REVEALER: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities

Yinxi Liu, Mingxue Zhang, and Wei Meng
Chinese University of Hong Kong
{yxliu, mxzhang, wei}@cse.cuhk.edu.hk

***Abstract*—Regular expression Denial-of-Service (ReDoS) is a class of algorithmic complexity attacks. Attackers can craft particular strings to trigger the worst-case super-linear matching time of some vulnerable regular expressions (regex) with extended features that are commonly supported by popular programming languages. ReDoS attacks can severely degrade the performance of web applications, which extensively employ regexes in their server-side logic. Nevertheless, the characteristics of vulnerable regexes with extended features remain understudied, making it difficult to mitigate or even detect such vulnerabilities.**

In this paper, we aim to model vulnerable regex patterns generated by popular regex engines and craft attack strings accordingly. Our characterization fully supports the analysis of regexes with any extended feature. We develop REVEALER to detect vulnerable structures presented in any given regex and generate attack strings to exploit the corresponding vulnerabilities. REVEALER takes a hybrid approach. It first statically locates potential vulnerable structures of a regex, then dynamically verifies whether the vulnerabilities can be triggered or not, and finally crafts attack strings that can lead to recursive backtracking. By combining both static analysis and dynamic analysis, REVEALER can accurately and efficiently generate exploits in a limited amount of time. It can further offer mitigation suggestions based on the structural information it identifies.

We implemented a prototype of REVEALER for Java. We evaluated REVEALER over a dataset with 29,088 regexes, and compared it with three state-of-the-art tools. The evaluation shows that REVEALER considerably outperformed all the existing tools—REVEALER can detect all 237 vulnerabilities that can be detected by any other tool, find 213 new vulnerabilities, and beat the best tool by 140.64%. We further demonstrate that REVEALER successfully detected 45 vulnerable regexes in popular real-world applications. Our evaluation demonstrates that REVEALER is both effective and efficient in detecting and exploiting ReDoS vulnerabilities.

I. INTRODUCTION

Regular expression (regex) is a powerful technique developed in formal language theory to denote a regular language [24] that can be used to describe certain patterns. Regexes have been extensively used in modern software, including databases, text editors, search engines, *etc.* Modern regex engines are augmented with extended features (*e.g.*, conditionals, named groups, *etc.* [17]) for advanced pattern matching and processing.

Regular Expression Denial-of-Service (ReDoS) [18, 27] attacks are a form of algorithmic asymmetric DoS attacks [8] targeting the CPU resource of a victim server. Since developers may craft regexes that exhibit super-linear (*e.g.*, exponential) worst-case matching time, a specially-crafted (short) input can spend a server as much as several or more seconds on matching a vulnerable regex. Since it is hard to split the matching process of a regex into multiple independent steps, a single malicious request can block or freeze a (main) thread for a long time [32]. By supplying multiple such attack inputs, an attacker can significantly lower the availability of the victim server. Since regexes are supported by most languages and are widely used in modern applications, especially web/mobile applications that extensively rely on regexes to process untrusted user inputs, such attacks can have a huge impact on a vast number of applications on the Internet.

The theory of detecting ReDoS vulnerabilities in classical regular expressions based on statically analyzing NFA [38] has been well established. However, extended regular expressions can no longer be represented by an NFA. Hence, researchers have been working on establishing new theories to model the extended features [4, 9, 22]. Nevertheless, to the best of our knowledge, no static analyzer is able to fully support all extended features so far. Further, static analyzers usually report many false positives. How to well model the ReDoS problem in extended regular expressions still remains as a challenge.

Researchers also proposed to use dynamic approaches, *e.g.*, fuzzing, to generate malicious inputs to detect ReDoS and other types of algorithmic DoS vulnerabilities [5, 28, 30]. Such approaches are very effective in finding easy-to-trigger ReDoS vulnerabilities, and are able to find vulnerable *extended* regexes, because they usually do not require knowledge of the regex structures. However, they are very limited in detecting complex ones, because it is difficult to generate the correct sequences of inputs to reach the vulnerable parts of a regex without understanding its structure and features. Further, they usually require a high computation cost for searching different inputs.

In this paper, we aim to tackle the challenge of automatically detecting and exploiting ReDoS vulnerabilities in *extended* regexes. We take a hybrid approach combining both static analysis and dynamic analysis methods.

We first statically model the ReDoS vulnerabilities in an extended NFA (e-NFA) [30], which is the structure modern regex engines use to represent an extended regex. Our theory is inspired by the vulnerable NFA patterns defined in [38]. We model two types of vulnerable e-NFA patterns in exponential worst-case time complexity, and one type in polynomial worst-case time complexity. Such models allow us to statically locate the potential vulnerable structures in a regex.

Next, we dynamically exploit the potential vulnerability by generating *attack strings*. In particular, we focus on generating the *attack core*, whose repetitions in the attack string can lead to catastrophic backtracking [3] when the engine fails to match

the *attack suffix*. Our approach is different from fuzzing as we do not generate attack strings by mutating input seeds. Instead, we simulate the matching process of an extended regex on top of an E-TREE, which is our simplified representation of an e-NFA. Our match simulator can construct a match string for one or multiple sub regular expressions. By simulating the match of multiple subexpressions that are detected from the vulnerable e-NFA structure, we can effectively generate the attack core. Similarly, we can also accurately generate the *attack prefix* for reaching the vulnerable regex pattern, and the attack suffix that causes backtracking on match failure.

We develop REVEALER, a system for automatically detecting and exploiting ReDoS vulnerabilities. REVEALER takes a regex as input, and can produce an attack string that can exploit the ReDoS vulnerability in the regex, if any. It incorporates a static analysis for locating the vulnerable subexpressions and a dynamic analysis for producing the attack strings and validating the potential vulnerabilities. For each potential vulnerable regex, it dynamically tests it with the attack string for super-linear matching steps at runtime. Such dynamic verification helps it exclude any false positives that may be reported by the static analysis. Further, it can also report the vulnerable subexpression in the regex to help the developers identify and fix the vulnerability. We implemented a prototype of REVEALER based on the Java 8 regex engine. We will make the source code of our prototype implementation publicly available.

We systematically evaluated REVEALER with a benchmark dataset containing 29,088 regexes, and compared it with three state-of-the-art ReDoS detection tools. REVEALER significantly outperformed all three tools by detecting 213 *previously unknown* ReDoS vulnerabilities, and *all 237 known* ones detected by the other tools. It beat the best performing tool by 140.64% in our evaluation, and took several orders or magnitude less time on analyzing one regex. We further applied REVEALER to 178 popular open-source projects on GitHub, and detected 45 *new* vulnerabilities. We responsibly disclosed our detected vulnerabilities to the relevant developers. The evaluation results demonstrate that REVEALER can both *effectively* and *efficiently* detect and exploit ReDoS vulnerabilities.

In summary, this paper makes the following contributions:

- We statically modeled ReDoS vulnerabilities for extended regular expressions based on extended NFA.
- We developed REVEALER, an effective and efficient system for automatically detecting and exploiting ReDoS vulnerabilities by using a hybrid approach.
- Using REVEALER, we detected 213 previously unknown ReDoS vulnerabilities in a benchmark dataset, and 45 previously unknown ReDoS vulnerabilities in popular real-world open-source applications.

II. BACKGROUND

We introduce the necessary background related to ReDoS and its existing mitigation approaches in this section.

A. ReDoS Attacks

In general, ReDoS vulnerabilities are caused because the worst-case time complexity of regex matching algorithm in modern regex engines is super-linear with the length of input.

Traditionally, regex engines accept only *classical regular expression*, which is the expression of a *regular language* [24]. From *Kleene's theorem* [39], a regular language can be transferred to an equivalent *nondeterministic finite automaton (NFA)*. Therefore, we can construct a NFA for each classical regex. The regex matching algorithm then becomes the process of verifying whether the NFA accepts a certain input string or not, whose worst-case complexity is only linear with the input length because it visits each input symbol once and each visit takes constant time.

However, modern regex engines add support for extended features, which require a matching algorithm with super-linear worst-case complexity. Specifically, in the *Chomsky hierarchy*, classical regular expressions are generated by *regular grammar*s, which belong to *context-free grammar*. But the expressive power of many extended features goes beyond context-free grammars and can only be described by *context-sensitive grammar*s [9]. We use the term *extended regular expression* to denote the regular expression containing such features[1]. For example, the extended feature "backreferences" can match the same text previously matched in a capturing group. A regex example using backreferences is `(.+)&\1`, which can match "`Hello&Hello`" and "`World&World`" but not "`Hello&World`". Regex matching is *NP-hard* when regexes are allowed to have backreferences, as discussed in [16].

Therefore, to fully support these complex extended features, modern engines first parse the input regular expression into an NFA like structure (called *e-NFA* in [30]), and then perform a *backtracking search* on top of this structure according to the input string. Since Shen *et al.* [30] did not provide a detailed definition of e-NFA, we will provide one later in §III-A. In particular, backtracking search is used by modern regex engines when a regex contains optional quantifiers or alternation constructs [15]. Such a behavior has been discussed in [30] and [17].

