDoS) attacks. This paper presents the *first* PBE repair method that is guaranteed to generate only invulnerable regexes. Importantly, our method can handle *real-world regexes* containing *lookarounds* and *backreferences*. Due to the extensions, the existing formal definitions of ReDoS vulnerabilities that only consider pure regexes are insufficient. Therefore, we first give a novel *formal semantics and complexity of backtracking matching algorithms for real-world regexes*, and with them, give the *first formal definition of ReDoS vulnerability for real-world regexes*. Next, we present a novel condition called *real-world strong 1-unambiguity* that is sufficient for guaranteeing the invulnerability of real-world regexes, and formalize the corresponding PBE repair problem. Finally, we present an algorithm that solves the repair problem. The algorithm builds on and extends the previous PBE methods to handle the real-world extensions and with constraints to enforce the real-world strong 1-unambiguity condition.**

***Index Terms*—Real-world regexes, ReDoS, synthesis, repair**

I. Introduction

Regular expressions (regexes for short) have become an integral part of modern programming languages and software development, e.g., they are used as general purpose libraries [1], [2], for sanitizing user inputs [3], [4], and extracting data from unstructured text [5], [6]. Despite the widespread use of regexes in practice, it is an unfortunate fact that developers often write regexes which are vulnerable to *regex denial-of-service* (ReDoS) attacks in which attackers craft inputs that cause the regex matching algorithm to take super linear time [7], [8]. ReDoS is a significant threat to our society due to the widespread use of regexes [7], [9]–[11]. While some regex engines offer mechanisms to limit their run time (directly by timeout or indirectly by limiting the number of backtrackings), determining a proper limit is often difficult, not to mention that vulnerable regexes may not even have any reasonable limits that can be assigned as they may struggle even on legitimate inputs. Furthermore, such options are not available in many popular regex engines including those in the standard libraries of Python, Java, and Node.js.

To address the issue, there has been much research on the topic of overcoming ReDoS vulnerability [12]–[17]. However, the previous works have focused mainly on the problem of detecting vulnerable regexes, and the problem of *repairing* them remains largely open. As reported by Davis et al. [7], [18], writing invulnerable regexes is a formidable task that developers often fail to achieve in practice.

Meanwhile, recent years have seen remarkable progress on *programming-by-example* (PBE) methods for synthesizing and repairing regexes [5], [19]–[24]. In these methods, a set of positive examples (strings to be accepted) and negative examples (strings to be rejected) are provided with the goal to synthesize a regex that correctly classify the examples, often with additional constraints to bias the synthesis toward ones syntactically close to the pre-repair regex [22], [24]. PBE methods have the salient advantage of easing the burden of writing correct regexes by letting the users reflect their intention by examples [20]–[22], [24]. However, the existing PBE methods are not designed with resilience to ReDoS in mind and may generate vulnerable regexes.[1]

In this paper, we rectify the situation by proposing the *first* PBE repair method that is *guaranteed to generate only invulnerable regexes*. Importantly, our method can handle the so-called *real-world* regexes that have extensions such as *lookarounds*, *capturing groups*, and *backreferences* [25].

While previous works have investigated formal definitions of ReDoS vulnerability [12], [14], [16], they only address the pure regex fragment. The overarching challenge in ReDoS vulnerabilities is to define the complexity of backtracking matching algorithm. The previous works for pure regexes have used nondeterministic finite automata (NFA) to formalize the behavior of backtracking matching algorithms and its complexity. Unfortunately, such a NFA-based definition is difficult for real-world regexes because the expressive power of real-world regexes is not regular [26].

Our first contribution is the first formal definition of ReDoS vulnerability for real-world regexes. For this, we introduce a novel formal semantics of backtracking matching algorithm for real-world regexes and, by building on it, formally define the time complexity of backtracking matching algorithms for real-world regexes. Also, we have discovered a subtle bug in a previous formal definition of ReDoS vulnerability for pure regexes [12] which can misclassify some vulnerable regexes as invulnerable (even for pure regexes). Although the bug is fixable, this shows the subtlety of formalizing ReDoS vulnerability.

[1] The only exception is the recent work by Li et al. [24], but they only handle pure regexes and also lack the guarantee to generate only invulnerable regexes (cf. Section VIII).

Our repair method ensures invulnerability by enforcing the novel *real-world strong 1-unambiguity* (RWS1U) introduced in this paper. RWS1U is inspired by a notion for pure regexes called *strong 1-unambiguity* [27], and can be considered as an extension of it to real-world regexes. We show that RWS1U is a sufficient condition for invulnerability, and formalize a PBE repair problem, *RWS1U repair problem*, whose goal includes ensuring RWS1U. We prove that the RWS1U repair problem is NP-hard. We also show that a related notion for pure regexes called *1-unambiguity* (also called *deterministic regex*) [24], [28], [29] is insufficient for guaranteeing invulnerability (even for pure regexes).

Our third contribution is an algorithm for solving the RWS1U repair problem. Our algorithm builds on the previous PBE regex repair methods. However, significant extensions are needed because *the previous methods neither support real-world regexes nor concern ReDoS vulnerability* (with the exception of [24] mentioned above). A key step of the algorithm is generating SMT constraints that enforce both the RWS1U condition and consistency with examples. The latter is enforced by following our novel formal semantics of real-world regexes, and the former is enforced by using our novel *extended NFA translation* that is used to define RWS1U. We also adapt and extend the key techniques proposed for PBE regex synthesis and repair, such as the state space pruning technique by under- and over-approximations [20], [22], with the support for the real-world extensions and concerns for ReDoS vulnerability.

We have implemented a prototype of our algorithm in a tool called REMEDY (Regular Expression Modifier for Ensuring Deterministic propertY), and have experimented with the tool on a set of benchmarks of real-world regexes taken from [7]. The experimental results show that REMEDY was able to successfully repair non-trivial vulnerable regexes from a real-world data set.

The contributions of the paper are summarized below.

- We initiate a study of ReDoS vulnerabilities for real-world regexes. To this end, we give a novel formal semantics and the time complexity of backtracking matching algorithms for real-world regexes, and with it, give the first formal definition of their ReDoS vulnerability. We also show a subtle bug in a previous proposal for pure regexes [12]. (Section III)
- We present the novel *real-world strong 1-unambiguity* (RWS1U), and prove that the condition is sufficient for guaranteeing invulnerability for real-world regexes. We define the RWS1U repair problem and prove that the problem is NP-hard. We also show that a related condition, 1-unambiguity (i.e., deterministic regex) for pure regexes [24], [28], [29] is insufficient for ensuring invulnerability (even for pure regexes). (Section IV)
- We give an algorithm for solving the RWS1U repair problem that builds on and extends the previous PBE synthesis and repair methods. Our algorithm extends the previous methods in two important ways: support for the real-world extensions and the incorporation of RWS1U to enforce invulnerability. (Section V)
- We present an implementation of the algorithm in a tool called REMEDY, and present an evaluation of the tool on a set of real-world benchmarks. (Section VI)

II. OVERVIEW

We give an informal overview of our repair algorithm by an example. To illustrate, we use the regex $<(\cdot^*)_1>\cdot^*</\backslash 1>$ which is inspired by the one posted in [30]. The regex is intended to accept a *non-nested* XML tag, i.e., a tag that appears as a leaf in an XML document. For example, it should accept `<li></li>` and `<body>text</body>`, but it should reject `<li></body>`, `<body><li></li></body>`, and `<body><li></body>`. Unfortunately, the regex is both incorrect and vulnerable. It is incorrect because it accepts tags such as `<body><li></body>`. It is vulnerable because it takes quadratic time to match strings such as $<><>< \cdots ></>$ where $\cdots$ repeats $><$. Indeed, running a regex engine such as Python's *re* on the regex will get stuck on suitably long strings of the above form.

REMEDY can help the user automatically repair a regex like this into a correct invulnerable one. To this end, the user provides the regex to be repaired along with sets of positive and negative examples. Positive examples are strings that should be accepted, and negative examples are those that should be rejected.

Sampling examples. As usual in a PBE scenario [20], [22]–[24], the user prepares test inputs that consists of positive and negative examples to validate the correctness of the regex. Such examples may be prepared afresh by the user [31] or obtained from an existing collection such as RegExLib [32]. Generally, the result of PBE depends on the example selection. Therefore, if the user cannot obtain an intended repair, she adds or removes examples and re-runs the tool to improve the result. We note that, for usability, PBE should only use a relatively small number of examples. For the running example, suppose that the user prepared positive examples `<ab></ab>` and `<a>ab</a>` and negative examples `<a></b>`, `<a><b></b></a>`, and `<a><ab></a>`.

REMEDY explores a regex that is consistent with the examples and has *real-world strong 1-unambiguity* (RWS1U). Also, REMEDY looks for regexes that are syntactically close to the given one to bias toward synthesizing regexes that are close to the user's intention. The assumption is that the given regex may not be correct but is close to the one user intended.

RWS1U ensures the invulnerability of the synthesized regex. Roughly, it makes the behavior of the matching algorithm *backtrack-free* thus ensuring linear running time. The regex $<(\cdot^*)_1>\cdot^*</\backslash 1>$ violates the RWS1U condition because there are two ways to match $>$ in the input string after the first $<$ is matched, that is, it can match the first $\cdot^*$ or the first $>$. Likewise, after $</$ is matched, there are again two ways to match $>$: $\backslash 1$ if it refers to a string that starts with $>$ or the second $>$. There are also multiple ways to match $<$ in the input string. Next, we describe the steps of the repair process.

Generating templates. REMEDY generates *templates*, which are regexes containing *holes*. Informally, a hole $\square$ is a placeholder that is to be replaced with some concrete regex. REMEDY starts with the initial template set to be the input regex `<(.*)1>.*</\1>`. Since the regex is vulnerable and does not satisfy the RWS1U condition, REMEDY replaces the subexpressions with holes and expands the holes by replacing them with templates such as $\square\square$, $\square|\square$, $\square^*$, (?=$\square$), and $\backslash i$. After some iterations, we get the template `<(`$\square_1^*$`)1>`$\square_2^*$`</\1>`.

Searching assignments. Next, REMEDY checks if the template can be instantiated to a regex that satisfies the required conditions by replacing its holes with some sets of characters. For this, REMEDY generates two types of constraints: *consistency-with-examples constraint* that ensures that the regex is consistent with the examples, and *linear-time constraint* that asserts RWS1U. REMEDY looks for a regex that satisfies the constraints by using an SMT solver. If the constraints are unsatisfiable, then REMEDY backtracks to explore more templates. We give the details of the constraint generation in Section V-A. REMEDY also performs *template pruning* to filter out templates that can be efficiently detected impossible to be instantiated to a regex that is consistent with the examples. The details are presented in Section V-A.

Using an SMT solver, REMEDY finds that the constraints are satisfiable, and replaces $\square_1$ and $\square_2$ with [^>] and [^<], respectively. Here, [^a] is a regex that matches any character besides a. Finally, REMEDY returns `<([^>]*)1>[^<]*</\1>` as the repaired regex which is invulnerable and matches the user's intention.

III. REAL-WORLD REGULAR EXPRESSIONS

In this section, we give the definition of real-world regexes. We also present the novel formal model of the backtracking matching algorithm for real-world regexes, and with it, we formally define their ReDoS vulnerability.

Notations. Throughout this paper, we use the following notations. We write Σ for a finite alphabet; $a,b,c,\in\Sigma$ for a character; $w,x,y,z\in\Sigma^*$ for a sequence of characters; ε for the empty sequence; r for a real-world regex; $\mathbb{N}$ for the set of natural numbers. For the string $x = x[0]...x[n-1]$, its length is $|x| = n$. For $0 \le i \le j < |x|$, the string $x[i]...x[j]$ is called a substring of x. We write $x[i..j]$ for the substring. In addition, we write $x[i..j)$ for the substring $x[i]...x[j-1]$. We assume that $x[i..j) = \$$, where $\$ \notin \Sigma$, when $i < 0$ or $|x| < j$. For f a (partial) function, $f[\alpha \mapsto \beta]$ denotes the (partial) function that maps α to β and behaves as f for all other arguments. We write $f(\alpha) = \perp$ if f is undefined at α. We define $ite(true,A,B) = A$ and $ite(false,A,B) = B$.

A. Syntax and Informal Semantics

The syntax of *real-world regexes* (simply *regexes* or *expressions* henceforth) is given below:

$$\begin{array}{rcl} r & ::= & [C] \mid \varepsilon \mid rr \mid r|r \mid r^* \\ & \mid & (r)_i \mid \backslash i \mid (?{=}r) \mid (?!r) \mid (?{<}{=}x) \mid (?{<}!x) \end{array}$$

Here, $C \subseteq \Sigma$ and $i \in \mathbb{N}$. A set of characters $[C]$ exactly matches a character in C. We sometimes write a for $[\{a\}]$, and write $\cdot$ for $[\Sigma]$. The semantics of empty string ε, concatenation r_1r_2, union $r_1|r_2$ and repetition r^* are standard. Many convenient notations used in practice such as options, one-or-more repetitions, and interval quantifiers can be treated as syntactic sugars: $r? = r|\varepsilon$, $r^+ = rr^*$, and $r\{i,j\} = r_1...r_ir_{i+1}?...r_j?$ where $r_k = r$ for each $k \in \{1,\ldots,j\}$.

The remaining constructs, that is, capturing groups, backreferences, (positive and negative) lookaheads and lookbehinds, comprise the real-world extensions. In what follows, we will explain the semantics of the extended features informally in terms of the standard backtracking matching algorithm which attempts to match the given regex with the given (sub)string and backtracks when the attempt fails. The formal definition is given later in the section.

A *capturing group* $(r)_i$ attempts to match r, and if successful, stores the matched substring in the storage identified by the index i. Otherwise, the match fails and the algorithm backtracks. A *backreference* $\backslash i$ refers to the substring matched to the corresponding capturing group $(r)_i$, and attempts to match the same substring if the capture had succeeded. If the capture had not succeeded or the matching against the captured substring fails, then the algorithm backtracks. For example, let us consider the regex `([0-9])1([A-Z])2\1\2`. Here, `\1` and `\2` refer to the substring matched by `[0-9]` and `[A-Z]`, respectively. The language represented by the regex is $\{abab \mid a \in$ `[0-9]` $\wedge\, b \in$ `[A-Z]`$\}$. Capturing groups in practice often do not have explicit indexes, but we write them here for clarity. We assume without loss of generality that each capturing group always has a corresponding backreference and vice versa. We assume that capturing group indexes are always distinct in a regex.