The search-based algorithm has potential performance issues. It may lead to *catastrophic backtracking* since it needs to check all branches in the e-NFA. For example, regex `^'(a+)*'$` contains a vulnerable subexpression `(a+)*`. Suppose this subexpression matches a repeated string "`aaaaaa...`" of length n, and its next subexpression `'` fails to match the following symbol '`b`', the regex engine would perform backtracking on the already matched string "`aaaaaa...`". Each symbol '`a`' in the repeated string can be matched with either the subexpression `(a+)` of the outer quantifier `*`, or the subexpression `a` of the inner quantifier `+`. Since the length of the repeated string is n, there are 2^n possible backtracking searches until the engine declares a failure. In contrast, the NFA-based matching algorithm would terminate immediately when it fails to match the symbol '`b`' using linear time.

Nevertheless, only GO and Rust adopt NFA based matching algorithm while sacrificing the support for some extended features. Other popular languages, including JavaScript, Python, Java, C++, C#, PHP, Perl, and Ruby, implement the e-NFA based regex engines with super-linear worst-case complexity to support extended regular expressions [11], which makes ReDoS

[1]Some extended features like "lookaround" can be simulated in a regular language by treating its surrounding subexpressions as parts of the language, so they are not considered as such features.

a common problem.

B. Mitigation of ReDoS Attacks

ReDoS attacks are difficult to prevent because 1) developers might write vulnerable regex patterns; and 2) regex engines need to support backtracking and extended features. In practice, developers lack tools to validate the security of regular expressions they write, and focus on correctness while neglecting the performance when writing regexes [32]. A survey [23] shows that only 38% developers are aware of ReDoS attacks, and even for those that know such problems, there is a lack of tools or knowledge to help detect such vulnerabilities.

Regex engines can prevent ReDoS attacks by disabling extended features that cannot be represented by context-free grammar, but this can significantly limit their functionalities. Alternatively, some engines limit the number of backtracking searches (PHP and Perl) or set timeout (the .NET framework) to stop ReDoS attacks [11]. However, it is difficult to configure a limit that does not break legitimate regexes.

Recently, researchers focus on detecting vulnerable regexes. In most of the cases, developers can change the vulnerable regex to a vulnerability-free one while preserving the functionality. In some other cases where attack strings that can trigger ReDoS are under certain patterns (§IV-D), developers can filter such patterns before sending the input string to the regex. Specifically, there are two classes of approaches on detecting vulnerable regexes. We discuss the latest development of them below.

1) Static Analyses: Static analyses detect ReDoS vulnerabilities by identifying vulnerable patterns in the regex. Existing static methods usually build a NFA-based parse structure, and find vulnerable patterns in the structure. Detecting vulnerable patterns in NFA has been a theoretically well studied problem [38]. For extended regular expressions that cannot be represented by an NFA, researchers try to add support for some extended features in the parse structure, such as additional support for "backreferences" [9]; "capture groups" [4]; or "capture groups", "greedy quantifiers" and "lazy quantifiers" [22]. Nevertheless, no parse structure can support all extended features so far. Further, static analysis approaches usually have false positives.

2) Dynamic Analyses: Dynamic analyses (or fuzzers) detect ReDoS vulnerabilities by generating inputs to trigger the worst-case matching. Such approaches usually do not require knowledge of the regex structures and are not restricted by context-free grammars. Therefore, they are able to find vulnerable *extended* regular expressions. For example, both SlowFuzz [28] and ReScue [30] get seeds from the regex engine and use genetic algorithms to generate inputs.

Fuzzing methods do not work well for finding complex vulnerabilities. Since vulnerable patterns in e-NFA have not been well studied, fuzzers usually use only general information like e-NFA state coverage rate, alphabet strings in regexes, and the matching steps of a certain input string. Therefore, it is difficult for these methods to find specially formatted worst-case inputs for complex vulnerable regexes (*e.g.*, in §VI-C3, the fuzzing tool ReScue failed to generate the correct prefix, but all the other static analysis tools succeeded). In addition, while being effective, fuzzers may spend a lot of time on generating inputs that cannot trigger the vulnerabilities.

III. Problem Statement

In this section, we first provide the necessary definitions in our approach (§III-A), then discuss our research goals and the research challenges (§III-B).

A. Definitions

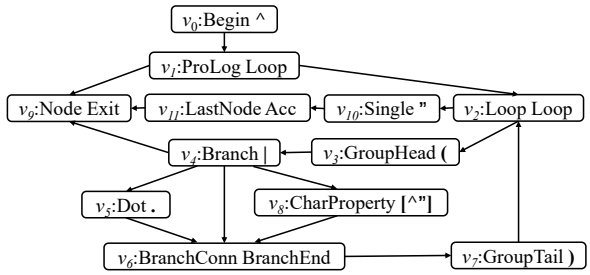
We present our definition of e-NFA, which was previously defined in [30] informally. We formalize the common implementations of modern regex engines, thus support all extended features naturally. The definition includes the syntax of an e-NFA (*e.g.* states, transition functions, *etc.*), the semantics of an e-NFA, and the e-NFA match process. We do not formalize the process that translates a regex to an e-NFA, because our method depends directly on the e-NFA instead of the specific translation implementation in a regex engine.

Definition 1 (e-NFA)**.** *An e-NFA A is represented by a 6-tuple $(V, \Sigma, \Delta, \Delta', v_0, v_f)$ where V is a finite set of states and Σ is a finite alphabet of symbols. Let $0 \leq p \leq |s|$ be the position in the currently processed input string s, and t as a snapshot of global matching information when the engine runs to the current state v. A state v includes several attributes: a set AS_v of strings acceptable for a match, its current match count c_v, its minimum required match count c_v^{min}, and its maximum allowed match count c_v^{max}. The latter two represent the match count requirements of the state. A state v has also two corresponding functions: the match function δ_v in Δ, and the transition function δ'_v in Δ'. $\delta_v : (s, p, t) \to (S_v, p', t')$ produces the status $S_v \in \{0, 1, -1\}$ of state v, and updates global information p and t if necessary. $\delta'_v : (S_v, t) \to v'$ produces the next state to transit to from S_v and t. Here, $v_0 \in V$ is the initial state, and $v_f \in V$ is the only accepting state.*

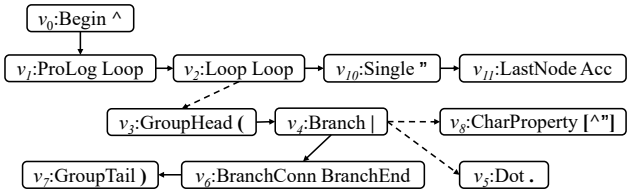
1) The e-NFA Semantics: The e-NFA matches a string s if there exists a sequence of transitions $\tau = v_0 \to \cdots \to v_f$ that, starting at $(v_0, p = 0, t = \epsilon)$ and following the transitions lead to the accepting state. We call τ as a matching path, and s is a match string of τ. To consume the substring $s[p_i : p_j]$ while matching s, the e-NFA takes a sequence of transitions $v_i \to \cdots \to v_j$ on the path τ. This sequence of transitions (we simplify as $v_i, \ldots, v_j$) represents a (sub-) matching path of the substring $s[p_i : p_j]$.

2) The e-NFA Match Process: Regex engines conduct a match with a stack L that stores a sequence of states v. The stack starts with the initial state v_0, with $L_0 = \{v_0\}$ by default. For each transition $v \to v'$, the engine iteratively computes the status S_v of the stack's top element v and pops it if S_v is either 1 (v is matched successfully) or -1 (the match fails). The engine stops popping when S_v is 0, which indicates that the engine cannot determine whether v could be matched at this moment, or v is currently matched but could possibly be backtracked later. Then, it computes the next state $v' \leftarrow \delta'_v(S_v, t)$ and pushes it onto the stack. The match process ends successfully when the stack gets cleared and A reaches v_f.

We use the regex example `^(.|[^"])*"` in Figure 1a to illustrate the above process. Assume the engine tries to match it with a string `"aa"`. ① Before the engine consumes any character (*i.e.*, from v_0 to the branch state v_4), it first pops v_0, then pushes v_1, v_2, v_3, v_4 without popping any state. When v is v_4,



(a) The e-NFA implementation in the Java 8 engine.



(b) An E-TREE example.

Fig. 1: An example of regex `^(.|[^"])*"`

the engine can select either v_5 or v_8 as the next state v'. ② Assume the engine always transits to v_5 first by pushing v_5. Its corresponding sub pattern "Dot" matches the first character `'a'`[2]. Its match function δ_{v_5} increments p to 1, and sets its status S_{v_5} to 1, which makes the engine pop v_5 and push v_6. Since $v = v_6$ is the branch end, the engine pops v_6, v_4, and pushes v_7. Similarly, it then pops v_7 and v_3. ③ Now the loop state v_2 is the top element, the engine attempts to match another repetition of it. With similar steps, it returns to v_2 again with $p = 2$. ④ The engine cannot find another character to match either v_4 or v_{10}, and has to backtrack by decrementing p to 1. This time, it transits to v_4 and v_8 (v_5 was marked as failed), and matches $s[1]$. After transiting back to v_2, it has to backtrack again by decrementing p to 0. It matches $p[0]$ with v_8[3], and $p[1]$ with first v_5 and then v_8 in the next backtracking. The engine can match each `'a'` with two states. With n `'a'`, it has to try 2^n possible matching paths until it finally fails.

3) The Match Function: $\delta_v : (s, p, t) \rightarrow (S_v, p', t')$ has two behaviors. First, when the acceptable string set AS_v of the state v (*e.g.*, v_{10}: string, v_8: set operation, and v_5: the 'Dot' feature) includes all strings that v can match, δ_v matches a string $s[p : p']$ and updates $p \rightarrow p', S_v \rightarrow 1$ if $s[p : p']$ is in AS_v. Second, when the AS_v of the state v is empty, δ_v determines the match using the information in t, which includes the status of other states (*e.g.*, v_4 is matched if v_5 or v_8 is matched) and strings already matched (*i.e.*, capturing groups[4] record matched strings in t for backreferences). Note that the need of run-time variable t makes e-NFA a context-sensitive grammar. However, the output of the function δ_v is determined by s, p, and t.

4) The Transition Function: $\delta'_v : (S_v, t) \rightarrow v'$ determines v' using the current match count c_v (*e.g.*, $c_v = 0$ if state v has not been matched yet) for each state v stored in t. c_v indicates the current match status of v. The transition from v to v' is an *inclusion transition* if the match of v depends on v' (*i.e.*, the match substring of v includes that of v'); and is a *connection transition* if v and v' are matched independently.

Formally, each e-NFA state v represents a subexpression in the input regex r, which we call the subexpression of state v. Let $r[i : j]$ and $r[i' : j']$ represent the subexpressions of v and v', respectively. In Figure 1a, the subexpression of the loop state v_2 is `(.|[^"])*`, which is $r[1 : 10]$; the subexpression of the group head state v_3 is `(`, which is $r[1 : 2]$; the subexpression of the single string v_{10} is ", which is $r[10 : 11]$. The transition is an *inclusion transition* if $i' \geq i \wedge j' \leq j$, or a *connection transition* if $i' \geq j$. There is no case that two subexpressions partially overlap. There could be many states reached from v by inclusion transitions, but at most one state reached from v by a connection transition. The current state v must be matched for at least c_v^{min} times, before the engine can take a connection transition to match next subexpressions. It can, however, take an inclusion transition to help match v as long as $c_v < c_v^{max}$. For instance, branch state v_4 has $c_{v_4}^{min} = c_{v_4}^{max} = 1$, which means it needs to and can be matched only once. When $c_{v_4} = 0$, it can be only matched by transiting to v_5 or v_8 via inclusion transitions; when $c_{v_4} = 1$, it has to transit to v_6 by a connection transition. Similarly, the loop state v_2 has $c_{v_2}^{min} = 0$ and $c_{v_2}^{max} = +\infty$. It can transit to v_3 by an inclusion transition or v_{10} by a connection transition.

B. Research Goals and Challenges

We aim to investigate the ReDoS problem in extended regular expressions. More specifically, we study how to precisely and efficiently detect extended regular expressions that are vulnerable to ReDoS attacks. Further, we generate auxiliary information to help mitigate the vulnerabilities, *e.g.*, by highlighting the vulnerable subexpressions. We do not claim to detect all vulnerable regexes, *i.e.*, our method is not sound. Rather, we aim to develop a complete method that reports only true positives.

We face the following challenges in detecting vulnerable regexes with extended features.

1) **Definition of Vulnerable Patterns in e-NFA.** To precisely identify vulnerable regexes, we need a clear specification of the vulnerable e-NFA patterns. Although the theory of ReDoS vulnerabilities in NFA has been well established, there exists no formal definition of vulnerable patterns in e-NFA. Unlike NFA, e-NFA uses a context-sensitive grammar. It is hard to define a vulnerable pattern that fits in all contexts.

2) **Extended Feature Support.** Regular expressions have been extended with rich features to facilitate powerful string matching. However, the extensive use of those features also makes it hard to detect vulnerabilities through static analysis. For example, "backreferences" allow the same text to be matched more than once. Without executing the matching algorithm, it is infeasible to know the exact text in backreference. Further, it is difficult for static analysis methods to support all extended features, because they usually depend on a dedicated parser. Therefore, static structural analysis is imprecise and could miss many vulnerabilities.

3) **Attack String Generation.** In order to identify true positives from all vulnerable patterns, we need to construct attack strings to trigger the timeouts. Precisely and efficiently

[2]The "Dot" pattern matches any single character.

[3]It matches any single character except ".

[4]They include traditional groups "()", lookarounds, named groups, and atomic groups.

generating the attack strings, however, is non-trivial. On the one hand, pure static analysis cannot analyze the meaning of some extended features and would simply use the literal values in regexes. For instance, they would use "\s" to represent a 'blank' character, which is apparently incorrect. On the other hand, dynamic analysis methods, *e.g.*, fuzzers, are usually inefficient, because they need to search over a huge number of possible strings.

IV. Modeling ReDoS Vulnerabilities

In this section, we aim to model ReDoS vulnerabilities by proposing vulnerable e-NFA patterns and corresponding attack string patterns. We analyze the characteristics of a critical substring (*attack core*) in the attack string that would cause catastrophic backtracking when a match attempt fails (§IV-A). To define vulnerable e-NFA patterns, we then discuss the crucial states in an e-NFA that may lead to super-linear matching behavior (§IV-B). We propose that there are only two types of such states, and each type can be represented by a classical feature. Therefore, each vulnerable e-NFA pattern can be abstracted as a structure composed of these two types of classical features. Based on the above observations, we propose different types of vulnerable e-NFA patterns (§IV-C) and attack string patterns (§IV-D).

A. Attack Core Detection

The attack core is the most crucial part in an attack string. It has at least two distinct sub-matching paths in the e-NFA, and the transitions on these paths can be repetitively taken for matching it. In other words, it is the *common match string* of multiple subexpressions and their repetitions. Therefore, when matching an attack core, the matching algorithm has multiple traceable options. The repetition of attack core makes the overall matching complexity super-linear when the engine backtracks. We propose the following definition of *common match string* for subexpressions to help find an attack core.

Definition 2 (Common match string)**.** *Several distinct subexpressions $R = \{r_0, r_1, \dots\}$ have a common match string s **if** there exists a string s such that s can match (the repetition of) each subexpression $r \in R$.*

For example, "ab" is a common match string of the two subexpressions r_0 ="(a|b)" and r_1 ="ab". It matches two repetitions of r_0, and (one repetition of) r_1.

B. Crucial States in e-NFA

In this section we discuss which states in e-NFA would lead to super-linear matching behavior. [26] proposes two critical factors as necessary conditions for repeated backtracking in a regular expression:

1) the regular expression applies repetition to a complex subexpression;
2) for the repeated subexpression, there exists a match, which is also a suffix of another valid match.

Inspired by [26], we consider all states in an e-NFA that possibly meet the above conditions, and categorize them into the following two categories.

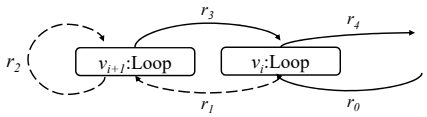


Fig. 2: Loop in Loop vulnerable structure. A dashed curve arrow denotes a matching path starting with an inclusion transition, and a solid curve arrow denotes a matching path starting with a connection transition.

Definition 3 (Loop state)**.** *A state v is a loop state if its maximum allowed match count $c_v^{max} > 1$.*

Definition 4 (Branch state)**.** *A state v is a branch state if v has more than one outgoing inclusion transition.*

We then introduce a theorem that only these two types of states could lead to the super-linear matching behavior in any e-NFA. The proof of the theorem is provided in Appendix §A2.

Theorem 1 (States that construct the vulnerable structure)**.** *If an e-NFA has neither loop states nor branch states, then the e-NFA match process runs in linear time.*

Theorem 1 does not suggest that loop states and branch states can lead to super linear matching time. We will show that in §IV-C. Knowing that the vulnerable structure could consist of only loop states and branch states, we analyze how regex features fall into these two types:

- Loop state: "classical quantifiers $* +$", "Greedy quantifiers $\{m,n\}\{n\}\{m,\}$"[5] and "Lazy quantifiers $??\ *?\ +?\ \{\}?$"[6].
- Branch state: "classical branch $|$" and "Lazy $?$".

"Lazy ?" can be considered as a special case of classical feature "Branch |". Greedy quantifiers and lazy quantifiers are also similar to the ordinary quantifiers '*' and '+' in the context of ReDoS, because the attack string is crafted to match toward their maximum repetition limits. Therefore, we can consider all loop states as classical quantifiers and all branch states as classical branches, and refer to the complexity theory of NFA to propose ours.

C. Vulnerable e-NFA Patterns

We define each vulnerable e-NFA pattern as a structure composed of loop states and/or branch states. The "Loop in Loop", "Branch in Loop", and "Loop after Loop" vulnerable e-NFA structures are shown in Figure 2, Figure 3, and Figure 4, respectively. In these pattern representations, we only remain the crucial states and simplify the others into subexpressions (*e.g.*, Figure 2 represents regexes with the format $r_0(r_1(r_2)^*r_3)^*r_4$, in which * can be replaced by other quantifiers). There is a special case of "Loop after Loop" vulnerable structure, that is "Loop in Branch" structure, which has the same polynomial complexity. We do not provide it here due to page limit.

We provide the proofs of the theorems in Appendix §A3.

Theorem 2 (Loop-in-Loop vulnerable e-NFA pattern)**.** *An e-NFA pattern has exponential worst-case complexity if there exist two loop states v_i and v_{i+1} that v_{i+1} can be reached*

[5]They instruct the engine to match as many instances of the quantified subpattern as possible, thus are called greedy quantifiers.

[6]They instruct the engine to match as few instances of the quantified subpattern as needed, thus are called lazy quantifiers.