A *positive (resp. negative) lookahead* (?=r) (resp. (?!r)) attempts to match r without any character consumption, and proceeds if the match succeeds (resp. fails) and backtracks otherwise. A *fixed-string positive (resp. negative) lookbehind* (?<=x) (resp. (?<!x)) looks back (i.e., toward the left), attempts to match x without any character consumption, and proceeds if the match succeeds (resp. fails) or otherwise backtracks. Fixed-string lookbehinds are supported by major regex engines such as those in Perl and Python [33]. Note that most regex engines do not support general lookbehinds [25].

B. Formal Semantics and Vulnerability

We now formally define the semantics of regexes. Traditionally, the language of pure regexes is defined by induction on the structure of the expressions. However, such a definition would be difficult for real-world regexes because of the extended features and also unsuitable for formalizing vulnerability because the notion concerns the complexity of backtracking matching algorithms. To this end, we define the semantics by the matching relation $\rightsquigarrow$ that models the behavior of backtracking matching algorithms.

A matching relation is of the form $(r,w,p,\Gamma) \rightsquigarrow \mathscr{N}$ where p is a position on the string w such that $0 \le p \le |w|$, Γ is

$$\frac{(r,w,p,\Gamma) \rightsquigarrow \mathcal{N}}{((r)_j,w,p,\Gamma) \rightsquigarrow \{(p_i,\Gamma_i[j \mapsto w[p..p_i])] \mid (p_i,\Gamma_i) \in \mathcal{N}\}} \text{(CAPTURING GROUP)}$$

$$\frac{\Gamma(i) \neq \bot \qquad (\Gamma(i),w,p,\Gamma) \rightsquigarrow \mathcal{N}}{(\backslash i,w,p,\Gamma) \rightsquigarrow \mathcal{N}} \text{(BACKREFERENCE)}$$

$$\frac{\Gamma(i) = \bot}{(\backslash i,w,p,\Gamma) \rightsquigarrow \emptyset} \text{(BACKREFERENCE FAILURE)}$$

$$\frac{(r,w,p,\Gamma) \rightsquigarrow \mathcal{N}}{((?{=}r),w,p,\Gamma) \rightsquigarrow \{(p,\Gamma') \mid (\_,\Gamma') \in \mathcal{N}\}} \text{(POSITIVE LOOKAHEAD)}$$

$$\frac{(r,w,p,\Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset,\emptyset,\{(p,\Gamma)\})}{((?!r),w,p,\Gamma) \rightsquigarrow \mathcal{N}'} \text{(NEGATIVE LOOKAHEAD)}$$

$$\frac{(x,w[p-|x|..p),0,\Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset,\{(p,\Gamma)\},\emptyset)}{((?{<=}x),w,p,\Gamma) \rightsquigarrow \mathcal{N}'} \text{(POSITIVE LOOKBEHIND)}$$

$$\frac{(x,w[p-|x|..p),0,\Gamma) \rightsquigarrow \mathcal{N} \qquad \mathcal{N}' = ite(\mathcal{N} \neq \emptyset,\emptyset,\{(p,\Gamma)\})}{((?{<}!x),w,p,\Gamma) \rightsquigarrow \mathcal{N}'} \text{(NEGATIVE LOOKBEHIND)}$$

Fig. 1: Selected rules of the matching relation $\rightsquigarrow$

a function that maps each capturing group index to a string captured by the corresponding capturing group, and $\mathcal{N}$ is a set of matching results. A *matching result* is a pair of a position and a capturing group function. Roughly, (r,w,p,Γ) is read: a regex r tries to match the string w from the position p, with the information about capturing groups Γ. For example, for the regex a on the strings a and b, the matching relations are $(a,a,0,\emptyset) \rightsquigarrow \{(1,\emptyset)\}$ and $(a,b,0,\emptyset) \rightsquigarrow \emptyset$, respectively. From these, the matching relation of the regex $(a|b)$ on the string a is $((a|b),a,0,\emptyset) \rightsquigarrow \{(1,\emptyset)\}$.

Figure 1 shows some rules for deducing the matching relation. For space, we show only the rules for handling the extended features, deferring the full rules to the Appendix. The rules are inspired by [34] who have given natural-semantics-style rules for pure regexes and parsing expression grammars. However, to our knowledge, we are the first to give the formal semantics of real-world regexes in this style and use it to formalize vulnerability.

In the rule (CAPTURING GROUP), we first get the matching result $\mathcal{N}$ from matching w against r at the current position p. And for each matching result $(p_i,\Gamma_i) \in \mathcal{N}$ (if any), we record the matched substring $w[p..p_i)$ in the corresponding capturing group map Γ_i at the index i. The rule (BACKREFERENCE) looks up the captured substring and tries to match it with the input at the current position. The match fails if the corresponding capture has failed as stipulated by the rule (BACKREFERENCE FAILURE).

In the rule (POSITIVE LOOKAHEAD), the expression r is matched against the given string w at the current position p to obtain the matching results $\mathcal{N}$. Then, for every match result $(p',\Gamma') \in \mathcal{N}$ (if any), we reset the position from p' to p. This models the behavior of lookaheads which does not consume the string. The rule (NEGATIVE LOOKAHEAD) is similar, except that we reset and proceed when there is no match. Note that captures made inside of a negative lookahead cannot be referred outside of the lookahead, which agrees with the behavior of regex engines in practice. The rules (POSITIVE LOOKBEHIND) and (NEGATIVE LOOKBEHIND) for handling fixed-string lookbehinds are self-explanatory.

Definition III.1 (Language)**.** The *language* of a regex r is defined as $L(r) = \{w \mid (r,w,0,\emptyset) \rightsquigarrow \mathcal{N} \wedge \exists \Gamma.(|w|,\Gamma) \in \mathcal{N}\}$.

We show some examples of matchings. For brevity, we omit capturing group information from Examples III.1 and III.2 because it is not used there, i.e., it is always $\emptyset$.

Example III.1. The matching of the regex $(a^*)^*$ on the string ab is as follows:

$$\frac{\dfrac{0 < |ab| \quad a \in \{a\}}{(a,ab,0) \rightsquigarrow \{1\}} \qquad \dfrac{\dfrac{1 < |ab| \quad b \notin \{a\}}{(a,ab,1) \rightsquigarrow \emptyset}}{(a^*,ab,1) \rightsquigarrow \{1\}}}{(a^*,ab,0) \rightsquigarrow \{0,1\}} \qquad \frac{\dfrac{1 < |ab| \quad b \notin \{a\}}{(a,ab,1) \rightsquigarrow \emptyset}}{((a^*)^*,ab,1) \rightsquigarrow \{1\}}$$

$$((a^*)^*,ab,0) \rightsquigarrow \{0,1\}$$

The regex rejects the string because $|ab| = 2 \notin \{0,1\}$.

Example III.2. The matching of $((?{=}a)^*)^*$ on ab is:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{0 < |ab| \quad a \in \{a\}}{(a,ab,0) \rightsquigarrow \{1\}}}{((?{=}a),ab,0) \rightsquigarrow \{0\}}}{((?{=}a)^*,ab,0) \rightsquigarrow \{0\}}}{(((?{=}a)^*)^*),ab,0) \rightsquigarrow \{0\}}$$

The regex rejects the string because $|ab| = 2 \notin \{0\}$.

Example III.3. The matching of $(a^*)_1\backslash 1$ on aa is:

$$\frac{A \quad B_0 \quad B_1 \quad B_2}{(a^*)_1\backslash 1 \rightsquigarrow \{(0,\Gamma_0),(2,\Gamma_1)\}}$$

where $\Gamma_0 = \{(1,\varepsilon)\}$, $\Gamma_1 = \{(1,a)\}$, $\Gamma_2 = \{(1,aa)\}$ and the subderivation A is:

$$\frac{\dfrac{0 < |aa| \quad a \in \{a\}}{(a,aa,0,\emptyset) \rightsquigarrow \{(1,\emptyset)\}} \qquad \dfrac{C_0 \qquad \dfrac{C_1}{(a^*,aa,2,\emptyset) \rightsquigarrow \emptyset}}{(a^*,aa,1,\emptyset) \rightsquigarrow \{(1,\emptyset),(2,\emptyset)\}}}{(a^*,aa,0,\emptyset) \rightsquigarrow \{(0,\emptyset),(1,\emptyset),(2,\emptyset)\}}$$

$$((a^*)_1,aa,0,\emptyset) \rightsquigarrow \{(0,\Gamma_0),(1,\Gamma_1),(2,\Gamma_2)\}$$

and the roots of the subderivations B_0, B_1, B_2, C_0, C_1 are, respectively, $(\backslash 1,aa,0,\Gamma_0) \rightsquigarrow \{(0,\Gamma_0)\}$, $(\backslash 1,aa,1,\Gamma_1) \rightsquigarrow \{(2,\Gamma_1)\}$, $(\backslash 1,aa,2,\Gamma_2) \rightsquigarrow \emptyset$, $(a,aa,1,\emptyset) \rightsquigarrow \{(2,\emptyset)\}$, $(a,aa,2,\emptyset) \rightsquigarrow \emptyset$. The regex accepts the string as $(|aa|,\Gamma_1) \in \{(0,\Gamma_0),(2,\Gamma_1)\}$.

We define the *size* of the derivation a matching relation to be the number of nodes in the derivation tree. Note that the size is well defined because our rules are deterministic.

Definition III.2 (Running time)**.** For a regex r and a string w, we define the *running time* of the backtracking matching algorithm on r and w, $Time(r,w)$, to be the size of the derivation of $(r,w,0,\emptyset) \rightsquigarrow \mathcal{N}$.

Definition III.3 (Vulnerable Regular Expressions)**.** We say that an expression r is *vulnerable* if $Time(r,w) \notin O(|w|)$.

Note that a regex r is vulnerable iff there exist infinitely many strings w_0, w_1,... such that $Time(r,w_i)$ (for $i \in \mathbb{N}$) grows

super-linearly in $|w_i|$. Such strings are often called *attack strings*. For example, $(a^*)^*$ in Example III.1 and $(a^*)_1\backslash 1$ in Example III.3 are vulnerable because there exist attack strings $\{a^n b \mid n \in \mathbb{N}\}$ on which $(a^*)^*$ and $(a^*)_1\backslash 1$ respectively take $\Omega(n!)$ and $\Omega(n^2)$ time. Indeed, running an actual regex engine such as Python's *re* on these regexes with these attack strings exhibits a super-linear behavior. By contrast, $((?{=}a)^*)^*$ in Example III.2 takes $O(n)$ time on these strings and is in fact invulnerable.

Our matching semantics captures the behavior of common backtracking matching algorithms used in most real regex engines, e.g., ones based on path traversal of some non-deterministic automaton [12], [13], [17]. We remark that our formal semantics may be less efficient than an actual regex engine because it computes all possible runs without any optimization. However, it is sound for defining invulnerability, and our repair algorithm synthesizes regexes that are invulnerable even with respect to the inefficient formal semantics. This implies that if a pure regex is considered vulnerable according to the definition of vulnerability in [12] then it is also considered vulnerable according to our definition.

It is worth noting that $(a^*)^*$ is incorrectly classified as invulnerable by [12], both according to their formal definition of vulnerability and by their vulnerability detection tool. Although the bug is fixable by adding ε transitions to their NFA-based definition in a certain way, this shows the subtlety of formalizing vulnerability.

IV. RWS1U and Its Repair Problem

This section presents our PBE repair algorithm. First, we define the novel notion of *real-world strong 1-unambiguity* (RWS1U) and prove it to be sound for ensuring invulnerability (Section IV-A). Then, we define *RWS1U repair problem* to be the problem of synthesizing a regex that correctly classifies the given positive and negative examples, satisfies RWS1U, and is syntactically close to the pre-repair regex (Section IV-B). We prove that the RWS1U repair problem is NP-hard. Section V presents an algorithm for solving the RWS1U repair problem.

A. Real-World Strong 1-Unambiguity

We begin by introducing some preliminary notions.

Definition IV.1 (Bracketing)**.** The *bracketing* of r, $r^{[]}$, is obtained by inductively mapping each subexpression s of r to $[_i s]_i$ where i is a unique index. Here, $[_i$ and $]_i$ are called *brackets* and are disjoint from the alphabet Σ of r.

Note that $r^{[]}$ is a regex over the alphabet $\Sigma \cup \Psi$, where $\Psi = \{[_i,]_i | i \in \mathbb{N}\}$. We call Ψ the *bracketing alphabet* of $r^{[]}$. For example, for $r = ((a)^*)^* b$, the bracketing is

$$r^{[]} = [_1[_2([_3([_4 a]_4)^*]_3)^*]_2[_5 b]_5]_1$$

with the bracketing alphabet $\{[_i,]_i \mid i \in \{1,2,3,4,5\}\}$.

Definition IV.2 (Lookaround removal)**.** The regex r with its *lookarounds removed*, $rmla(r)$, is r but with each of its lookaround replaced by ε.

A *non-deterministic automaton* (NFA) over an alphabet Σ is a tuple (Q, δ, q_0, q_n) where Q is a finite set of states, $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the transition relation, q_0 is the initial state, and q_n is the accepting state.

Definition IV.3 (*eNFAtr*)**.** For a lookaround-free regex r over Σ, its *extended NFA translation*, $eNFAtr(r^{[]})$, is a NFA over $\Sigma \cup \Psi$ defined by the rules shown in Figure 2 where Ψ is the bracketing alphabet of $r^{[]}$.

In the translation shown in Figure 2, we maintain a global map $\mathscr{I}$ from capturing group indexes to states. $\mathscr{I}$ is initially empty and is updated whenever a capturing group $(r)_i$ is encountered so that $\mathscr{I}(i)$ is set to be the initial state of the NFA constructed from r. $Fst(q)$ is defined as follows: $\rho a \in Fst(q)$ iff $\rho \in \Psi^*$, $a \in \Sigma$, and there is a ρa-labeled path from q. We define $\rho a^\natural = a$, and $Fst(q)^\natural = \{a \mid \rho a \in Fst(q)\}$. Roughly, $Fst(q)^\natural$ is the set of characters that r can reach without any character consumption where q is the initial state of $eNFAtr(r)$. For example, for $r = ab|ac|d^*ef$, $Fst(q)^\natural = \{a, d, e\}$ where q is the initial state of $eNFAtr(r)$.