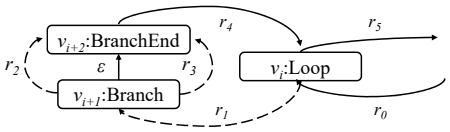


Fig. 3: Branch in Loop vulnerable structure.

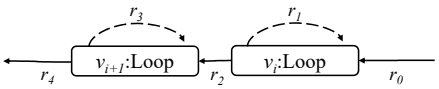


Fig. 4: Loop after Loop vulnerable structure.

via a matching path starting with an inclusion transition from v_i, such that the two subexpressions $r_1r_2r_3$ and r_1r_3 have a common match string.

Theorem 3 (Branch-in-Loop vulnerable e-NFA pattern)**.** *An e-NFA pattern has exponential worst-case complexity if there exist a loop state v_i and a branch state v_{i+1} that v_{i+1} can be reached via a matching path starting with an inclusion transition from v_i, such that the two subexpressions $r_1r_2r_4$ and $r_1r_3r_4$ have a common match string.*

Theorem 4 (Loop-after-Loop vulnerable e-NFA pattern)**.** *An e-NFA pattern has polynomial worst-case complexity if there exist two loop states v_i and v_{i+1} that neither can be reached via a matching path starting with an inclusion transition from the other, such that **(i)** there exists a transition from v_i to v_{i+1} either indirectly through a subexpression r_2 or directly (where $r_2 = \epsilon$), and **(ii)** if $r_2 = \epsilon$ the two subexpressions r_1 and r_3 have a common match string, otherwise three subexpressions r_1, r_2 and r_3 have a common match string.*

D. Vulnerable Attack String Patterns

The attack string pattern of vulnerable e-NFA structures can be represented by $s_0.s^k.s_1$, where s_0, s_1 and s are the *prefix*, *suffix*, and *attack core*, respectively. We can also locate the vulnerable structure from two special states: 1) prefix tail, the last state on the prefix matching path; and 2) suffix head, the first state on the suffix matching path.

The attack string patterns are constructed as follows: 1) for all three vulnerable e-NFA patterns, s_0 is a match string of r_0, and s_1 makes $s_0.s^k.s_1$ fail to match the entire regex; 2) for Loop in Loop patterns, s is a common match string of subexpressions r_1r_3 and $r_1r_2r_3$, v_i is both the prefix tail and the suffix head; 3) for Branch in Loop patterns, s is a common match string of subexpressions $r_1r_2r_4$ and $r_1r_3r_4$, v_i is both the prefix tail and the suffix head; and 4) for Loop after Loop patterns, s is a common match string of subexpressions r_1, r_2 and r_3, v_i is the prefix tail and v_{i+1} the suffix head.

V. REVEALER

In this section, we present REVEALER, a hybrid system based on the theory in the last section to detect and exploit ReDoS vulnerabilities. The workflow of REVEALER is shown in Figure 5. REVEALER first locates vulnerable e-NFA patterns with a simplified e-NFA structure called E-TREE in static analysis (§V-B). It then finds a common match string (*i.e.*, the attack core) in its dynamic analysis (§V-C), and generates the attack prefix and the attack suffix to form an entire attack string (§V-D). It finally validates whether the attack string can trigger super-linear matching behavior (§V-E). We next discuss an overview and the novelty of REVEALER in §V-A. We will also discuss some of its limitations in §V-F.

A. Overview

As we had introduced in §II, both existing static approaches and dynamic approaches have their limitations in detecting ReDoS vulnerabilities in extended regexes. On the one hand, the existing formalization of static approaches, regardless of the design details, belongs to context-free grammar, which prevents them from supporting all extended features that can be described only by context-sensitive grammars. The difficulty of adopting a context-sensitive grammar lies in not only developing corresponding theories for formally modeling the problem (which we did in §III-A and §IV-C), but also solving the problem of attack string generation that is NP-hard[7]. On the other hand, dynamic approaches can be very inefficient in finding the attack strings especially for complex patterns, as most fuzzers use only basic genetic methods for generating inputs, which are unlikely to trigger the vulnerabilities.

We overcome such limitations by proposing a hybrid approach. First, we design a static analysis to identify the vulnerable patterns in an e-NFA representation for supporting the context-sensitive grammar. Second, we reduce the problem of attack string generation to one with a polynomial-time solution by introducing extra constraints—the maximum string length and the minimum matching step count—on the generated attack string. To meet the constraints, we design a dynamic analysis as the constraint solver to generate the attack core by simulating the existing matching mechanisms of extended regexes. By leveraging the regex structures, our dynamic analysis can directly generate the right attack cores for exploitation in a more intelligent and efficient manner.

B. Static Analysis

Our static analysis consists of two parts. First, we introduce E-TREE: our simplified data structure of e-NFA; Next, we traverse the E-TREE to find a set of vulnerable patterns P, in which each pattern p is represented by two e-NFA states $\langle v, w \rangle$.

1) E-TREE*:* The Java 8 regex engine parses a regex into an e-NFA A. E-TREE is a simplified representation of A. It reduces the complexity of searching certain states from in a graph (the Java e-NFA Figure 1a) to in a tree (the E-TREE Figure 1b). Searching the loop/branch states for finding vulnerable patterns on E-TREE is simpler than on the original Java e-NFA data structure.

To build an E-TREE, we first remove the state 'Exit' and its corresponding transitions from A, because it does not represent any regex feature. We keep all the other states in E-TREE. Next, we determine the transitions in E-TREE.

Existing e-NFA implementation does not differentiate the inclusion transitions from the connection transitions, but includes the logic of selecting a transition inside the transition function

[7]It is equivalent to regex matching, whose difficulty was proved in [16].

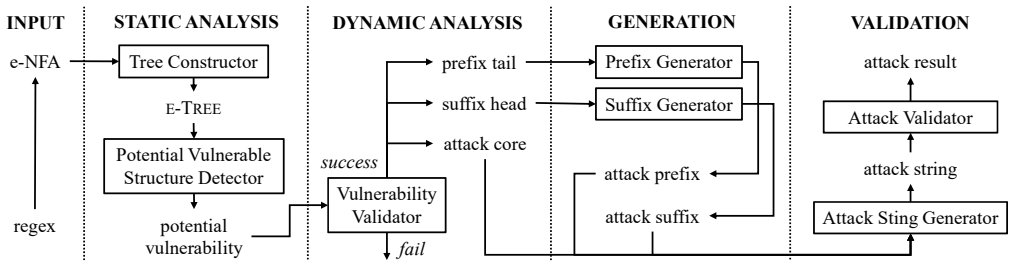


Fig. 5: An architecture overview of REVEALER.

δ'_v. In Figure 1a, all transitions are treated equally, making it hard to determine a traversal order. We extract the transition types statically in E-TREE. A solid arrow and a dashed arrow represent a connection transition and an inclusion transition in Figure 1b, respectively. Most states, except for 'BranchEnd' and states after a 'GroupTail' and before a 'GroupHead', have only one incoming transition. For the two types of states, we keep only one incoming transition in E-TREE. 'BranchEnd' has an incoming connection transition from the 'Branch' state, and one from each branch. We remain the one from the 'Branch' state and omit the others (*e.g.*, $v_5 \rightarrow v_6, v_8 \rightarrow v_6$). The second-type states have an incoming connection transition from 'GroupTail' and an outgoing inclusion transition to 'GroupHead'. We remove such transitions from 'GroupTail' (*e.g.*, $v_7 \rightarrow v_2$).

2) Vulnerable Structure Detection: We define the E-TREE traversal algorithm $Traverse$ for finding states related to vulnerable patterns. It is basically a depth-first search. From a state v, $Traverse$ first takes the inclusion transitions and then the connection transitions to visit other states. For example, $Traverse(v_0)$ visits all states in order $v_0, v_1, v_2, v_3, v_4, v_5, v_8, v_6, v_7, v_{10}, v_{11}$.

Since all vulnerable patterns start from a loop state, we collect all loop states into a list L from $Traverse(v_0)$. We then traverse from each state in L and find other loop/branch states to get a set P of pairs $\langle v, w\rangle$ where v is a loop state and w is a loop or branch state. A pair is a "Loop in Loop" or "Branch in Loop" pattern, if w can be (indirectly) reached by taking a direct inclusion transition from v (*i.e.*, w represents a subpattern of that represented by v). Similarly, a pair is a "Loop after Loop" pattern, if w can be (indirectly) reached through a direct connection transition from v.

C. Dynamic Analysis

The static analysis finds a set of potential vulnerable patterns $P = \{\langle v^0, w^0\rangle, \langle v^1, w^1\rangle, \dots\}$. The dynamic analysis functions as a constraint solver, *i.e.*, for each vulnerable pattern $p \in P$, it verifies whether the corresponding subexpressions r_0, r_1, r_2 (defined in §IV-C) have a common match string s. For example, in Figure 1a, there is only one "Branch in Loop" pattern $\langle v_2, v_4\rangle$. Its corresponding subexpressions are: $r_0 =$ `.`, $r_1 =$ `[^"]`, $r_2 = \epsilon$.

In §V-C1, we propose the algorithm $SingMatch$ that generates a match string s for a matching path τ. The algorithm is based on the existing matching mechanisms in the e-NFA A. In §V-C2, we present $CommMatch$, which generates a common match string s of several matching paths $\{\tau_0, \tau_1, \dots\}$. In §V-C3, we demonstrate how dynamic analysis is performed on top of the algorithm $CommMatch$.

1) The Single Match Algorithm: The single match algorithm $SingMatch$ generates a match string s for a single matching path τ by progressively building s from sub-match string s'.

We defined in §III-A that a state v may have a corresponding acceptable string set AS_v. The match function $\delta_v : (s, p, t) \rightarrow (S_v, p', t')$ would search if there exists p' such that $s[p : p'] \in AS_v$. We change the "match" logic into "generation" by starting with $s = \epsilon, p = 0$, finding a valid match s' from AS_v, extending s with s' (*i.e.*, $s = s.s'$), and then executing the match function δ_v. After these operations, δ_v would match the pre-selected substring s' naturally, and p' would become the length of the new s. Iteratively, s becomes a match string of the matching path τ when v successfully reaches the end state.

Take the matching path $\tau = v_2, v_3, v_4, v_8$ as an example. The algorithm follows the transitions along τ until it needs to generate a match string at state v_8. It randomly selects a symbol as s' from AS_{v_8}, which includes any symbol in the alphabet Σ except `'"'`. For example, it selects `"a"` as s and sets p as 0 in match function δ_{v_8}, and would cause a successful match. Since v_8 is the last state in τ, the algorithm ends with a valid match string `"a"`.

2) The Common Match Algorithm: The $CommMatch$ algorithm generates a common match string s of several matching paths. Here we discuss only cases with three matching paths: τ_0, τ_1, τ_2; other cases work similarly. It performs $SingMatch$ on each matching path simultaneously and syncs on the substring s' generated in each step.

$CommMatch$ holds a current state vector V, which stores the current states for all matching paths. It is initialized by the first states of all the matching paths: *e.g.*, $V_0 = [\tau_0[0], \tau_1[0], \tau_2[0]]$. Let v' represent a possible next state v can transit to if the algorithm can find a substring $s' \in AS_{v'}$. The common s' is therefore determined by the intersection of $AS_{v'}$ for each state v in V. Let $V = [v_0, v_1, v_2]$, the common acceptable strings AS_V would be $AS_{v'_0} \cap AS_{v'_1} \cap AS_{v'_2}$. The algorithm selects one string s' from AS_V for building the common match string s. $CommMatch$ terminates with a failure when $AS_V = \Phi$ and outputs ϵ. It ends successfully when s can match all matching paths for at least one repetition.

In our example, $r_2 = \epsilon$, so we only need to consider r_0 and r_1 and run $CommMatch(\tau_0, \tau_1, \epsilon)$. The only matching paths of r_0 and r_1 are $\tau_0 = [v_2, v_3, v_4, v_5]$ and $\tau_1 = [v_2, v_3, v_4, v_8]$, respectively. We initialize $V_0 \leftarrow [v_2, v_2]$, conduct transitions on τ_0 and τ_1 simultaneously, until $V = [v_4, v_4]$. The next common acceptable strings for both matching paths would be: $AS_V = AS_{v'_5} \cap AS_{v'_8} = \{\alpha \in \Sigma | \alpha \neq "\}$. If the algorithm randomly selects `"a"` from AS_V as the common s', then $s =$ `"a"` would lead to two successful matches. Now that τ_0 and

τ_1 both get matched once, the $CommMatch$ algorithm ends successfully and outputs a common match string "a".

3) Performing Dynamic Analysis: We present the entire dynamic analysis in Algorithm 1. It takes the set of possible vulnerable patterns found in the static analysis as input. For each pattern, it extracts the matching paths of the three corresponding subexpressions, and leverages the $CommMatch$ algorithm to find a common match string. The common match string will be repeated as the attack core for many times for generating the attack string in §V-D.

However, each subexpression could have numerous matching paths, resulting in numerous path combinations for each group of three subexpressions. Our analysis may spend much time on analyzing path combinations (especially those including long matching paths) that might not lead to DoS. Further, we need a shorter common match string s because more repetitions of s lead to (exponentially) more backtrackings. To limit the search space and find more powerful attack strings, we set a maximum length l'_m of the common match string.

We derive l'_m from two thresholds—maximum attack string length l_m and minimum matching step count γ—for vulnerability validation. We set l_m as 128 and γ as 10^5 as we will explain in §VI-A. Let l denote the length of the attack string ($l \leq l_m$) , and l' denote the length of the common match string s ($l' \leq l'_m$). Let n denote the number of repetitions of s in the attack string, we have $n \leq n_m = \lfloor \frac{l_m}{l'} \rfloor$. The maximum condition happens when both attack prefix and suffix are ϵ. We set $a^n, a \in \mathbb{N}^*$ as the maximum matching complexity a vulnerable pattern can trigger, where a is the number of choices in each backtracking step. One feature (state) can match multiple characters, and in general at most three states (feature start, match, feature ends) are used to match a character. Therefore, we denote $k \cdot l'$ as the maximum length of a matching path of s, where $k \leq 3$. When matching the attack string, the engine backtracks at most a^n times, and each backtracking takes at most $k \cdot l'$ steps, thus the maximum matching step count is $k \cdot l' \cdot a^{n_m}$, which shall be no less than γ to pass the runtime validation. Therefore, we have $k \cdot l' \cdot a^{n_m} \geq \gamma$, and thus $l' \cdot a^{\lfloor \frac{l_m}{l'} \rfloor} \geq \frac{\gamma}{k}$. We get $l' \leq 9$ under the settings of $a = 2, k = 3$, and set l'_m as 9. We use the setting because other cases (*e.g.*, a regex has a complexity over 2^n, or each character needs more than three states to match) are rare. Besides, even if a rare case occurs, only under specific conditions, it will become a false negative (FN). For example, if there exists a regex with 3^n complexity, the result becomes $l' \leq 16$. It would be a FN only if the common match string happens to have a length $9 < l' \leq 16$. In practice, we tried l'_m from 9 to 14, and REVEALER reported the same number of true positives. So we finalize our setting with $l'_m = 9$.

D. Generation

With the attack core s generated in dynamic analysis, the generation phase produces attack prefix s_0 and attack suffix s_1 of a vulnerable pattern, to generate the final attack string. We get prefix tail and suffix head from states v and w in a vulnerable pattern $\langle v, w \rangle$ according to our definition in §IV-D.

1) Prefix: We use the $SingMatch$ algorithm to generate a match string for the path τ starting from v_0 to the prefix tail. To make the prefix as short as possible, we instruct the algorithm to take a lazy matching strategy by requiring the matching count c_v of a state $v \in \tau$ to be no more than c_v^{min} (*e.g.*, for a+ we need only one instance of a).

Algorithm 1 Dynamic Analysis: Getting a common match string for each potential vulnerable pattern.

Input: P: a set of possible vulnerable patterns
Output: S: a set of vulnerable patterns with corresponding attack strings

$S \leftarrow \{\}$
for all $p \in P$ **do**
  $r_0, r_1, r_2 \Leftarrow p$ // *Get the subexpressions of a vulnerable pattern* p
  $T_0 \Leftarrow r_0, T_1 \Leftarrow r_1, T_2 \Leftarrow r_2$ // T_i *stores all matching paths with the largest match string length* $l'_m = 9$ *for a subexpression* r_i
  for all $\tau_0 \in T_0, \tau_1 \in T_1, \tau_2 \in T_2$ **do**
    $s \leftarrow CommMatch(\tau_0, \tau_1, \tau_2)$
    if $s \neq \epsilon$ **then**
      $S \leftarrow S \cup \{\langle p, s \rangle\}$
      break
    end if
  end for
end for
return S

2) Suffix: A valid suffix has to cause the regex match to fail on all possible matching paths. We first collect states where the match of attack core s could possibly end in a set ES. We run $CommMatch$ with initial $V = ES$, but by generating rejecting characters $\Pi = \Sigma - \{s[0] \mid s \in AS_{v'} \wedge v' = \delta'_v(S_v, t) \wedge v \in V\}$, where Σ is the alphabet of symbols. Then we select a character from Π as s' to cause next matches of all possible paths to fail. If $\Pi = \Phi$, we let $CommMatch$ match one state in a lazy way, and continue the process until we find a character to make the next matches fail.

3) Attack String: We combine the attack prefix s_0, the attack core s, and the attack suffix s_1 under the format $s_0.s^k.s_1$ to generate the final attack string, where $k = \lfloor \frac{l_m - length(s_0) - length(s_1)}{length(s)} \rfloor$.

E. Validation

In this part, we use the original regex engine together with an extra variable for counting matching steps to verify if the attack string can trigger a ReDoS vulnerability. Inside the original matcher in the Java 8 regex engine, there is a structure called "Trace", which records the log information generated along the matching process. Its log size is the matching step count. We break the matching process and report a successful attack if matching step count exceeds the upper limit γ.

Even the dynamic analysis can find a valid attack core, the final attack string might not trigger a ReDoS. For example, the vulnerable pattern `.*?tn=.*id=.*` is for sure of polynomial complexity, but its attack core `tn=id=` is relatively long. The attack string generated cannot pass the validation in our evaluation (§VI).

F. Limitations

The validation phase of REVEALER ensures it reports no false positive. However, the choices of the thresholds in the validation phase can have impact on the results, *i.e.*, a case may be considerred as either a true positive or a negative under different attack string lengths and matching step count thresholds. We demonstrate that in detail in §VI-A.