Our extended NFA translation may be seen as the standard Thompson's translation for pure regexes [35], [36] extended to real-world regexes. However, unlike the Thompson's translation, it does not preserve the semantics (necessarily not so because real-world regexes are not regular even without lookarounds). Instead, we use the translation only for the purpose of defining RWS1U. For a pair of states q and q' of a NFA, we write $paths(q, q')$ for the set of strings that take the NFA from q to q'.

Definition IV.4 (*Bps*)**.** For r a regex over Σ, Ψ the bracketing alphabet of $r^{[]}$, $[_i \in \Psi$, $a \in \Sigma$, and $(\_, \delta, \_, \_) = eNFAtr(r^{[]})$, we define $Bps(r, [_i, a)$ to be the set below:

$$\{\rho \in \Psi^* \mid \exists (q_j, [_i, \_), (q_l, a, \_) \in \delta . \rho \in paths(q_j, q_l)\}.$$

Roughly, $Bps(r, [_i, a)$ are the sequences of brackets appearing in paths from the unique edge labeled $[_i$ to an edge labeled a in the extended NFA translation of r.

Example IV.1. Figure 3 shows the extended NFA translation of $(a^*)^*$ where unlabeled edges denote ε transitions. Note that $Bps((a^*)^*, [_1, a) = \{[_1([_2]_2)^n[_2[_3 \mid n \in \mathbb{N}\}$.

Definition IV.5 (RWS1U)**.** We say that a regex r satisfies *real-world strong 1-unambiguity* (RWS1U) if (1) $|Bps(rmla(r), [_i, a)| \leq 1$ for all $a \in \Sigma$ and $[_i \in \Psi$ where Ψ is the bracketing alphabet of $rmla(r)^{[]}$ and (2) lookarounds in r do not contain repetitions and backreferences.

Roughly, condition (1) ensures that the matching algorithm can determine which subexpression to match next by looking at the next character in the input string. It therefore rules out the need for backtracking. The condition is inspired by a notion called *strong 1-unambiguity* for pure regexes [27] and can be seen as an extension of it to regexes containing backreferences. We do not impose the condition in lookarounds, because the condition prohibits some important use patterns of them. For instance, it will preclude any meaningful use

$$
\begin{array}{rcl}
eNFAtr([C]) & = & (\{q_0,q_1\},\{(q_0,a,q_1) \mid \forall a \in C\},q_0,q_1) \\
eNFAtr(r_1r_2) & = & (Q_1 \cup Q_2, \delta_1 \cup \delta_2 \cup \{(q_{n_1},\varepsilon,q_{0_2})\},q_{0_1},q_{n_2}) \;\; \textit{where} \;\; (Q_1,\delta_1,q_{0_1},q_{n_1}) = eNFAtr(r_1) \textit{ and } (Q_2,\delta_2,q_{0_2},q_{n_2}) = eNFAtr(r_2) \\
eNFAtr(r_1|r_2) & = & (Q_1 \cup Q_2 \cup \{q_0,q_n\}, \delta_1 \cup \delta_2 \cup \{(q_0,\varepsilon,q_{0_1}),(q_0,\varepsilon,q_{0_2}),(q_{n_1},\varepsilon,q_n),(q_{n_2},\varepsilon,q_n)\},q_0,q_n) \\
 & & \textit{where} \;\; (Q_1,\delta_1,q_{0_1},q_{n_1}) = eNFAtr(r_1) \textit{ and } (Q_2,\delta_2,q_{0_2},q_{n_2}) = eNFAtr(r_2) \\
eNFAtr(r^*) & = & (Q \cup \{q_0,q_n\}, \delta \cup \{(q_0,\varepsilon,q_{0_1}),(q_0,\varepsilon,q_n),(q_{n_1},\varepsilon,q_n),(q_{n_1},\varepsilon,q_{0_1})\},q_0,q_n) \;\; \textit{where} \;\; (Q,\delta,q_{0_1},q_{n_1}) = eNFAtr(r) \\
eNFAtr((r)_i) & = & eNFAtr(r) \textit{ and } \mathscr{I} = \mathscr{I}[i \mapsto q_0] \;\; \textit{where} \;\; eNFAtr(r) = (\_,\_,q_0,\_) \\
eNFAtr(\backslash i) & = & (\{q_0,q_1\},\{(q_0,a,q_1) \mid a \in Fst(\mathscr{I}(i))^\natural\} \cup \{(q_0,\varepsilon,q_1) \mid (r)_i \text{ and } \varepsilon \in L(r)\},q_0,q_1)
\end{array}
$$

Fig. 2: The extended NFA translation.

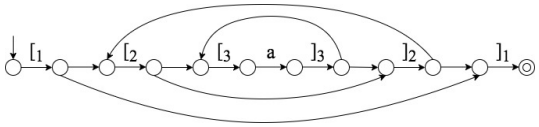


Fig. 3: The extended NFA translation of $(a^*)^*$.

of a positive lookahead because if the lookahead succeeds then the subexpression immediately following the lookahead must match the same string. Therefore, for lookarounds, we impose the condition stipulated by (2). The condition prohibits repetitions and backreferences to appear in a lookaround and ensures that the matching within a lookaround finishes in constant time. Therefore, (1) and (2) combined guarantee that the overall matching finishes in linear time.

Example IV.2. Recall $r_1 = (a^*)^*$, $r_2 = ((?{=}a)^*)^*$, $r_3 = (a^*)_1\backslash 1$ from Examples III.1, III.2, III.3. The regex r_1 does not satisfy the RWS1U condition because as shown in Example IV.1, $|Bps(r_1,[_1,a)| = \aleph_0 > 1$. Also, r_3 does not satisfy the RWS1U condition because $Bps(r_3,[_1,a) = \{[_1[_2[_3[_4,[_1[_2[_3]_3]_2[_5\}$ where

$$r_3^{[]} = [_1[_2([_3([_4a]_4)^*]_3)_1]_2\,[_5\backslash 1]_5]_1,$$

and so $|Bps(r_3,[_1,a)| = 2 > 1$. By contrast, r_2 (trivially) satisfies the RWS1U condition because $rmla(r_2) = (\varepsilon^*)^*$ which contains no characters.

Example IV.3. The regex $r_4 = a^*b^*$ satisfies the RWS1U condition because $|Bps(r_4,[_i,a)| = |Bps(r_4,[_i,b)| = 1$ for $i \in \{1,2,3\}$, and $|Bps(r_4,[_i,a)| = 0$ and $|Bps(r_4,[_i,b)| = 1$ for $i \in \{4,5\}$, where $r_4^{[]} = [_1[_2([_3a]_3)^*]_2\,[_4([_5b]_4)^*]_5]_1$. The regex $r_5 = ((?{=}\cdot^*)\cdot)^*$ does not satisfy the RWS1U condition because the positive lookahead contains a repetition, violating condition (2).

We show that RWS1U is a sufficient condition for invulnerability.

Theorem IV.1. *A regex that satisfies RWS1U is invulnerable.*

The proof appears in the Appendix. We remark that while RWS1U is a sufficient condition, it is not a necessary condition for invulnerability. For example, $a|aa$ is invulnerable but does not satisfy RWS1U.

Finally, we note that a related notion called *1-unambiguity* for pure regexes (also called *deterministic regexes*) [24], [28], [29] is insufficient for guaranteeing invulnerability (even for pure regexes). For example, $(a^*)^*$ is 1-unambiguous, because any character occurs at most once, but it is vulnerable as shown in Section III-B[2].

B. Repair Problem

In this section, we define the RWS1U repair problem. First, we adapt the notion of *distance* between regexes from a recent work on PBE regex repair [22]. In what follows, a regex is identified with its *abstract syntax tree* (AST) representation. For an AST r, we define its *size*, $|r|$, to be the number of nodes of r.

Definition IV.6 (Distance)**.** For non-overlapping subtrees r_1, $\ldots$, r_n of a regex r, an *edit* $r[r_1'/r_1,\cdots,r_n'/r_n]$ replaces each r_i with r_i'. The *cost* of the edit is $\sum_{i\in\{1,\ldots,n\}} |r_i| + |r_i'|$. The *distance* between r_1 and r_2, $D(r_1,r_2)$, is the minimum cost of an edit that transforms r_1 to r_2.

For example, $D(a|b|c,d|c) = 4$, which is realized by the edit that replaces $a|b$ by d. We now define the repair problem.

Definition IV.7 (RWS1U Repair Problem)**.** Given a regex r_1, a finite set of *positive examples* $P \subseteq \Sigma^*$, and a finite set of *negative examples* $N \subseteq \Sigma^*$ where $P \cap N = \emptyset$, the *real-world strong 1-unambiguity repair problem* (*RWS1U repair problem*) is the problem of synthesizing r_2 such that (1) r_2 satisfies RWS1U, (2) $P \subseteq L(r_2)$, (3) $N \cap L(r_2) = \emptyset$, and (4) $D(r_1,r_2) \leq D(r_1,r_3)$ for any regex r_3 satisfying (1)-(3).

Condition (1) guarantees that the repaired regex r_2 is invulnerable. Conditions (2) and (3) assert that r_2 correctly classifies the examples. Condition (4) says that r_2 is syntactically close to the original regex r_1.

We note that the repair problem is easy without the closeness condition (4): one can construct an invulnerable regex that accepts just P (or $\Sigma^* \setminus N$) in time linear in $\sum_{w\in P}|w|$ (or $\sum_{w\in N}|w|$). However, such a regex is unlikely to be one intended by the user, that is, it suffers from *overfitting*. Condition (4) is an important ingredient of a PBE synthesis and repair that biases the solution toward the intended one. The assumption is that the given regex may not be quite correct but is close to the one user intended.

[2]Further details are in Appendix E.

Algorithm 1: The repair algorithm

Input: regex r, positive examples P, negative examples N
Output: a RWS1U regex that is consistent with P and N

```
1: Q ← { r }
2: while Q is not empty do
3:     t ← Q.pop()
4:     if P ⊆ L(t⊤) and N ∩ L(t⊥) = ∅ then
5:         Φ ← getInvulnerableConstraint(t, P, N)
6:         if Φ is satisfiable then
7:             return solution(t, Φ)
8:         Q.push(expandHoles(t))
9:     Q.push(addHoles(t))
```

We show that the RWS1U repair problem is NP-hard by a reduction from EXACTCOVER which is NP-complete [37]. More formally, we consider the decision problem version of the RWS1U repair problem in which we are asked if there is a repair r_2 of r_1 satisfying conditions (1)-(3) and $D(r_1, r_2) \leq k$ for some given $k \in \mathbb{N}$. Note that the decision problem is no harder than the original repair problem because the solution to the repair problem can be used to solve the decision problem.

Theorem IV.2. *The RWS1U repair problem is NP-hard.*

The proof appears in the Appendix.

V. REPAIR ALGORITHM

In this section, we describe the details of our PBE repair algorithm. As discussed in Section II, our algorithm builds on the previous approaches that use template-based search with search pruning [20], [22] and the SMT-based constraint solving to find a solution within the given candidate template [22]. Our algorithm extends the constraint generations and the pruning techniques of the previous approaches with the support for real-world extensions and the assertion of RWS1U to ensure invulnerability. We give the overview of the repair algorithm in Section V-A. The details of the constraint generation is given in Section V-B.

A. Algorithm Overview

Algorithm 1 shows the high-level structure of the repair algorithm. The algorithm takes a regex r, a set of positive examples P, and a set of negative examples N as input. Its output is a regex that satisfies the RWS1U condition and is consistent with P and N. At a high level, our algorithm consists of the following four key components.

Generate the initial template. The priority queue Q maintains regex templates. A *regex template* t is a regex that may contain a *hole* $\square$ denoting a placeholder that is to be replaced by a concrete regex. Its syntax is formally the extension of that of regexes (cf. Section III) and is defined by: $r ::= \cdots \mid \square$. To distinguish, we will use t to range over regex templates and reserve r for concrete regexes.

The queue Q is initialized by pushing the input regex (line 1). Q ranks its elements by the distance defined in Section IV so that templates closer to the input regex are placed before. Due to this, REMEDY outputs a regex that satisfies condition (4) of RWS1U repair problem, i.e., the regex is minimal.

Pruning by approximations. The algorithm next retrieves and removes a template t from the head of Q (line 3), and applies the *feasibility check* to the template (line 4). The feasibility check is introduced by [20] for pure regexes. It is known to substantially reduce the search space and is also used in subsequent works on PBE regex synthesis and repair [22], [23]. We extend the idea with the support for the real-world features.

The over- and under-approximation $t_\top$ and $t_\bot$ are built to satisfy the properties $L(r') \subseteq L(t_\top)$ and $L(t_\bot) \subseteq L(r')$ for any regex r' obtainable by filling the holes of t. If $P \nsubseteq L(t_\top)$ or $N \cap L(t_\bot) \neq \emptyset$, then there is no way to get a regex consistent with P and N from the template, and thus we safely discard the template from the search.

The approximations are built by filling each hole in t with either $\cdot^*$ or $[\emptyset]$ based on whether an under- or over-approximation is to be made and whether the hole appears in even or odd number of negative lookarounds. Let $\overline{\cdot^*} = [\emptyset]$ and $\overline{[\emptyset]} = \cdot^*$. Then, $t_\top = \alpha(t, \cdot^*)$ and $t_\bot = \alpha(t, [\emptyset])$ where $\alpha(t, r)$ is inductively defined as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
\alpha(\square, r) & = & r & \alpha([C], r) & = & [C] \\
\alpha(t_1 t_2, r) & = & \alpha(t_1, r)\alpha(t_2, r) & \alpha(\varepsilon, r) & = & \varepsilon \\
\alpha(t_1|t_2, r) & = & \alpha(t_1, r)|\alpha(t_2, r) & \alpha(t^*, r) & = & \alpha(t, r)^* \\
\alpha((t)_i, r) & = & (\alpha(t, r))_i & \alpha(\backslash i, r) & = & \backslash i \\
\alpha((?{=}t), r) & = & (?{=}\alpha(t, r)) & \alpha((?!t), r) & = & (?!\alpha(t, \overline{r})) \\
\alpha((?{<=}x), r) & = & (?{<=}x) & \alpha((?{<}!x), r) & = & (?{<}!x)
\end{array}
$$

Searching assignments by constraints solving. If the feasibility check passes, the algorithm decides if the template can be instantiated into a regex that is consistent with the examples and satisfies the RWS1U condition by filling each hole with a set of characters (i.e., some $[C]$). This is done by encoding the search problem as a constraint satisfaction problem which is then solved by an SMT solver (lines 5-6). We defer the details of this phase to Section V-B.

Expanding and adding holes to a template. The failure of the SMT solver to find a solution implies that there exists no instantiation of the template obtainable by filling the holes by sets of characters that is consistent with the examples and satisfies the RWS1U condition. In such a case, our algorithm expands the holes in the template to generate unexplored templates and add them to the queue (line 8). For example, the template $(\square)_1\backslash 1$ is expanded to $(\square\square)_1\backslash 1$, $(\square|\square)_1\backslash 1$, $(\square^*)_1\backslash 1$, and so on. Here, to ensure the RWS1U condition, we do not replace the holes in lookarounds with templates containing repetitions.

Finally, if the current template fails the feasibility check and no more templates are in the queue, we generate new templates by adding holes to the current template and add the new templates to the queue, because it would be fruitless to expand the current template any further (line 9). The addition of a new hole is done by replacing a set of characters by a

$$\frac{(t,w,p,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})}{((t)_i,w,p,\Gamma,\phi) \dashrightarrow (\bigcup_{(p_i,\Gamma_i,\phi_{ci})\in\mathscr{S}}(p_i,\Gamma_i[i\mapsto w[p..p_i)],\phi_{ci}),\mathscr{F})}\ \text{(CAPTURING GROUP)}$$

$$\frac{\text{Let } x=\Gamma(i) \qquad x=w[p..p+|x|)}{(\backslash i,w,p,\Gamma,\phi) \dashrightarrow (\{(p+|x|,\Gamma,\phi)\},\emptyset)}\ \text{(BACKREFERENCE)}$$

$$\frac{(t,w,p,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})}{((?{=}t),w,p,\Gamma,\phi) \dashrightarrow (\{(p,\Gamma',\phi') \mid (\_,\Gamma',\phi')\in\mathscr{S}\},\mathscr{F})}\ \text{(POSITIVE LOOKAHEAD)}$$

$$\frac{(t,w,p,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})}{((?!t),w,p,\Gamma,\phi) \dashrightarrow (\{(p,\Gamma,\phi') \mid (\bot,\bot,\phi')\in\mathscr{F}\},\{(\bot,\bot,\phi') \mid (\_,\_,\phi')\in\mathscr{S}\})}\ \text{(NEGATIVE LOOKAHEAD)}$$

$$\frac{(x,w[p-|x|,p),0,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})}{((?{<=}x),w,p,\Gamma,\phi) \dashrightarrow (\{(p,\Gamma,\phi') \mid (p',\Gamma',\phi')\in\mathscr{S}\},\mathscr{F})}\ \text{(POSITIVE LOOKBEHIND)}$$

$$\frac{(x,w[p-|x|,p),0,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})}{((?{<}!x),w,p,\Gamma,\phi) \dashrightarrow (\{(p,\Gamma,\phi') \mid (\bot,\bot,\phi')\in\mathscr{F}\},\{(\bot,\bot,\phi') \mid (\_,\_,\phi')\in\mathscr{S}\})}\ \text{(NEGATIVE LOOKBEHIND)}$$

$$\frac{\Box \text{ is the } i\text{-th hole}}{(\Box,w,p,\Gamma,\phi) \dashrightarrow (\{(p+1,\Gamma,\phi\wedge v_i^{w[p]})\},\{(\bot,\bot,\phi\wedge\neg v_i^{w[p]})\})}\ \text{(HOLE)}$$

Fig. 4: Selected rules for generating consistency-with-examples constraints.

hole or replacing an expression with a hole when an immediate subexpression of the expression is a hole. Note that changing an operator is possible because addHoles can replace an operator with a hole when an immediate subexpression is a hole, and then expandHoles can replace the hole with a different operator. For example, by this, $(a|b)c$ may be repaired to d^*c.

B. Generating Constraints

We show the construction of the SMT constraint. The constraint is a conjunction of the following two constraints: the *consistency-with-examples constraint* which asserts that regex obtained by replacing the holes in the template with the sets of characters is consistent with the given positive and negative examples, and the *linear-time constraint* which further constrains such a regex to satisfy the RWS1U condition. We describe the constructions of each constraint in Section V-B1 and V-B2, respectively.

1) Consistency with Examples: To construct the constraint for ensuring the consistency with examples, we adapt and extend the approach proposed by [22] for constructing a similar constraint for pure regexes to real-world regexes. The main idea of [22] is to have a propositional variable v_i^a for each $a\in\Sigma$ and i that ranges over the number of holes in the given template t so that v_i^a is true iff the set of characters $[C]$ to fill the i-th hole satisfies $a\in C$. Then, the constraint is formulated to find an instantiation of t that satisfies (1) for each positive example, there is a run of the matching algorithm that accepts it, and (2) no run accepts a negative example.

To this end, we define the function `encode` which takes a template t and a string w. It outputs the constraint ϕ_w that is satisfiable iff there exists an instantiation r of t obtained by filling its holes with sets of characters such that $w\in L(r)$. The function `encode` is defined by rules deriving judgements of the form $(t,w,p,\Gamma,\phi) \dashrightarrow (\mathscr{S},\mathscr{F})$. Here, ϕ accumulates the constraints asserted thus far, and $\mathscr{S}$ and $\mathscr{F}$ are sets of *constrained matching results* for successes and failures, respectively. A constrained matching result is a tuple (p,Γ,ϕ), where p is a position, Γ is a function that stores information about capturing groups, and ϕ is a constraint asserting the condition that must be satisfied for the corresponding matching to succeed or fail. Matching results of the form $(\bot,\bot,\_)$ indicate matching failures. Then, we define $\texttt{encode}(t,w) = \bigvee_{(|w|,\_,\phi)\in\mathscr{S}}\phi$ where $(t,w,0,\emptyset,\mathit{true}) \dashrightarrow (\mathscr{S},\_)$.

Figure 4 shows the selected rules of $\dashrightarrow$. Here, the notation $\mathscr{M}[(p,\Gamma,\phi)\mapsto(p',\Gamma',\phi')]$, where $\mathscr{M}$ is either $\mathscr{S}$ or $\mathscr{F}$, denotes $\mathscr{M}$ but with (p,Γ,ϕ) replaced by (p',Γ',ϕ'). For space, we only show the rules for handling the extended features and defer the full rules to the Appendix.

Thanks to our rigorous formalization of the matching relation (cf. Section III-B), the constraint generation rules follow the corresponding rules of the matching relation and are almost straightforward. The main difference is the rule (HOLE) for processing holes. The rule adds constraints to assert that the character $w[p]$ has to be included or not included in the set of characters that replaces the hole by conjoining $v_i^{w[p]}$ to the accumulated constraint ϕ for the success case, and conjoining $\neg v_i^{w[p]}$ to ϕ for the failure case.

Finally, the consistency-with-examples constraint for t is: $\phi_c \triangleq \bigwedge_{w\in P}\texttt{encode}(t,w)\wedge\bigwedge_{w\in N}\neg\ \texttt{encode}(t,w)$.

Example V.1. Consider the template $t=(?!\Box)\Box bc$, the positive examples $P=\{abc,cbc\}$, and the negative examples $N=\{bbc\}$. For the positive examples, we have

$$\texttt{encode}(t,abc)=\neg v_0^a\wedge v_1^a \quad\text{and}\quad \texttt{encode}(t,cbc)=\neg v_0^c\wedge v_1^c.$$

For the negative example, we have $\texttt{encode}(t,bbc)=\neg v_0^b\wedge v_1^b$. Therefore, $\phi_c=((\neg v_0^a\wedge v_1^a)\wedge(\neg v_0^c\wedge v_1^c))\wedge\neg(\neg v_0^b\wedge v_1^b)$.

Algorithm 2: Generation of linear-time constraint

Input: a template t
Output: a constraint ϕ_l

1: $t\leftarrow \mathit{rmla}(t)$
2: $\mathscr{A}\leftarrow \mathit{eNFAtr}(t^{\natural})$ // $\mathscr{A}=(Q,\delta,q_0,q_n)$
3: $\phi_l\leftarrow \mathit{true}$
4: **for each** $(q,[_i,q')\in\delta$ **do**
5: $\quad L\leftarrow \mathit{Fst}(q)$
6: $\quad$ **if** $\rho_i a,\ \rho_j a\in L$, where $\rho_i\neq\rho_j$ **then**
7: $\quad\quad$ **return** *false*
8: $\quad$ **for each** $a\in L^{\natural}$ and $\Box_i\in L^{\natural}$ **do**
9: $\quad\quad\phi_l\leftarrow\phi_l\wedge\neg v_i^a$
10: $\quad$ **for each** $a\in\Sigma$ and $\Box_i,\Box_j\in L^{\natural}$ where $i\neq j$ **do**
11: $\quad\quad\phi_l\leftarrow\phi_l\wedge(\neg v_i^a\vee\neg v_j^a)$
12: **return** ϕ_l

2) Linear Time: Algorithm 2 shows the construction of the linear-time constraint for enforcing RWS1U. It takes as input a template t and returns the linear-time constraint ϕ_l.

The algorithm first removes lookarounds from the template by using *rmla* defined in Definition IV.2 (line 1). Here, we

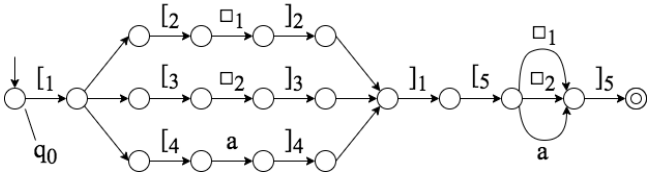


Fig. 5: Simplified version of the extended NFA translation of $[_1([_2\square_1]_2|[_3\square_2]_3|[_4\mathtt{a}]_4)_1]_1[_5\backslash 1]_5$.

extend *rmla* to templates by treating each hole $\square_i$ as a set of characters. Condition (2) of RWS1U which asserts repetition-freedom in lookarounds (cf. Definition IV.5) is ensured by not placing repetitions in lookarounds of a template (cf. **Expanding and adding holes to a template** in Section V-A).

Next, the algorithm constructs a NFA for $t^{[]}$ via the extended NFA translation defined in Definition IV.3 (line 2). Then, for each open bracket $[_i$ in the NFA, the algorithm computes the set of paths $Fst(q)$ where q is the source state of the (unique) $[_i$-labeled edge. Here, we extend *Fst* so that a hole $\square_i$ is treated as the set of characters $[\square_i]$ (cf. Section IV-A).

We then check if there are multiple brackets-only routes from $[_i$ that reach a same character (line 6). If the check passes, then $\mathit{—Bps}(\mathit{rmla}(r),[_i,a)| \geq 2$ for any regex r obtainable from t violating condition (1) of RWS1U, and we safely reject t by returning the unsatisfiable formula *false*.

Otherwise, we proceed to add two types of constraints in lines 8-11. The constraints of the first type added in lines 8-9 assert that, if some character $a \in \Sigma$ and a hole $\square_i$ are both reachable from $[_i$ by bracketing-only paths, then the hole must not be filled with a set of characters that contains a. Here, $L^{\natural} = \{\alpha \mid \rho\alpha \in L\}$ (cf. Section IV-A). The constraints of the second type added in lines 10-11 assert that, if there are two different holes $\square_i$ and $\square_j$ reachable from $[_i$ by bracketing-only paths, then for any character $a \in \Sigma$, at most one of the hole can be filled with a set of characters that contains a. It is easy to see that condition (1) of RWS1U is satisfied iff these constraints are satisfied for all $[_i$. Finally, the algorithm returns the resulting constraint ϕ_l (line 12).

Example V.2. Let us consider running the algorithm on the template $t = (\square_1|\square_2|\mathtt{a})_1\backslash 1(?!\mathtt{a})$. The algorithm first removes lookarounds in the template (line 1), and thus the template becomes $(\square_1|\square_2|\mathtt{a})_1\backslash 1$. Next, the algorithm applies the extended NFA translation to $t^{[]}$ (line 2). Here, $t^{[]}$ is $[_1([_2\square_1]_2|[_3\square_2]_3|[_4\mathtt{a}]_4)_1]_1[_5\backslash 1]_5$. For brevity, we omit the some redundant brackets for sequences and unions. Figure 5 shows the obtained NFA. Then, the algorithm constructs the constraints (lines 4-11). Let us consider the case of q_0. In this case, $Fst(q_0) = \{[_1[_2\square_1, [_1[_3\square_2, [_1[_4\mathtt{a}\}$. Line 9 adds the constraint $\neg v_1^{\mathtt{a}} \wedge \neg v_2^{\mathtt{a}}$ and line 11 adds the constraint $\bigwedge_{a\in\Sigma}(\neg v_1^a \vee \neg v_2^a)$ to ϕ_l.

C. Optimization

We show an optimization to the algorithm. When adding new holes to a template at line 9 of Algorithm 1, we select the sets of characters to be replaced by holes as follows. We analyze the result of the extended NFA translation (which is done anyway for the linear-time constraint) to identify the sets of characters that violate the RWS1U condition, and replace only those with holes. This has the effect of reducing the search space by focusing the synthesis to the parts that contribute to vulnerability.

For example, from the template $\texttt{<s}\square\texttt{an}\cdot^*\texttt{>}$, without the optimization, we may generate up to 2^6 templates by replacing the sets of characters by holes. But with the optimization, we only generate one template $\texttt{<s}\square\texttt{an}\square^*\square$ because $\cdot$ and $>$ are the only sets of characters that violate the RWS1U condition.

VI. Implementation and Evaluation

In this section, we present the results of our evaluation. We evaluate the performance of ReDoS by answering the following questions.

- RQ1 Can ReDoS repair vulnerable regexes efficiently?
- RQ2 Can ReDoS find high-quality regexes?
- RQ3 What is the effect of the optimization?

For the first question, we measure the time taken to repair vulnerable regexes on a real-world data set. For the second question, we measure the quality of repaired regexes using the metrics also used in [22]. For the last question, we compare the running times of ReDoS and ReDoS with the optimization described in Section V-C. Henceforth, we refer to ReDoS with the optimization as ReDoS-o, and use ReDoS-h to denote the hybrid of ReDoS and ReDoS-o that returns the regex returned by the faster of the two.