Further, REVEALER may have false negatives for the following reasons. First, our definition of vulnerable patterns

in §IV-C may be incomplete. We tried our best to consider all possible structural patterns. But the completeness of such definition would need another work to prove. Second, although e-NFA can describe all extended features, our prototype implementation does not fully support them. It extracts the e-NFA from the Java 8 regex engine, which cannot parse regexes with *conditionals*. Further, it removes *backreferences* when constructing E-TREE. A detailed discussion about extended features is in §VI-C1. Third, our implementation of the end state set ES may be incomplete. We only use state v, its next state v' through an inclusion transition, and its next state v'' through a connection transition, *i.e.*, $ES = \{v, v', v''\}$. Fourth, we do not find attack cores that are longer than l'_m in our dynamic analysis for targeting the most powerful attack strings and for efficiency concerns. There might exist false negatives of which the attack cores are longer than l'_m. Nevertheless, such attack cores result in fewer rounds of backtracking.

VI. EVALUATION

In this section, we evaluate the effectiveness of REVEALER. We first demonstrate that REVEALER is able to effectively generate attack strings that help trigger and detect (unknown) ReDoS vulnerabilities (§VI-B). We then characterize the detected vulnerabilities (§VI-C), and validate the attacks with regex engines of other languages (§VI-D). Finally, we apply it to detect unknown ReDoS vulnerabilities in popular real-world applications (§VI-E). We describe the experiment setup next.

A. Setup

To evaluate whether REVEALER is able to effectively detect ReDoS vulnerabilities, we use the dataset—a collection of 29,088 regexes from three different sources—used in [30]. We compare REVEALER with the following three state-of-the-art ReDoS vulnerability detection tools on the same dataset: 1) ReScue [30], a genetic fuzzing tool for detecting ReDoS vulnerabilities; 2) RXXR2 [29], an improved version of the static analysis tool RXXR [18] based on transition production; and 3) Rexploiter [38], a static analysis tool based on vulnerable structure identification. These tools were among the best performing tools used in two recent works about ReDoS vulnerability detection [11, 30]. Since the authors of [30] did not disclose publicly their detected vulnerabilities, we cannot directly compare with their results. We preprocess the regexes for RXXR2 and Rexploiter according to their requirements. We apply each tool for generating the corresponding prefix s_0, attack core s_1 and suffix s_2 to construct an attack string s in the form $s_0.s_1^k.s_2$ for validating a vulnerability. The experiments are performed on a 20-core Intel Xeon server with 240 GB RAM running Ubuntu 16.04.

We limit the length l of the generated attack string to be less than 128[8] by following the practice in [30]. Under such a limited input length, it is hard to differentiate super-linear complexity regexes from linear complexity ones by wall-clock time. A super-linear complexity criteria was proposed in [11] that a 10-second timeout shall be triggered with at most 85,615 pumps (100K-1M characters)[9]. We used REVEALER with this large length limit to generate attack strings and found 2,172 triggered the timeout. However, by reducing the limit to 128, these vulnerable regexes can be matched in as low as only 0.159 second, which is even lower than the matching time of many linear regexes. Therefore, we use matching step count (also used in [30]) instead of wall-clock time as the metric for validating an attack.

[8]A larger limit lets ReScue run for longer time without improving its results.
[9]A pump represents one repetition of the attack core in the attack string.

TABLE I: Matching steps of super-linear regexes.

Range	$(1e3, 1e4]$	$(1e4, 1e5]$	$(1e5, 1e6]$	$(1e6, 1e7]$	$(1e7, 1e8]$	$(1e8, +\infty)$
Count	393	1,332	94	92	25	236

TABLE II: The overall evaluation results.

Tool	# of Vul.	# of FP	Error Rate (%)	Avg. Time (s)
REVEALER	450	0	0.00	0.0076
ReScue	187	0	0.00	18.2259
RXXR2	112	103	47.91	0.0042
Rexploiter	63	1,959	96.88	0.4472

We conclude that a reported vulnerability is a *true positive* if the matching step count is greater than the threshold γ. To determine γ, we count the matching steps of those 2,172 verified super-linear cases with a 128-character attack string length limit and show in Table I. To include all severe vulnerabilities that could cause a 10-minute timeout with 100 pumps, we choose 10^5 as γ because there is one in the $(1e5, 1e6]$ group as discussed in §VI-B. However, this prevents REVEALER from reporting more than 1,700 regexes as vulnerable under an attack string not longer than 128 characters. Indeed, these 1,700+ cases found by REVEALER can be exploited to cause a DoS if an attacker uses a very long attack string according to the criteria in [11].

We also measure the time each tool spends on analyzing one regex. We found that ReScue can spend up to 12.49 hours on analyzing one single regex without limitation in one round, but its reported vulnerable regexes were all found within 250 seconds. Therefore, we set a time limit of 250 seconds per regex for ReScue in our experiment. In three runs, it initially found 174 true positives, which were fewer than reported in [30]. We found using a larger time limit did not help much and was time-consuming for analyzing all 29K regexes. We instead focused on the additional ones found by other tools with the 10-minute limit (as used in [30]), and finally detected 187 vulnerabilities in 20 rounds, which were even 1 more than reported in [30]. We used the same 10^8 matching step threshold as in [30] for ReScue because we found a smaller one would result in a worse performance, which we will explain in §VI-B.

B. Results

The overall evaluation results are shown in Table II. In total, the four tools detected 450 true positive regexes that are vulnerable under ReDoS attacks. The numbers of vulnerabilities reported by the three tools we compared are close to the ones reported in [30]. Therefore, we believe our evaluation results are valid. Note that the validation threshold we used is smaller than that (10^8) in [30] as we allow ALL tools to report more super-linear (and sub-exponential) time vulnerabilities. We draw a Venn diagram based on the numbers of vulnerabilities detected by the four tools in Figure 6.

REVEALER significantly outperformed all three state-of-the-art tools. It could detect *all* 237 vulnerabilities that were found

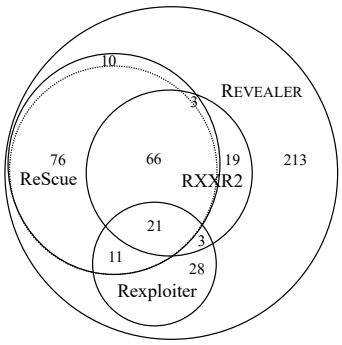


Fig. 6: Venn diagram of vulnerabilities.

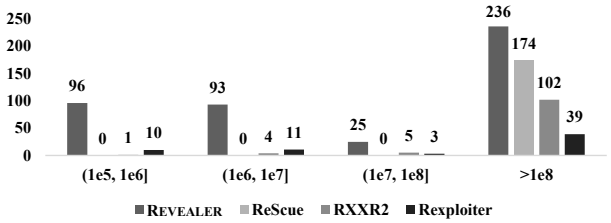


Fig. 7: The matching steps of regexes reported by each tool.

by them. It further detected 213 *previously unknown* vulnerable regexes, which is a 89.87% improvement. ReScue performed the second, by finding 187 (41.56%) vulnerabilities, which were 1 more than it did in [30]. However, it took 20 additional rounds to find 13 vulnerabilities, indicating that it has a high requirement on computing resource. RXXR2 and Rexploiter were less effective, reporting 112 and 63 vulnerabilities, respectively.

Figure 7 presents the matching step distribution of vulnerabilities reported by each tool. Different groups divided by matching steps approximately indicate different complexities, although each group contains both exponential and polynomial cases. However, some polynomial cases could also cause noticeable ReDoS in practice. To find the severe cases, we increase the input length limit by using 100 pumps and set a 10-minute timeout. We confirmed 1, 4, 4 and 229 such severe cases in the four groups, respectively. Nine cases were not in the `>1e8` group because their attack cores or attack prefixes were long, resulting in fewer backtrackings when the input lengths were short (≤ 128). Five cases in the `>1e8` group took at least 2.963-second but less than 10-minute matching time using 100 pumps. They were complex "Loop-after-Loop" (polynomial) cases with short attack cores. They were separated from those severe cases when the input lengths became large. The rest two in the `>1e8` group took less than 1 second under long inputs, but more than 10 minutes with 128-character long inputs. We will discuss them in §VI-D.

Considering only the 238 (52.89%) severe ones, REVEALER still greatly outperformed the other tools. ReScue detected only 187 regexes in the `>1e8` group (183 were severe, 2 were not, 2 were the special cases mentioned above) and did not detect any in other groups because it used $1e8$ at its internal matching step threshold. Actually, it could not detect some of the 187 vulnerabilities if an internal threshold smaller than $1e8$ was used, because it would stop the search when the matching steps of a generated input reached the threshold.

TABLE III: Extended features supported by each tool. ✓ means the feature is supported; × indicates the feature is not supported.

Extended Features	REVEALER	ReScue	RXXR2	Rexploiter
Unicode chars	✓	✓	×	✓
Set operations	✓	×	×	×
Lookarounds	✓	✓	×	×
Backreferences	×	✓	✓	×
Non-capturing groups	✓	✓	✓	×
Named groups	✓	✓	×	✓
Atomic groups	✓	✓	×	×
Conditionals	×	×	×	×
Greedy quantifiers	✓	✓	✓	✓
Lazy quantifiers	✓	✓	✓	✓
Possessive quantifiers	✓	✓	×	×

As expected, both RXXR2 and Rexploiter reported high error rates, which are the ratio of false positivies to reported positives. This demonstrates that a dynamic approach like ReScue or a hybrid approach like REVEALER would be better suited for detecting and exploiting vulnerable regexes, especially those using extended features.