Finally, we present a comparison of our tool ReDoS with the other state-of-the-art tools in Section VI-E. We compared ReDoS with three state-of-the-art tools AlphaRegex [20], RFixer [22], and FlashRegex [24]. AlphaRegex only supports synthesizing a regex, while RFixer and FlashRegex support both synthesizing and repairing a regex.

A. Experimental Setup

We have implemented ReDoS in Java. We use Z3 [38] as the SMT solver. All experiments were performed on a machine with Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz.

Benchmark. We used *Ecosystem ReDoS data set* collected by Davis et al. [7], which contains real-world regexes in Node.js (JavaScript) and Python core libraries. The data set contains 13,670 regexes that contain real-world extensions (i.e., lookarounds or backreferences). Initially, the regexes are not classified whether they are vulnerable or not. Thus, we contacted the authors of [7] to obtain the subset that they classified as vulnerable. As a consequence, the data set contains 13,591 regexes that contain real-world extensions and are unknown whether they are vulnerable or not, and 79 regexes that contain real-world extensions and are vulnerable. Due to the size, for the former, we selected 100 of them randomly. For the latter, we selected all of them. The average and maximum sizes of the regexes (measured as number of AST nodes) are 32.1 and 383, respectively.

We note that there are no known sound-and-complete ReDoS vulnerability detection methods for real-world regexes (in fact, even whether such a detection is possible is an open question). The 79 regexes that are classified as vulnerable are manually classified as so by Davis et al. We have chosen this data set because its regexes represent real use cases and are also considered to be vulnerable.

Sampling Examples. Since the data set of [7] do not come with examples, we prepared the examples by ourselves. Many of them were made manually, but some were generated automatically, due to the large sizes of the regexes, by the following input generation technique that is inspired by that of [39] for pure regexes.

We first convert the given regex to a backreference-free regex by replacing each capturing group $(r)_i$ and backreference $\backslash i$ by fresh symbols α_i and β_i, respectively. The resulting pure regex (lookarounds can be eliminated for backreference-free regexes [40]) is converted to a DFA. We enumerate the accepting paths of the DFA so that each edge appears in at least one path, with the requirement that an edge β_i can only be taken if the corresponding edge α_i was taken before in the path. Each path is turned into a set of positive examples by replacing each α_i and β_i by a positive example of the regex r where $(r)_i$ is the capturing group (positive examples of r are generated by recursively applying this process). Negative examples are generated similarly by considering the rejecting paths of the DFA.

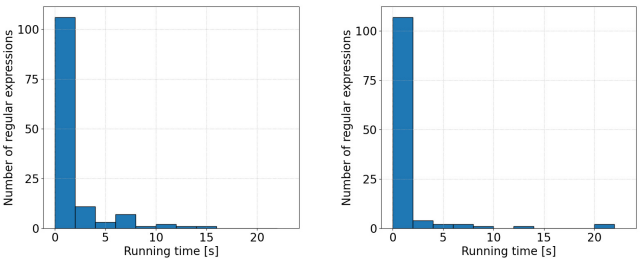
Finally, we used at most 5 positive and negative examples each. We note that, for usability, PBE should only use relatively small numbers of examples.

Consistency with Examples. By construction, REMEDY is guaranteed to only generate regexes that are consistent with the given examples. We have also validated that all regexes that REMEDY generated in the experiment were indeed consistent with the given examples by running the Java's regex library `util.regex`.

ReDoS Invulnerability. By construction, REMEDY is guaranteed to only generate regexes that satisfy RWS1U and hence ReDoS invulnerable. We have also validated that all regexes that REMEDY generated in the experiment indeed satisfied RWS1U. Note that whether a regex satisfies RWS1U can be easily checked by analyzing the extended NFA translation of the regex (cf. Section IV-A).

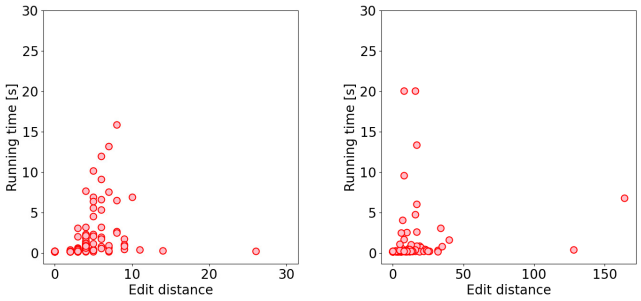
B. RQ1: Efficiency

1) Performance: To evaluate the performance, we ran REMEDY and the variants with a timeout of 30 seconds. We chose 30 seconds because the improvement by setting the timeout to more than 30 seconds was little. The table below summarizes the results. The columns **Solved** and **Average** show that the number of test cases which were repaired within the timeout range and the average running time, respectively. Additionally, Figure 6a and 6b summarize the running times.



(a) Running times of REMEDY (b) Running times of REMEDY-o

Fig. 6: Results of the repairs.



(a) REMEDY (b) REMEDY-o

Fig. 7: Scalability with respect to edit distances.

	Solved(179)	**Average**(s)
REMEDY	132	1.54
REMEDY-o	119	1.08
REMEDY-h	147	0.97

In total, REMEDY, REMEDY-o, and REMEDY-h repaired 73.7%, 66.5%, and 82.1% of regexes, respectively. More than 82.3% of regexes were repaired within 1 second. On the other hand, we observed that the tools could not repair 17.9% of regexes within the time limit. Our inspection showed that the tools struggled on repairs that require large changes from the original. Such repairs may need to explore a large space of possible regexes. An example of such failure cases is a regex that contains a concatenation of many vulnerable sub-regexes, e.g., `(.*[,])*[,]+[ ]+([’"]?)[ a]*\2`$\cdots$, where $\cdots$ is a further concatenations of vulnerable sub-regexes. The finding agrees with that of [22] who have reported that their method also struggled on repairs with large changes, and whose techniques are adapted to our method (cf. Section V). In summary, REMEDY *can repair vulnerable regexes that contain real-world extensions efficiently*.

2) Scalability: Based on the finding, we plot the running times of REMEDY over the edit distances from the input regexes to their repair results. As Figure 7a shows, in the case of REMEDY, we observed a general correlation between the running times and the edit distances: large edit distances require long running times. This observation affirms our initial findings that the size of edit distance affects the running time.

We also observed that, in some cases, REMEDY finished

repairing with a large edit distance and a short running time. To explain the behavior, we use as example the regex below that is derived from the actual case.

$$(?=(\cdot^*)_1[\ \ ]([0\text{–}9][:]\backslash 1[:][:])^*)\cdot^*$$

The regex contains a lookahead with repetitions and backreferences thus violating the condition (2) of RWS1U. REMEDY immediately detects the violation and replaces the repetitions and backreferences with holes. As result, REMEDY reaches the template $(?=(\square)_1[\ \ ]\square)\cdot^*$ in one step. Note that the template replaced quite large sub-expressions with holes and is of a large edit distance. This substantially reduced the search space, and thus the short running time was achieved even with the large edit distance.

On the other hand, as shown in Figure 7b, we observed less correlation between the running times and the edit distances for REMEDY-o. Note that the scale of the edit distance axis is significantly wider than that of Figure 7a. The observation also coincides with our initial findings because REMEDY-o uses the optimization described in Section V-C that can increase the edit distance in a small number of steps, namely all sets of characters violating condition (1) of RWS1U are immediately replaced by holes.

Additionally, to understand how our tool scales as the size of a regex increases, we plot the running time of REMEDY-h over the size of the regex (measured as number of AST nodes). Figure 8 shows the result. The points on the border (colored in blue) indicate that REMEDY-h could not repair the regex within the time limit. Note that the figure is truncated to omit redundant space where no points appear.

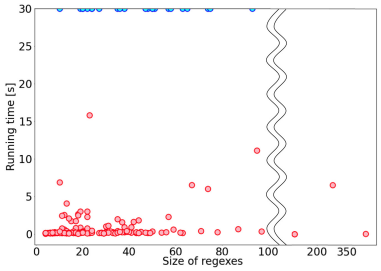


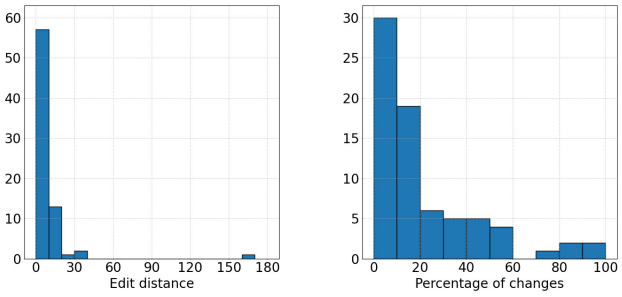
Fig. 8: Scalability wrt. regex sizes.

One can observe that, except for the regexes which led to timeout, REMEDY could repair almost all regexes within 1 second regardless of their size. Additionally, we inspected some of the regexes which require more than 1 seconds to repair and confirmed that they require large changes from the original. That is, the impact of the size of regexes on the implementation is little, while the size of edit distance affects the implementation.

In summary, *the performance of* REMEDY *scales with the size of regexes*. Additionally, *there is a correlation between the running times and the edit distances of* REMEDY.

C. RQ2: Quality

As mentioned by [22], repairs that are similar to the original ones are often considered good in PBE because they are similar to what the user intended. Therefore, to evaluate the quality of repaired regexes objectively, we measure the similarity to the original regex. A large change indicates low quality as such repairs may be far from what the user intended. Figure 9a shows a histogram plotting the number of regexes against the edit distances to their repair results by REMEDY-h. Most of the regexes were repaired within the small edit distances, with about 81% repaired within edit distance 12.



(a) Edit distances. (b) Percentages of changes.

Fig. 9: Histograms for repair quality.

We also measure the ratio of changes, i.e., the size of the regex portion changed by its repair divided by the size of the entire regex. Figure 9b shows a histogram plotting the number of regexes against their ratios of changes. We observe that most repairs are close to the original regexes, with the average ratio of change being 24.3%.

We discuss some typical cases of repairs that we observed in our experiments. For example, the data set contained vulnerable regexes that use positive lookaheads to assert an appearance of some keyword, e.g., $\cdot^*(?=[\ \ ]^*[;])\cdot^*$. For this, REMEDY returns the repaired invulnerable regex $[\hat{}\,;]^*[;]\cdot^*$, which is semantically equivalent to the original, with the edit distance of 7. We also refer to the XML example from Section II as an exemplar repair case that we observed in our experiments. In summary, REMEDY *can produce repaired regexes that have high-similarity, and therefore of high-quality*.

D. RQ3: Effect of the Optimization

We evaluate the effectiveness of the optimization described in Section V-C. The comparison of the running times of REMEDY and REMEDY-o are shown in Figure 10. We note that, in 23 cases, REMEDY-o solved the instance within the time limit while REMEDY could not, and conversely in 19 cases, REMEDY solved the instance within the time limit while REMEDY-o could not.

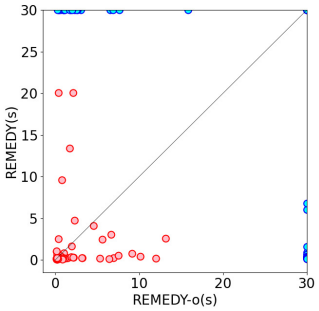


Fig. 10: Optimization effect.

We have observed that REMEDY-o often outperformed REMEDY for regexes that violate the RWS1U condition at many places. For example, for the regex

$$(?=[\hat{}\,,])\cdot^*,(?=[\hat{}\,,])\cdot^*,(?=[\hat{}\,,])\cdot^*,(?=[\hat{}\,,])\cdot^*,\cdot^+$$

the desired template is one in which the first four any character, i.e., ·, is replaced with a hole. REMEDY reaches such a template only after trying $2^4 - 1$ many other templates, whereas REMEDY-o reaches it immediately. Conversely, in cases where REMEDY performed better, we have observed that the repair benefits from templates that replace the non-RWS1U-violating parts of the regex with holes. Since the optimization prevents addHoles from replacing such parts with holes, it can negatively affect the performance in such cases.

In summary, *the optimization helps* REMEDY *to repair regexes that violate the RWS1U condition at many parts, while it negatively affects cases where non-RWS1U-violating parts should be repaired*. Thus, *running both* REMEDY *and* REMEDY*-o, i.e.,* REMEDY*-h, achieves better performance than running one of them alone*.

E. Comparison to Other State-of-the-art Tools

Table I summarizes the characteristics of different PBE tools for generating regexes, and compare them with our tool REMEDY. The **Invulnerability Guarantees** column shows that REMEDY is the only one to guarantee the invulnerability. Indeed, for AlphaRegex and RFixer, AlphaRegex generates vulnerable regexes, e.g., some regexes shown in Table 3 of [20] are vulnerable, and RFixer often generates vulnerable regexes as reported by Li et al. [24]. For FlashRegex, we could not confirm whether FlashRegex actually generates vulnerable regexes or not because FlashRegex is not publicly available (only the dataset is available from the GitHub repository). Additionally, we have contacted the authors, but the implementation was not available. However, FlashRegex claims to generate invulnerable regexes by only generating deterministic (i.e., 1-unambiguous) regexes, which unfortunately is insufficient for guaranteeing invulnerability as we have shown in Section IV-A.

The **Real-world Extensions** column shows that REMEDY is the only one to support real-world extensions. The other state-of-the-art tools, i.e., AlphaRegex, RFixer, and FlashRegex, only support pure regexes and supporting real-world extensions is out of scope for their work as mentioned in their papers. Additionally, supporting them would require a substantial overhaul as described in Sections III, IV, and V.

In summary, REMEDY *improves the other state-of-the-art tools from a theoretical point of view, and is the only one to support all characteristics*. We emphasize that all the other state-of-the-art tools can repair none of the regexes used in the evaluation.

Furthermore, we have compared REMEDY against the DFA-based approach of [41]. Their approach is not PBE but claims to produce an invulnerable pure regex that is semantically equivalent to the given pure regex. We performed an experiment by using 100 pure regexes randomly selected from the data set of [7], and compare the size of the regex repaired by the DFA-based approach and REMEDY.

We observed that (1) 100/100 of the regexes repaired by REMEDY are more concise than those of the DFA-based approach, and (2) the size of the regex repaired by the DFA-based approach is 37.3 times larger than that of REMEDY on average. We observe that this is partly because REMEDY can use real-world extensions (even for repairing pure regexes), and also because the DFA-based approach ensures semantic equivalence, which is often undesirable (cf. Section VII). Note that a semantics-preserving DFA conversion can generate exponentially large DFAs. For example, for the regex `·*·*=`, which is vulnerable, REMEDY returns the repaired invulnerable regex `··*(?<=[=])`. On the other hand, the one produced by the DFA-based approach is

```
[^=]*[=]([=]|[^=][^=]*[=])*
```

As another example, for the vulnerable regex

```
<span[^>]*font-style:italic[^>]*>
```

REMEDY returns the repaired invulnerable regex

```
<span([^".1-8B-Y\[\\\]^b-dfh-y]*)
        font-style:italic([^>]*)>
```

while the one returned by the DFA-based approach is of size 73,433,094. In summary, compared to the DFA-based approach, REMEDY *can find simpler and more understandable regexes*.

F. Availability

Our tool is available in [42].

VII. LIMITATIONS AND FUTURE WORK

We discuss some limitations of our approach and directions for future work. The first limitation is that we do not consider *extraction* of captured strings. This is a common limitation in regex repair and synthesis and many other recent works also do not support extraction [20], [22]–[24].

Extraction is especially problematic for real-world regexes as which string is captured in a lookahead is regex engine dependent.[3] A recent work has proposed an approach to cope with the issue in the context of symbolic execution [44] that involves executing an actual regex engine. But such an approach would be less ideal for repairs where we want to generate a regex that is correct and invulnerable in all contexts. We leave as future work to investigate the support for extraction. It is important to note that our formal definition of vulnerability considers all possible captures that can happen in a lookahead, and thus our approach is regex-engine-*in*dependently sound with respect to invulnerability.

The second limitation is the lack of support for *semantic equivalence*. As in other PBE methods, we consider the use case where the given regex is incorrect or only partly built. As argued by others [7], [13], [24], often, semantic equivalence is too strong to use in practice and PBE is better at reflecting users' intentions. But in future work, we would like to also

[3]It can even cause differences in the matching results in the rare cases where strings captured in lookaheads are backreferenced. For example, matching `(?=(a*)+)\1a` with `a` succeeds in Python's *re* and PCRE, but fails in ECMAScript [43].

TABLE I: A comparison of current state-of-the-art tools. ✓ and ✗ indicate that the tool has and does not have the characteristic, respectively. **Binary Alphabet** and **Multiple Alphabet** indicate that the tool can synthesize a regex over binary and multiple alphabets, respectively. **Correctness Guarantees** indicates that the tool guarantees that synthesized regexes are consistent with all examples. **Invulnerability Guarantees** indicates that the tool guarantees that synthesized regexes are not ReDoS vulnerable. **Real-world Extensions** indicates that the tool can support real-world extensions.

Tool	Binary Alphabet	Multiple Alphabet	Correctness Guarantees	Invulnerability Guarantees	Real-world Extensions
REMEDY	✓	✓	✓	✓	✓
AlphaRegex [20]	✓	✗	✓	✗	✗
RFixer [22]	✓	✓	✓	✗	✗
FlashRegex [24]	✓	✓	✓	✗	✗

support the case where the user is interested in only repairing vulnerability (e.g., because the regex is built correct by using some PBE method). However, whether a regex can be repaired to be invulnerable while preserving its semantics in general is an open problem. At least for real-world regexes, there are some reasons to doubt the possibility: semantic equivalence of real-world regexes is undecidable [26] and regexes with backreferences are not determinizable in general [45].

VIII. Related work

As remarked in Section I, there has been substantial work on PBE methods for synthesizing and repairing regexes [5], [19]–[24]. However, the existing methods do not support the real-world features such as lookarounds and backreferences. Furthermore, with the exception of [24] discussed below, the existing methods are not designed with resilience to ReDoS in mind and may generate vulnerable regexes.

A recent work by Li et al. [24] proposes a PBE regex synthesis and repair method that addresses vulnerability. Their method guarantees that the generated regex is deterministic (i.e., 1-unambiguous) [28], [29]. However, as we have shown in Section IV-A, 1-unambiguity is insufficient for invulnerability. Therefore, their method does not guarantee the invulnerability of the returned regexes. Also, their method only synthesizes and repairs pure regexes and does not support the real-world extensions. By contrast, our work supports real-world regexes and also formally guarantees the invulnerability of the synthesized regexes.

While not PBE, the work by van der Merwe et al. [41] proposes a technique based on DFA conversion and insertion of positive lookaheads to convert a vulnerable regex into an invulnerable one. However, they only consider the fragment with the positive lookahead extension. Also, as discussed in Section VI-E, the DFA-based approach can produce complex regexes that are hard to understand. In a similar vein, Cody-Kenny et al. [46] proposes a genetic-programming based method to convert a regex into one with more efficient matching. However, their method only supports pure regexes and does not guarantee invulnerability.

While our work concerns *repairing* vulnerability, there has been considerable work on the related problem of *detecting* vulnerability [12]–[16]. It is worth noting that while some (namely [12], [14], [16]) proposes to detect vulnerability formally rather than experimentally, no prior work on formal vulnerability detection supports the real-world extensions. Whether a sound-and-complete detection of vulnerability for real-world regexes is possible is an open question.

Another related work is a recent work by Davis et al. [47] that proposes a regex engine optimization to eliminate super-linear behavior of real-world regex matching at run time. Finally, a recent work by Loring et al. [44] presents a dynamic symbolic execution method for real-world regexes that addresses the regex-engine-dependent capturing issue mentioned in Section VII.

IX. Conclusion

We have presented a novel PBE regex repair method that guarantees the invulnerability of synthesized regexes and supports real-world regexes containing extended features of lookarounds, capturing groups, and backreferences. For this, we have defined a novel formal semantics of backtracking matching algorithm for real-world regexes and a formal definition of its time complexity. With them, we have defined the *first formal definition of ReDoS vulnerability for real-world regexes*. Additionally, we have presented a novel condition called *real-world strong 1-unambiguity* (RWS1U) which we proved to be sound for guaranteeing ReDoS invulnerability of real-world regexes, formalized the RWS1U repair problem and proved its NP-hardness. We have presented an algorithm for solving the RWS1U repair problem and experimentally evaluated its implementation, REMEDY, on a real-world data set. The evaluation have shown that REMEDY can repair vulnerable real-world regexes successfully and efficiently.

To the best of our knowledge, we are the first to tackle the ReDoS vulnerabilities for real-world regexes and the challenge of repairing them, whose theoretical properties are substantially different from that of pure regexes which are tackled by prior works [5], [19]–[24], [41], [46] that only considered pure regexes and/or did not concern ReDoS vulnerability.

Acknowledgements

We thank the anonymous reviewers for their useful comments. This work was supported by JSPS KAKENHI Grant Numbers 17H01720, 18K19787, 20H04162, and 20K20625.

REFERENCES

[1] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 282–293. [Online]. Available: https://doi.org/10.1145/2931037.2931073

[2] R. Cox, "Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...)," 2007, https://swtch.com/∼rsc/regexp/regexp1.html [Online; accessed 10-December-2021].

[3] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. USA: USENIX Association, 2011, p. 1.

[4] F. Yu, C.-Y. Shueh, C.-H. Lin, Y.-F. Chen, B.-Y. Wang, and T. Bultan, "Optimal sanitization synthesis for web application vulnerability repair," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 189–200. [Online]. Available: https://doi.org/10.1145/2931037.2931050

[5] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, May 2016.

[6] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. V. Jagadish, "Regular expression learning for information extraction," in *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*. Honolulu, Hawaii: Association for Computational Linguistics, Oct. 2008, pp. 21–30. [Online]. Available: https://www.aclweb.org/anthology/D08-1003

[7] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 246–256. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236027

[8] A. Weidman, "Regular expression denial of service - redos," 2017, https://owasp.org/www-community/attacks/Regular_expression_-Denial_of_Service_-_ReDoS [Online; accessed 10-December-2021].

[9] J. Graham-Cumming, "Outage postmortem july 20, 2016," 2016, https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016 [Online; accessed 10-December-2021].

[10] ——, "Details of the cloudflare outage on july 2, 2019," 2019, https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/ [Online; accessed 10-December-2021].

[11] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 361–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[12] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig, "Static detection of dos vulnerabilities in programs that use regular expressions," in *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 3–20. [Online]. Available: https://doi.org/10.1007/978-3-662-54580-5_1

[13] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "Rescue: Crafting regular expression dos attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 225–235. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238159

[14] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa," in *Implementation and Application of Automata*, Y.-S. Han and K. Salomaa, Eds. Cham: Springer International Publishing, 2016, pp. 322–334.

[15] J. Kirrage, A. Rathnayake, and H. Thielecke, "Static analysis for regular expression denial-of-service attacks," in *Network and System Security*, J. Lopez, X. Huang, and R. Sandhu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–148.

[16] S. Sugiyama and Y. Minamide, "Checking time linearity of regular expression matching based on backtracking," *Information and Media Technologies*, vol. 9, no. 3, pp. 222–232, 2014.

[17] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 1468–1484. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00062

[18] J. C. Davis, "The impact of regular expression denial of service (redos) in practice," 2018, https://infosecwriteups.com/introduction-987fdc4c7b0 [Online; accessed 10-December-2021].

[19] R. Alquezar and A. Sanfeliu, "Incremental grammatical inference from positive and negative data using unbiased finite state automata," in *In Proceedings of the ACL'02 Workshop on Unsupervised Lexical Acquisition*, 1994, pp. 291–300.

[20] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," *SIGPLAN Not.*, vol. 52, no. 3, pp. 70–80, Oct. 2016. [Online]. Available: https://doi.org/10.1145/3093335.2993244

[21] A. Bartoli, G. Davanzo, A. D. Lorenzo, E. Medvet, and E. Sorio, "Automatic synthesis of regular expressions from examples," *Computer*, vol. 47, no. 12, pp. 72–80, dec 2014.

[22] R. Pan, Q. Hu, G. Xu, and L. D'Antoni, "Automatic repair of regular expressions," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360565

[23] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 487–502. [Online]. Available: https://doi.org/10.1145/3385412.3385988

[24] Y. Li, Z. Xu, J. Cao, H. Chen, T. Ge, S.-C. Cheung, and H. Zhao, "Flashregex: Deducing anti-redos regexes from examples," in *Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering, ASE 2020, Virtual Event, Australia, September 21-25, 2020*, 2020. [Online]. Available: https://doi.org/10.1145/3324884.3416556

[25] J. E. F. Friedl, *Mastering Regular Expressions: Understand Your Data and Be More Productive (3th ed.)*. O'Reilly Media, 2006.

[26] D. D. Freydenberger, "Extended regular expressions: Succinctness and decidability," *Theory of Computing Systems*, vol. 53, no. 2, pp. 159–193, 2013. [Online]. Available: https://doi.org/10.1007/s00224-012-9389-0

[27] C. Koch and S. Scherzinger, "Attribute grammars for scalable query processing on xml streams," *The VLDB Journal*, vol. 16, no. 3, pp. 317–342, Jul. 2007. [Online]. Available: https://doi.org/10.1007/s00778-005-0169-1

[28] A. Brüggemann-Klein and D. Wood, "One-unambiguous regular languages," *Information and Computation*, vol. 142, no. 2, pp. 182–206, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S089054019792695X

[29] A. Brüggemann-Klein, "Unambiguity of extended regular expressions in sgml document grammars," in *Algorithms—ESA '93*, T. Lengauer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 73–84.

[30] G. Tiwari, "HTML/XML tag parsing using regex in Java," 2011, http://blog.gtiwari333.com/2011/12/htmlxml-tag-parsing-using-regex-in-java.html [Online; accessed 10-December-2021].

[31] L. G. Michael, J. Donohue, J. C. Davis, D. Lee, and F. Servant, "Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2019, pp. 415–426.

[32] RegExLib, 2021, https://regexlib.com/.

[33] J. Goyvaerts and S. Levithan, *Regular Expressions Cookbook (2nd ed.)*. O'Reilly Media, 2012.

[34] S. Medeiros, F. Mascarenhas, and R. Ierusalimschy, "From regexes to parsing expression grammars," *Sci. Comput. Program.*, vol. 93, pp. 3–18, 2014. [Online]. Available: https://doi.org/10.1016/j.scico.2012.11.006

[35] K. Thompson, "Programming techniques: Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, p. 419–422, Jun. 1968. [Online]. Available: https://doi.org/10.1145/363347.363387

[36] M. Sipser, *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[37] R. M. Karp, *Reducibility among Combinatorial Problems*. Boston, MA: Springer US, 1972, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-1-4684-2001-2_9

[38] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340.

[39] P. Wang and K. T. Stolee, "How well are regular expressions tested in the wild?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 668–678. [Online]. Available: https://doi.org/10.1145/3236024.3236072

[40] T. Miyazaki and Y. Minamide, "Derivatives of regular expressions with lookahead," *J. Inf. Process.*, vol. 27, pp. 422–430, 2019. [Online]. Available: https://doi.org/10.2197/ipsjjip.27.422

[41] B. van der Merwe, N. Weideman, and M. Berglund, "Turning evil regexes harmless," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, ser. SAICSIT '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3129416.3129440

[42] REMEDY, 2022, https://github.com/NariyoshiChida/SP2022.

[43] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 443–454. [Online]. Available: https://doi.org/10.1145/3338906.3338909

[44] B. Loring, D. Mitchell, and J. Kinder, "Sound regular expression semantics for dynamic symbolic execution of javascript," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 425–438. [Online]. Available: https://doi.org/10.1145/3314221.3314645

[45] M. L. Schmid, "Characterising regex languages by regular languages equipped with factor-referencing," *Information and Computation*, vol. 249, pp. 1 – 17, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540116000109

[46] B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, and M. O'Neill, "A search for improved performance in regular expressions," in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1280–1287. [Online]. Available: https://doi.org/10.1145/3071178.3071196

[47] J. C. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (redos)," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 543–559. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40001.2021.00032

APPENDIX A
FULL RULES OF THE FORMAL SEMANTICS

The full rules for deriving the matching relation $\rightsquigarrow$ is shown in Figure 11. We describe the rules for the pure regex features which were not explained in Section III. In the two rules for a set of characters, the regex $[C]$ tries to match the string w at the position p with the function capturing Γ. If the p-th character $w[p]$ is in the set of character C, then the matching succeeds returning the matching result $(p+1,\Gamma)$ (SET OF CHARACTERS). Otherwise, the character $w[p]$ does not match or the position is at the end of the string, and $\emptyset$ is returned as the matching result indicating the match failure (SET OF CHARACTERS FAILURE). The rules (EMPTY STRING), (CONCATENATION), (UNION) and (REPETITION) are self explanatory. Note that we avoid self looping in (REPETITION) by not repeating the match from the same position.