While being *effective*, REVEALER can also *efficiently* detect ReDoS vulnerabilities. On average, analyzing one regex took it only 0.0076 second, which is close to the 0.0042 second of the fastest tool—RXXR2. Rexploiter took two orders of magnitude longer time than RXXR2 on average. ReScue was the slowest, and was 2,397 times slower than REVEALER—it spent 18.2259 seconds on analyzing one regex on average. We were unable to reproduce the 0.6128 second as the authors reported in [30], even we tried many times with different settings.

C. Characterization of Detected Vulnerabilities

To understand why REVEALER can significantly outperform the state-of-the-art tools, we characterize the detected vulnerabilities by extended features (§VI-C1), vulnerable structures (§VI-C2), and the generated prefixes and suffixes (§VI-C3).

TABLE IV: Breakdown of vulnerable regexes by extended features.

Features	REVEALER	ReScue	RXXR2	Rexploiter
Classical features only	40	10	10	4
Unicode chars	4	3	0	1
Set operations	339	154	89	52
Lookarounds	30	16	0	0
Backreferences	10	7	1	0
Non-capturing groups	95	55	30	7
Named groups	34	10	0	4
Atomic groups	0	0	0	0
Conditionals	0	0	0	0
Greedy quantifiers	156	84	44	26
Lazy quantifiers	107	32	22	10
Possessive quantifiers	10	6	1	0

1) Extended Features: We list the extended features supported by each tool in Table III. As explained in §V-F, our prototype implementation does not support all extended features. Nevertheless, both REVEALER and ReScue support more extended features than RXXR2 and Rexploiter, and consequently detected more vulnerable regexes. In Table IV, we list the categorization of vulnerabilities by extended features. As shown, only 40 vulnerable regexes do not use any extended feature, which indicates the need to support them.

Supporting a feature might help a tool detect a vulnerable regex using such a feature. For instance, Rexploiter supports

TABLE V: Detected vulnerabilities classified by structure.

Type	REVEALER	ReScue	RXXR2	Rexploiter
Loop in Loop	185	142 (76.76%)	87 (47.03%)	38 (20.54%)
Branch in Loop	50	38 (76%)	25 (50%)	3 (6%)
Loop after Loop	215	7 (3.26%)	0 (0%)	22 (10.23%)

'Unicode chars' and 'Named groups' while RXXR2 does not, so it detected 1 and 4 vulnerabilities under these two types while RXXR2 detected none. Similarly, RXXR2 supports 'Backreferences' and 'Non-capturing groups' while Rexploiter does not, RXXR2 thus detected 1 and 23 more vulnerabilities with the two features than Rexploiter, respectively.

A tool could detect vulnerable regexes using a feature it does not support. All tools detected many vulnerable regexes with the feature "Set operation" even only REVEALER supported it, because they substituted the set expression with simpler but incomprehensive character literals and were able to find some valid match strings. Similarly, REVEALER detected 10 cases with 'Backreferences' although the prototype did not support it, because they met one of the following three conditions. **C1**: backreference was not on the matching path of the attack string, so REVEALER did not transit to a state of it. **C2**: backreference could be ignored according to the semantics of the regex, *e.g.*, both `(\1)*` and `(\1)?` can be matched zero time in `(a)b(\1)*c(\1)?(d*)*`. The generated attack strings were still valid even by removing it in E-TREE. **C3**: backreference matched the attack core. The subexpression state it referred to was probably on the matching path, *e.g.*, `\1` in `a*([a-z])a*(\1)a*` can be matched by the attack core "`a`". Thus even it consumed extra attack core(s), the attack was still valid.

The tools actually have different levels of feature support. This is also one of the reasons why the tools have different performances even though they all support the same feature. ReScue supports most extended features because it drives the Java regex engine as a gray box. But it does not consider the real semantics of a feature as REVEALER does. For instance, ReScue supports 'Named groups'. But without understanding the semantics, it adds the name (*e.g.*, "`a`" is the group name in `(?<a>x)*`) as an input seed and could generate many incorrect inputs. Besides, RXXR2's 'Backreferences' support is incomplete (and worse than ours), because it ignores such features under **C1** and terminates the analysis under other conditions. Overall, it cannot support the full semantics of this feature because it considers only context-free grammars.

2) Vulnerable Structures: Covering all three types of vulnerable e-NFA patterns also allows REVEALER to detect more vulnerabilities. We classify the detected vulnerabilities and present the breakdown in Table V.

Other than REVEALER, Rexploiter is the only tool considering the 'Loop after Loop' vulnerable structure, and therefore performed the best among the three tools in detecting vulnerabilities in this class. But its performance was strictly limited by its ability of generating attack strings, which we will discuss in §VI-C3.

ReScue detected over 76% of 'Loop in Loop' and 'Branch in Loop' vulnerabilities, but found about only 3% of 'Loop after Loop' cases. This is because the 'Loop after Loop' vulnerabilities are more difficult to trigger. The attack string must match two loops as well as the path in between. But ReScue does not consider multiple paths at the same time.

3) Prefixes and Suffixes: REVEALER can well generate both the prefix and suffix of an attack string. This also helps it find much more vulnerabilities. Without generating a valid prefix, the attack string cannot drive the regex engine to the repeated states. A valid suffix is also needed to force the match to fail at the end such that the regex engine has to backtrack.

Prefix generation is a difficult task for dynamic analysis approaches, *e.g.*, fuzzers, as they need to produce valid sequences of symbols to reach certain inner or deep states. Only a very limited number of sequences are valid whereas the fuzzers might (blindly) search over a huge number of possibilities. Genetic algorithms can help, but not much because genetic algorithms generate offspring by crossing over or mutating the parent string, which would not directly lead to deeper states. Even though ReScue tried to alleviate this problem by improving its node coverage rate, the improvement is still limited by the huge search space and the time/resource constraint. One typical example is the regex (<[^>]*?tag[^>]*?(?:identify_-by)[^>]*>)((?:.*?(?:<[\r\t]*tag[^>]*>?.*?(?:<.*?/.*?tag.*?>)?)*)*), where the shortest prefix of an attack string would be a simple string—"`<tagidentify_by>`". Nevertheless, ReScue went into a meaningless search on other possible inputs and therefore triggered a time-out on each run. The static analysis approaches, on the contrary, will not get stuck in exploring all the possibilities, because they directly find the attack core and pump the shortest prefix. Therefore, both RXXR2 and Rexploiter were successful in this case.

Suffix generation, however, is a task more suitable for dynamic analysis. A static method can only generate a string from the designated transition path, and cannot ensure that the string will not be matched by other paths. For example, given the regex `^(\S+\.{1})*([^\.]\s+)?$`, whose attack core is '`.`', RXXR2 and Rexploiter would try to generate a string that fails to match the suffix regex `([^\.]\s+)?` but can actually match `(\S+\.{1})*`. Hence the entire attack string is accepted. REVEALER considered both the attack core and the suffix regex, and generated the correct suffix '`\t`'. ReScue achieved this through multiple searches.

By taking a hybrid approach, REVEALER inherits the strengths from both static and dynamic approaches. It can statically generate a valid prefix by analyzing the structures, and dynamically find a valid suffix by testing against multiple subexpressions. Therefore, it generated much more valid attack strings that can trigger ReDoS vulnerabilities.

D. Validation with Other Regex Engines

We detected 450 vulnerable regexes with a 10^5 matching step threshold and a 128 attack string length limit. In practice, attackers can choose longer attack strings. To understand their practical impact, we relax the input limit to up to 65,536 pumps, and measure the wall-clock matching time in the Java 8 regex engine on which we built REVEALER. We count the regexes that take a matching time longer than 10 seconds, which is the criteria used in [11]. We also cross-validate them on regex

TABLE VI: Number of timeout regexes with different pumps.

# of Pumps	128	1,024	8,192	65,536
Java 8	234	381	443	443
JavaScript	196	239	368	391
Python	198	242	373	398
Ruby	187	227	357	386
PHP	0	0	3	10

engines of other languages—JavaScript, Python, Ruby and PHP. The results are listed in Table VI.

381 (84.67%) and 443 (98.44%) regexes caused a timeout on Java 8 regex engine under 1,024 and 8,192 pumps, respectively. Seven regexes took less than 10 seconds to match even with 65,536 pumps. Four of them (including the 2 special cases mentioned in §VI-B) start with `(?=^.{1,254}$)`, which limits the maximum acceptable input length to 254. However, their matching time could be more than *10 minutes* when the number of pumps was only 128 (and even when the attack string length was only 128). The other three have strict limit on the repetition of attack cores. For example, `(\\dx\\.]{0,2}(?!\\n)){6,12}` requires the pattern to be matched for no more than 12 times, so longer inputs would not increase the matching time. Their matching time was as high as 0.578, 0.622 and 6.711 seconds, respectively, which were still quite large for real-time applications.

Most vulnerabilities REVEALER detected could also lead to ReDoS on other engines. But up to 64 regexes did not cause timeout on them even with a 65,536 pump limit for JavaScript, Python and Ruby. This might be because these engines do not support some features (*e.g.*, only the Java 8 engine supports possessive quantifiers among the four), or because the engines work differently for some features (*e.g.*, the '-' character in the set operation `[\d-z]` is treated as a literal in Java, but as a range sign in JavaScript and Python). As we mentioned in §II-B, PHP limits the number of backtracking searches to prevent ReDoS. With 65,536 pumps, 286 (63.56%) of the 450 vulnerable regexes triggered PHP's internal backtracking limit; 116 (25.78%) reached the stack limit. However, some vulnerable regexes can still cause a 10-second timeout in PHP, *i.e.*, 3 caused timeouts with 8,192 pumps, and 10 caused timeouts with 65,536 pumps. These regexes are "Loop-after-Loop" cases, whose backtracking behavior cannot be detected under PHP regex engine's current limit. Our results demonstrate that REVEALER can also find ReDoS vulnerabilities in PHP programs.