APPENDIX B
FULL RULES FOR GENERATING CONSISTENCY-WITH-EXAMPLE CONSTRAINTS

The full rules for generating the consistency-with-examples constraints is shown in Figure 12. The cases where the matching fails, that is, $(r,w,p,\Gamma,\phi) \dashrightarrow (\emptyset,\{(\bot,\bot,\phi)\})$, are omitted.

APPENDIX C
THE PROOF OF THEOREM IV.2

We first review EXACTCOVER.

Definition C.1 (Exact Cover)**.** Given a finite set $\mathscr{U}$ and $\mathscr{S} \subset \mathscr{P}(\mathscr{U})$, EXACTCOVER is the problem of deciding if there exists $\mathscr{S}' \subseteq \mathscr{S}$ such that for every $i \in \mathscr{U}$, there is a unique $S \in \mathscr{S}'$ such that $i \in S$.

Proof. We give a reduction from the exact cover to the repair problem. Let $\mathscr{S} = \{S_1, S_2, ..., S_k\}$. We create the following (decision version of) RWS1U repair problem:

- The alphabet $\Sigma = \mathscr{U}$;
- The set of positive examples $P = \mathscr{U}$;
- The set of negative examples $N = \emptyset$;
- The distance bound is $2k$; and
- The pre-repair expression $r_1 = r_{11}r_{12}$ where r_{11} and r_{12} are as defined below:

$$\begin{aligned} r_{11} &= \varepsilon(?{=}[S_1])^{2k}(\varepsilon)_1[S_1](\varepsilon)_2 \\ &\quad | \ \varepsilon(?{=}[S_2])^{2k}(\varepsilon)_3[S_2](\varepsilon)_4 \\ &\quad | \ ... \\ &\quad | \ \varepsilon(?{=}[S_k])^{2k}(\varepsilon)_{2k-1}[S_k](\varepsilon)_{2k} \\ r_{12} &= ((?!\backslash 1)|(?{=}\backslash 1\backslash 2))^{2k} \\ &\quad ((?!\backslash 3)|(?{=}\backslash 3\backslash 4))^{2k}...((?!\backslash 2k-1)|(?{=}\backslash 2k-1\backslash 2k))^{2k}. \end{aligned}$$

Here, r^{2k} is the expression obtained by concatenating r $2k$ times.

It is easy to see that this is a polynomial time reduction since the construction of r_1 can be done in time cubic in the size of the input EXACTCOVER instance. Also, note that the above is a valid RWS1U repair problem instance because $P = \mathscr{U} \subseteq L(r_1)$ and $L(r_1) \cap N = \emptyset$. We show that reduction is correct, that is, the input EXACTCOVER instance has a solution iff there exists r_2 satisfying conditions (1)-(3) of Definition IV.7 and $D(r_1, r_2) \leq 2k$. First, we show the only-if direction, let $\mathscr{S}' \subset \mathscr{S}$ be a solution to the EXACTCOVER instance. The repaired expression $r_2 = r_{21}r_{22}$ where $r_{22} = r_{12}$, and r_{21} is r_{11} but with each i-th head ε in the union replaced by $[\emptyset]$ iff $S_i \notin \mathscr{S}'$. Note that $D(r_1, r_2) = 2|\mathscr{S} \setminus \mathscr{S}'| \leq 2k$. Also, r_2 satisfies the RWS1U condition because for every $a \in \mathscr{U}$, there exists only one $S_i \in \mathscr{S}'$ such that $a \in S_i$, i.e., on any input string starting with a, we deterministically move to the i-th choice in the union (and there are no branches after that point). Also, r_2 correctly classifies the examples. To see this, consider an arbitrary $a \in P = \mathscr{U}$. Then, a is included in some $S_i \in \mathscr{S}'$. Therefore, the matching passes the r_{21} part with successful captures at indexes $2i-1$ and $2i$, and passes the r_{22} part because the negative lookahead $(?!\backslash j)$ succeeds for

$$\frac{p < |w| \qquad w[p] \in C}{([C], w, p, \Gamma) \rightsquigarrow \{(p+1, \Gamma)\}} \text{ (SET OF CHARACTERS)}$$

$$\frac{p \geq |w| \vee w[p] \notin C}{([C], w, p, \Gamma) \rightsquigarrow \emptyset} \text{ (SET OF CHARACTERS FAILURE)}$$

$$\frac{}{(\varepsilon, w, p, \Gamma) \rightsquigarrow \{(p, \Gamma)\}} \text{ (EMPTY STRING)}$$

$$\frac{(r_1, w, p, \Gamma) \rightsquigarrow \mathscr{N} \qquad \forall (p_i, \Gamma_i) \in \mathscr{N},\ (r_2, w, p_i, \Gamma_i) \rightsquigarrow \mathscr{N}_i}{(r_1 r_2, w, p, \Gamma) \rightsquigarrow \bigcup_{0 \leq i < |\mathscr{N}|} \mathscr{N}_i} \text{ (CONCATENATION)}$$

$$\frac{(r_1, w, p, \Gamma) \rightsquigarrow \mathscr{N} \qquad (r_2, w, p, \Gamma) \rightsquigarrow \mathscr{N}'}{(r_1 | r_2, w, p, \Gamma) \rightsquigarrow \mathscr{N} \cup \mathscr{N}'} \text{ (UNION)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathscr{N} \qquad \forall (p_i, \Gamma_i) \in (\mathscr{N} \backslash \{(p, \Gamma)\}),\ (r^*, w, p_i, \Gamma_i) \rightsquigarrow \mathscr{N}_i}{(r^*, w, p, \Gamma) \rightsquigarrow \{(p, \Gamma)\} \cup \bigcup_{0 \leq i < |(\mathscr{N} \backslash \{(p, \Gamma)\})|} \mathscr{N}_i} \text{ (REPETITION)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathscr{N}}{((r)_j, w, p, \Gamma) \rightsquigarrow \{(p_i, \Gamma_i[j \mapsto w[p..p_i])] \mid (p_i, \Gamma_i) \in \mathscr{N}\}} \text{ (CAPTURING GROUP)}$$

$$\frac{\Gamma(i) \neq \bot \qquad (\Gamma(i), w, p, \Gamma) \rightsquigarrow \mathscr{N}}{(\backslash i, w, p, \Gamma) \rightsquigarrow \mathscr{N}} \text{ (BACKREFERENCE)}$$

$$\frac{\Gamma(i) = \bot}{(\backslash i, w, p, \Gamma) \rightsquigarrow \emptyset} \text{ (BACKREFERENCE FAILURE)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathscr{N}}{((?{=}r), w, p, \Gamma) \rightsquigarrow \{(p, \Gamma') \mid (\_, \Gamma') \in \mathscr{N}\}} \text{ (POSITIVE LOOKAHEAD)}$$

$$\frac{(r, w, p, \Gamma) \rightsquigarrow \mathscr{N} \qquad \mathscr{N}' = ite(\mathscr{N} \neq \emptyset, \emptyset, \{(p, \Gamma)\})}{((?!r), w, p, \Gamma) \rightsquigarrow \mathscr{N}'} \text{ (NEGATIVE LOOKAHEAD)}$$

$$\frac{(x, w[p-|x|..p), 0, \Gamma) \rightsquigarrow \mathscr{N} \qquad \mathscr{N}' = ite(\mathscr{N} \neq \emptyset, \{(p, \Gamma)\}, \emptyset)}{((?{<=}x), w, p, \Gamma) \rightsquigarrow \mathscr{N}'} \text{ (POSITIVE LOOKBEHIND)}$$

$$\frac{(x, w[p-|x|..p), 0, \Gamma) \rightsquigarrow \mathscr{N} \qquad \mathscr{N}' = ite(\mathscr{N} \neq \emptyset, \emptyset, \{(p, \Gamma)\})}{((?{<}!x), w, p, \Gamma) \rightsquigarrow \mathscr{N}'} \text{ (NEGATIVE LOOKBEHIND)}$$

Fig. 11: Rules of the matching relation $\rightsquigarrow$

$$\frac{p < |w| \qquad w[p] \in C}{([C], w, p, \Gamma, \phi) \dashrightarrow (\{(p+1, \Gamma, \phi)\}, \emptyset)} \text{ (SET OF CHARACTERS)}$$

$$\frac{(t_1, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F}) \qquad \forall (p_i, \Gamma_i, \phi_i) \in \mathscr{S}.\ (t_2, w, p_i, \Gamma_i, \phi_i) \dashrightarrow (\mathscr{S}_i, \mathscr{F}_i)}{(t_1 t_2, w, p, \Gamma, \phi) \dashrightarrow (\bigcup_{0 \leq i < |\mathscr{S}|} \mathscr{S}_i, \mathscr{F} \cup \bigcup_{0 \leq i < |\mathscr{S}|} \mathscr{F}_i)} \text{ (CONCATENATION)}$$

$$\frac{(t_1, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}_1, \mathscr{F}_1) \qquad (t_2, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}_2, \mathscr{F}_2)}{(t_1 | t_2, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}_1 \cup \mathscr{S}_2, \mathscr{F}_1 \cup \mathscr{F}_2)} \text{ (UNION)}$$

$$\frac{(t, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F}) \qquad \forall (p_i, \Gamma_i, \phi_i) \in (\mathscr{S} \backslash \{(p, \Gamma, \_)\}).\ (t^*, w, p_i, \Gamma_i, \phi_i) \dashrightarrow (\mathscr{S}_i, \mathscr{F}_i)}{(t^*, w, p, \Gamma, \phi) \dashrightarrow (\{(p, \Gamma, \phi)\} \cup \bigcup_{0 \leq i < |\mathscr{S}|} \mathscr{S}_i, \emptyset)} \text{ (REPETITION)}$$

$$\frac{(t, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F})}{((t)_i, w, p, \Gamma, \phi) \dashrightarrow (\bigcup_{(p_i, \Gamma_i, \phi_{ci}) \in \mathscr{S}} (p_i, \Gamma_i[i \mapsto w[p..p_i)], \phi_{ci}), \mathscr{F})} \text{ (CAPTURING GROUP)}$$

$$\frac{\text{Let } x = \Gamma(i) \qquad x = w[p..p+|x|)}{(\backslash i, w, p, \Gamma, \phi) \dashrightarrow (\{(p+|x|, \Gamma, \phi)\}, \emptyset)} \text{ (BACKREFERENCE)}$$

$$\frac{(t, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F})}{((?{=}t), w, p, \Gamma, \phi) \dashrightarrow (\{(p, \Gamma', \phi') \mid (\_, \Gamma', \phi') \in \mathscr{S}\}, \mathscr{F})} \text{ (POSITIVE LOOKAHEAD)}$$

$$\frac{(t, w, p, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F})}{((?!t), w, p, \Gamma, \phi) \dashrightarrow (\{(p, \Gamma, \phi') \mid (\bot, \bot, \phi') \in \mathscr{F}\}, \{(\bot, \bot, \phi') \mid (\_, \_, \phi') \in \mathscr{S}\})} \text{ (NEGATIVE LOOKAHEAD)}$$

$$\frac{(x, w[p-|x|, p), 0, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F})}{((?{<=}x), w, p, \Gamma, \phi) \dashrightarrow (\{(p, \Gamma, \phi') \mid (p', \Gamma', \phi') \in \mathscr{S}\}, \mathscr{F})} \text{ (POSITIVE LOOKBEHIND)}$$

$$\frac{(x, w[p-|x|, p), 0, \Gamma, \phi) \dashrightarrow (\mathscr{S}, \mathscr{F})}{((?{<}!x), w, p, \Gamma, \phi) \dashrightarrow (\{(p, \Gamma, \phi') \mid (\bot, \bot, \phi') \in \mathscr{F}\}, \{(\bot, \bot, \phi') \mid (\_, \_, \phi') \in \mathscr{S}\})} \text{ (NEGATIVE LOOKBEHIND)}$$

$$\frac{\square \text{ is the } i\text{-th hole}}{(\square, w, p, \Gamma, \phi) \dashrightarrow (\{(p+1, \Gamma, \phi \wedge v_i^{w[p]})\}, \{(\bot, \bot, \phi \wedge \neg v_i^{w[p]})\})} \text{ (HOLE)}$$

Fig. 12: Rules for generating consistency-with-examples constraints.

all $j \neq i$ and the positive lookahead $(?{=}\backslash 2i-1 \backslash 2i)$ succeeds. Thus, r_2 is a correct repair.

We show the if direction. First, note that any valid repair of r_1 must preserve the k union choices of r_{11} because deleting any union choice would already exceed the cost $2k$. From this, it is not hard to see that the only possible change is to change the head ε in the union choices in r_{11}. For instance, it is useless to change $[S_i]$ to some r where $L(r)$ contains elements not in S_i because of the $2k$ many $(?{=}[S_i])$ preceding it. Note that changing $(?{=}[S_i])^{2k}$ would exceed the cost. Nor, can $[S_i]$ be changed to some r where $L(r)$ does not contain an element of S_i because of the capturing group $(\varepsilon)_{2i}$ and $(\varepsilon)_{2i-1}$ before and after $[S_i]$ and the check done in r_{12}. Note that changing any of the check in r_{12} would again exceed the cost. This also shows that the capturing groups $(\varepsilon)_{2i}$ and $(\varepsilon)_{2i-1}$ cannot be changed. Therefore, the only meaningful change that can be done is to change some of the head ε in r_{11} to some r. Note that for any r chosen here, by the RWS1U property, r_2 will not accept $\{a \mid aw \in L(r)\}$ as any input $a \in \{a \mid aw \in L(r)\}$ would direct the match algorithm deterministically to this choice but the match would fail when it proceeds to $[S_i]$. Therefore, the only change that can be done is to change it to some r such that $L(r) = \emptyset$. Then, from a successful repair r_2, we obtain the solution $\mathscr{S}'$ to the EXACTCOVER instance where $S_i \notin \mathscr{S}'$ iff the i-th head ε in r_{21} is changed to some r such that $L(r) = \emptyset$. □

APPENDIX D
CORRECTNESS OF RWS1U

In this section, we show that a regex that satisfies RWS1U is invulnerable. Before we go on with the main proof, we show that lookaheads that satisfy RWS1U runs in constant time to eliminate lookaheads from the later arguments.

Theorem D.1. *A regex that does not contain repetitions, unions, and backreferences runs in constant time.*

Proof. We prove that, for such a regex r, the size of $\mathscr{N}$ where $(r, w, 0, \emptyset) \rightsquigarrow \mathscr{N}$ is constant. This proof is by induction on the structure of the regex. □

By the definition of RWS1U, lookaheads that satisfy RWS1U do not contain repetitions and backreferences. Hence, lookaheads in a regex that satisfies RWS1U also run in constant time. For this, lookarounds, lookbehinds, and empty strings run in constant time. Also, they consume no characters. Thus, in what follows, without loss of generality, we assume that a regex does not contain empty strings, lookaheads, and lookbehinds.

We map a derivation tree to a *directed tree*.

Definition D.1 (Directed Tree)**.** A *directed graph* is a tuple $G = (V,E)$. Here, V is a finite set of vertices and E is a finite set of directed edges. A vertex $v = (\mathfrak{i}, p, \Gamma) \in V$ consists of a unique index $\mathfrak{i}$ and a matching result (p,Γ). A directed edge (*edge* for short) $e = (v_1, v_2) \in E$ consists of two vertices v_1 (often called *tail*) and v_2 (often called *head*). A *directed tree* is a directed graph that is of a tree shape (i.e., has no cycles and $|E| = |V| - 1$.).

We define the size of a directed tree $G = (V,E)$ as the size of E, i.e., $|G| = |E|$. We use the notation $d_{in}(v) = |\{v' \mid (v',v) \in E\}|$, $d_{out}(v) = |\{v' \mid (v,v') \in E\}|$, $\mathsf{leaf}(G) = \{v \mid v \in V \wedge d_{out}(v) = 0\}$, $\mathsf{root}(G) = v$ such that $v \in V \wedge d_{in}(v) = 0$, $\mathsf{tail}(e) = v$ and $\mathsf{head}(e) = v'$ for an edge $e = (v,v')$, and $E(v)$, where $v \in V$, for a set of heads, i.e., $E(v) = \{v' \mid (v,v') \in E\}$. For a tuple $t = (t_1, \cdots, t_n)$, we write $\#_i(t)$ for t_i, where $1 \leq i \leq n$.

We define a construction $AtoG$ from a derivation tree A to the directed tree G as follows: Here, $id()$ returns a fresh identifier, and, $a := b$ means that a is replaced with b. For a directed tree $G = (V,E)$ and $G' = (V',E')$ such that $V \cap V' = \emptyset$ and $v \in \mathsf{leaf}(G)$, we write $G[v \mapsto G']$ for the graph $(V \cup V' \setminus \{v\}, E \cup E' \cup \{(v', \mathsf{root}(G'))\} \setminus \{(v',v)\})$ where v' is the unique vertex such that $(v',v) \in E$. I.e., $G[v \mapsto G']$ is the graph obtained by replacing the leaf v of G by (the root of) G'. Additionally, we assume that $G_i = (V_i, E_i)$.

- Case (SET OF CHARACTERS). $G = (\{v_1, v_2\}, \{(v_1, v_2)\})$, where $v_1 = (id(), p, \Gamma)$ and $v_2 = (id(), p+1, \Gamma)$.
- Case (SET OF CHARACTERS FAILURE). $G = (\{v_1, v_2\}, \{(v_1, v_2)\})$, where $v_1 = (id(), p, \Gamma)$ and $v_2 = (id(), \emptyset, \Gamma)$.
- Case (CONCATENATION). Let $G_1 = AtoG((r_1, w, p, \Gamma) \rightsquigarrow \mathscr{N})$. For all $(p_i, \Gamma_i) \in \mathscr{N}$, there exists a vertex $(\_, p_i, \Gamma_i) \in \mathsf{leaf}(V)$. For all $(\_, p_i, \Gamma_i) \in \mathsf{leaf}(V)$, let $G_{2i} = AtoG((r_2, w, p_i, \Gamma_i) \rightsquigarrow \mathscr{N}_i)$, $G_1 := G_1[(\_, p_i, \Gamma_i) \mapsto G_{2i}]$. $G = (V_1 \cup \{v\}, E_1 \cup \{(v, \mathsf{root}(G_1))\})$ where $v = (id(), p, \Gamma)$.
- Case (UNION). Let $G_1 = AtoG((r_1, w, p, \Gamma) \rightsquigarrow \mathscr{N})$ and $G_2 = AtoG((r_2, w, p, \Gamma) \rightsquigarrow \mathscr{N}')$. $G = (V_1 \cup V_2 \cup \{v\}, E_1 \cup E_2 \cup \{(v, \mathsf{root}(G_1)), (v, \mathsf{root}(G_2))\})$, where $v = (id(), p, \Gamma)$.
- Case (REPETITION). Let $G_1 = AtoG((r, w, p, \Gamma) \rightsquigarrow \mathscr{N})$. For all $(p_i, \Gamma_i) \in \mathscr{N}$, there exists a vertex $(\_, p_i, \Gamma_i) \in \mathsf{leaf}(V_1)$. For all $(\_, p_i, \Gamma_i) \in leaf(V_1)$, let $G_{2i} = AtoG((r^*, w, p_i, \Gamma_i) \rightsquigarrow \mathscr{N}_i)$, $G_1 := G_1[(\_, p_i, \Gamma_i) \mapsto G_{2i}]$. $G = (V_1 \cup \{v_1, v_2\}, E_1 \cup \{(v_1, v_2), (v_1, \mathsf{root}(G_1))\})$, where $v_1 = (id(), p, \Gamma)$ and $v_2 = (id(), p, \Gamma)$.
- Case (CAPTURING GROUP). Let $G_1 = AtoG((r, w, p, \Gamma) \rightsquigarrow \mathscr{N})$. $G = (V_1 \cup \{v\}, E_1 \cup \{(v, \mathsf{root}(G_1))\})$.
- Case (BACKREFERENCE). Let $G_1 = AtoG((\Gamma(i), w, p, \Gamma) \rightsquigarrow \mathscr{N})$. $G = (V_1 \cup \{v\}, E_1 \cup \{(v, \mathsf{root}(G_1))\})$ where $v = (id(), p, \Gamma)$.
- Case (BACKREFERENCE FAILURE). $G = (\{v_1, v_2\}, \{(v_1, v_2)\})$, where $v_1 = (id(), p, \Gamma)$ and $v_2 = (id(), \emptyset, \Gamma)$.

Lemma D.1. *Given a derivation tree $(r, w, 0, \emptyset) \rightsquigarrow \mathscr{N}$. Let A be the derivation tree and $G = AtoG(A)$ be the directed tree. The size of the derivation tree A is greater than or equal to the size of the directed tree G.*

Proof. The proof is by induction on the structure of A. □

Definition D.2 (Main and Sub Branch)**.** Let $G = (V,E)$ be a directed tree. For each vertex $v \in V$, we say an edge $e \in E(v)$ is a *main branch* of v if $\forall e' \in E(v) \setminus \{e\}, \#_2(\mathsf{head}(e')) < \#_2(\mathsf{head}(e))$. Otherwise, we say the edge is *sub branch* of v.

Definition D.3 (Main Path)**.** We say a sequence of main branches $\mathfrak{p}_m = e_1 e_2 \cdots e_n$ is a *main path* if $\mathsf{head}(e_i) = \mathsf{tail}(e_{i+1})$ for $1 \leq i < n$, $\mathsf{tail}(e_1)$ is a root, i.e., $d_{in}(\mathsf{tail}(e_1)) = 0$, and, for every $e \in E(\mathsf{head}(e_n))$, e is a sub branch.

By the definition of the main path, there is at most one main path in a directed tree.

Lemma D.2. *Given a directed tree $G = (V,E)$, which is obtained by $AtoG((r, w, 0, \emptyset) \rightsquigarrow \mathscr{N})$. If G has a main path $\mathfrak{p}_m$, then the length $|\mathfrak{p}_m|$ is $O(|w|)$.*

Proof. By induction on the structure of r. The only interesting case is when r is a repetition, say, $r = r'^*$. We show that r'^* consumes at most $O(|w|)$ characters during the whole matching. Let n be the number of iterations of r'^* on $w[p_1..|w|)$ and $(r'^*, w, p_i, \Gamma_i) \rightsquigarrow \mathscr{N}_i$ be the i-th iteration, where $1 \leq i \leq n$. Let $e_1 e_2 \cdots e_n$ be the main path. For $1 \leq i < n$, $\#_2(\mathsf{tail}(e_i)) < \#_2(\mathsf{head}(e_i))$ because if $\mathsf{tail}(e_i) = \mathsf{head}(e_i)$, then it means that r' accepts an empty string and so it violates RWS1U because there are two or more paths to the first alphabet in r' or the next expressions. Hence, $\#_2(\mathsf{tail}(e_i)) < \#_2(\mathsf{head}(e_i))$ and r'^* consumes at most $O(|w|)$ characters. □

Definition D.4 (ε Subtree)**.** Let $G = (V,E)$. We say a subtree $G_\varepsilon = (V_\varepsilon, E_\varepsilon)$, where $V_\varepsilon \subseteq V$ and $E_\varepsilon \subseteq E$, is an *$\varepsilon$ subtree* if every $e \in E_\varepsilon$ is a sub branch, $|E_\varepsilon(\mathsf{root}(G_\varepsilon))| = 1$, $\#_2(\mathsf{tail}(e)) = \#_2(\mathsf{head}(e))$ or $\#_2(\mathsf{head}(e)) = \emptyset$ for every $e \in \{(v,v') \in E_\varepsilon \mid v \neq \mathsf{root}(G_\varepsilon)\}$, and, for every $v_1, v_2 \in V_\varepsilon$, where $v_1 \neq v_2$, there exists a sequence of E_ε edges $e_1 e_2 \cdots e_n$ such that $\mathsf{tail}(e_1) = v_1$, $\mathsf{head}(e_n) = v_2$, $\mathsf{head}(e_i) = \mathsf{tail}(e_{i+1})$ for $1 \leq i < n$.

Lemma D.3. *Given an ε subtree G_ε in $AtoG((r, w, p, \Gamma) \rightsquigarrow \mathscr{N})$. The size of the ε subtree $|G_\varepsilon|$ is constant.*

Proof. Suppose that $|G_\varepsilon|$ is not constant. Then, r contains a repetition r'^* and the repetition r'^* iterates at least twice because, if not, then the size is $O(|r|)$, i.e., constant. By the definition of ε subtrees, the repetition r'^* does not consume any character and this means that r' accepts an empty string. However, it means that the repetition r'^* violates the RWS1U condition because there are two or more paths to the first

character in r' or the next expression. Thus, r'^* iterates at most once. Hence, $|G_\varepsilon|$ is constant. □

Lemma D.4. *Let $G = AtoG((r,w,p,\Gamma) \rightsquigarrow \mathscr{N})$. Then the number of ε subtrees in G is $O(|w|)$.*

Proof. We show that the number of ε subtrees is $O(|\mathfrak{p}_m|)$, i.e., $O(|w|)$. The proof is by induction on structure of derivation trees. Here, we only focuses on the case (REPETITION). In the case of (REPETITION), let m be the number of iterations of $(r'^*,w,p',\Gamma') \rightsquigarrow \mathscr{N}'$. For the i-th iteration, let $(r',w,p_i,\Gamma_i) \rightsquigarrow \mathscr{N}_i'$, where $1 \leq i \leq m$, $p_1 = p'$, and $\Gamma_1 = \Gamma'$. Then, by inductive hypothesis, each derivation tree $(r',w,p_i,\Gamma_i) \rightsquigarrow \mathscr{N}_i'$ satisfies the assertion. Hence, the assertion holds. □

Lemma D.5. *Let $G = (V,E) = AtoG((r,w,0,\emptyset) \rightsquigarrow \mathscr{N})$. For all edges $e \in E$, e belongs to either a main path or an ε subtree.*

Proof. The proof is by induction on the structure of derivation trees. □

Theorem D.2. *Given a directed tree $G = (V,E)$. The size of the directed tree $|G|$ is $O(|w|)$.*

Proof. By Lemma D.5, G consists of a main path and ε subtrees. By Lemma D.2, the size of the main path is $O(|w|)$. By Lemma D.3, the size of the ε subtrees is $O(1)$, and by Lemma D.4, the number of ε subtrees is $O(|w|)$. Hence the size of G is $O(|w| \times 1 + 1 \times |w|) = O(|w|)$. □

Finally, we are now ready to proof Theorem IV.1.

Proof. (Proof of Theorem IV.1) Immediate from Lemma D.1 and Theorem D.2. □

APPENDIX E
INSUFFICIENCY OF DETERMINISTIC REGEXES

The details of why [24] is insufficient for guaranteeing unambiguity is as follows. The idea of [24] for repairing ReDoS-vulnerability is to synthesize so-called "deterministic" (also called 1-unambiguous [28], [29]) regexes. The definition of deterministic regex is as follows:

Definition E.1 (Deterministic, Definition 2.1 in [28], [29])**.** A regex E is *deterministic* (or *1-unambiguous*) iff, for all words $u,v,w \in \Pi^*$ and all symbols $x,y \in \Pi$,

$$uxv, uyw \in L(E') \wedge x \neq y \Rightarrow x^\natural \neq y^\natural.$$

Here, Π is the subscripted alphabet $\{a_i \mid a \in \Sigma, i \in \mathbb{N}\}$, $x^\natural$ is the character obtained by dropping the subscript of $x \in \Pi$ (e.g., $(a_1)^\natural = a$), and E' is E but with each characters in E annotated with distinct subscripts (e.g., if $E = ab(a|b)c$, then $E' = a_1b_1(a_2|b_2)c_1$).

Intuitively, a regex E is deterministic iff for any $w \in L(E)$, there is a unique subscripted word $v \in L(E')$ such that $v^\natural = w$. Thus, the vulnerable regex $E = (a^*)^*$ satisfies the definition because its subscripted regex E' is $(a_1^*)^*$ and $L(E') = \{\varepsilon, a_1, a_1a_1, ...\}$ (i.e., there are no $uxv, uyw \in L(E')$ such that $x \neq y$). Hence, $(a^*)^*$ is a deterministic regex while it is vulnerable as we have shown in Section III-B. Consequently, synthesizing deterministic regexes is insufficient for guaranteeing ReDoS invulnerability.