E. Detecting Real-World Vulnerabilities

We have demonstrated that REVEALER is able to effectively detect both known and unknown ReDoS vulnerabilities in a benchmark dataset. In this section, we explore whether it can also detect unknown vulnerabilities in real-world applications.

We extracted regexes from popular open-source projects for vulnerability detection. Specifically, we searched on GitHub for popular (with more than 600 stars) Python and JavaScript projects that contain the keywords "editor", "web app", or "database" in their description. These projects are likely to use regexes. For Python, we downloaded 28 database, 13 editor, and 7 web app projects. For JavaScript, we downloaded 31 database, 65 editor, and 34 web app projects. We applied REVEALER to regexes extracted from them with default settings.

```
var styfn = {};
styfn.applyFromString = function( string ){
// vulnerable to "/*\t\t\t\t\t\t\t\t\t\t\t ... \t\t\t\t\t\t"
    var remaining = '' + string;
    remaining = remaining.replace("[/][*](\s|.)+?[*][/]", '');
```

Listing 1: A vulnerable regular expression in library `cytoscape`.

```
def _negotiate_value(response):
    if hasattr(_negotiate_value, 'regex'):
        regex = _negotiate_value.regex
    else:
        regex = re.compile('(?:.*,)*\s*Negotiate\s*([^,]*),?', re.I)
        # vulnerable to ",,,,,,,,,,,,,,,,,, ... ,,,,,,,,,,,,,,,,,\b"

    if authreq:
        match_obj = regex.search(authreq)
```

Listing 2: A vulnerable regular expression in libraries `urllib2-kerberos` and `requests-kerberos`.

REVEALER detected 53 and 169 vulnerable regexes in Python and JavaScript projects, respectively. We cross-validated the vulnerabilities, and got 13 and 32 regexes in Python and JavaScript projects that trigger a 10-second timeout with attack strings no longer than 128, respectively. We discuss several interesting cases next.

Branch in Loop vulnerability. `OmniDB` (2.2K stars) is a lightweight web application for database management. One of its library `cytoscape.js` contains a vulnerable regex in the code of core functions, as shown in Listing 1. An attacker could provide a 128-character long attack string for the vulnerable style function, which would take the `OmniDB` server more than 24 hours to perform the match. Other users of this web application would therefore get affected.

The code snippet tries to remove comments from a style string. However, this regex contains the vulnerable structure "Branch in Loop", in which branch `\s` and branch `.` both accept a blank character, *e.g.*, the TAB symbol `\t`. To mitigate the vulnerability, we need to remove the overlapping parts of the two branches. In this case, since `.` generally includes `\s`, we can simply remove the latter branch.

Loop in Loop vulnerability. `hue` (4.1K stars) is a popular open-source SQL query assistant for databases and warehouses. It uses two libraries `urllib2-kerberos` and `requests-kerberos` that both contain such a vulnerable regex in the authentication handler, as shown in Listing 2. Since `requests-kerberos` is a popular library which is also used by other 600+ repositories, we also reported this vulnerability to `requests-kerberos`. If an attacker send the attack string to these two libraries for Kerberos HTTP negotiate authentication, it is likely that such authentication would take a large amount of time (more than 24 hours with a 128-character long attack string), thus affects other users of the corresponding databases and warehouses.

As shown, there is a "Loop in Loop" vulnerable structure, *i.e.*, `(?:.*,)*`, in which a sequence of `','` can be matched by both the outer and the inner loop. To solve the problem, we can change either loop into a quantifier not supporting backtracking. A quick fix is to use a possessive quantifier by

```
var d = "^[\_$a-z][\_$a-z0-9]*(\[.*?\])*(\.[\_$a-z][\_$a
    -z0-9]*\(\[.*?\])*)*$"
// vulnerable to "_[._[]._[]._[]._[]._[] ... ._[]._
    []\b"
e = ["true", "false"]; return {
D: [], Y: function (b) {
    k = e.substring(d, j +
        1); b.push(k); var n = "@ko_token_" + (b.length
     - 1) + "@",
        e = e.substring(0, d) + n + e.substring(j + 1),
        j = j - (k.length - n.length), d = o
```

Listing 3: A vulnerable regular expression in library `dropdownlist`.

placing an extra '+' after the original quantifier '*', but since possessive quantifier is not supported in Python, we can change the structure to `(?:.*,)`.

Loop after Loop vulnerability. `kendo-ui-core` (2.3K stars) is a HTML5, jQuery-based widget library for building modern web applications. The code in its `dropdownlist` library contains a vulnerable regex, as shown in Listing 3. If the attacker is able to pass a 128-character long attack string to the library, the corresponding web application would hang for 71.92 seconds.

Specifically, there is a "Loop after Loop" structure, where two adjacent loops, *i.e.*, '.*?' in `(\[.*?\])*` and `(\.[\_$a-z][\_$a-z0-9]*(\[.*?\])*)*`, both accept "._[]". For this kind of vulnerabilities, we need to separate the two loops by another expression that does not accept "._[]" so that these two loops will not be directly connected. Alternatively, we can remove the overlapping parts in the two loops. For example, we can modify '.*?' to one that will not overlap with the latter loop.

Responsible Disclosure. We are unable to notify the authors of the 213 newly detected vulnerable regexes in §VI-B as the dataset does not include the source of a regex. We did contact the relevant developers about new vulnerabilities we detected in §VI-E (including the above three) and are in the process of obtaining new CVE IDs. At the time of writing, 6 developers have confirmed and 2 have fixed the vulnerabilities.

VII. Related Work

Empirical Study of Regular Expression. Bates *et al.* found regular expressions could be exploited to bypass XSS filters [1]. Champan *et al.* studied the usage of regular expressions in Python [6]. Wang *et al.* measured the test coverage of regular expressions in [36]. Rex solves regular expression constraints using a symbolic representation [35]. EGRET and Reggae generate test strings to help identify flawed regular expressions [19, 20]. In [23] and [13], the authors investigated management difficulties and the portability problem of regular expressions. REVEALER specifically targets the problem of super-linear matching complexity in regular expressions and is orthogonal to these works.

Empirical Studies of ReDoS. ReDoS is found harmful in many application scenarios. Smith *et al.* found catastrophic backtracking being utilized to evade network intrusion detection [31]. Davis *et al.* investigated the incidence of super-linear regular expressions and claimed ReDoS as a common security vulnerability [11]. Staicu *et al.* found ReDoS attacks could compromise the availability of JavaScript-based web servers [32]. These works are orthogonal to REVEALER, which focuses on detecting and exploiting ReDoS vulnerabilities.

ReDoS Detection. Previous works have studied the detection of ReDoS vulnerabilities. Berglund *et al.* proposed static analysis methods to identify vulnerable regexes based on a novel automaton model [3]. Sugiyama *et al.* measured the complexity of regular expression matching by simulating the matching using a tree transducer and analyzing the size increase of the output tree [33]. Similarly, RXXR2 builds search trees from regular expressions and characterizes the exponential branching blowup in the tree as a symbol of super-linear complexity [29]. These works focus on detecting regex structures with an exponential matching complexity and would miss the polynomial vulnerabilities. Weideman *et al.* [37] and Rexploiter [38], however, can identify vulnerable NFA patterns with exponential or polynomial complexity. But all these static analysis methods support only a limited set of extended features, and have high false positive because they cannot verify the attack strings they generated. REVEALER supports all extended features and reports only true positives by verifying the vulnerabilities in dynamic analysis. ReScue is a fuzz testing technique specifically optimized for ReDoS [30]. It generates the attack strings using a genetic algorithm, which makes it less efficient than static analysis methods.

Several existing works for Algorithmic DoS detection can also be extended to ReDoS, *e.g.*, fuzzers like SlowFuzz [28], HotFuzz [5], and hybrid approaches like Badger [25].

ReDoS Mitigation. Several existing works proposed to mitigate ReDoS vulnerabilities by modifying the structure of regular expressions. Becchi *et al.* proposed to merge non-equivalent states by labeling the transitions [2]. In [7], the authors searched for variants of regular expressions with better performance. Merwe *et al.* removed ambiguity from regular expressions to mitigate algorithmic DoS vulnerabilities [34]. Some other works proposed flexible resource allocation methods to limit the impact of ReDoS attacks. Lin *et al.* used hierarchical parallel methods on GPU to accelerate regular expression matching [21]. DeDoS mitigates asymmetric DoS attacks by deploying the program in a replicable fusion [14]. Davis *et al.* incorporated timeouts at the event handler level to mitigate ReDoS [12]. They also proposed to dynamically allocate memory to cope with extreme situations in [10]. We aim to detect ReDoS vulnerabilities and provide useful information to help developers mitigate ReDoS.

VIII. Conclusion

Regular expression Denial-of-Service (ReDoS) attacks can severely degrade the performance and availability of an application and its hosting server. In this paper, we present REVEALER, which is a hybrid-approach system that automatically detects and exploits ReDoS vulnerabilities. We statically model the ReDoS vulnerabilities for regular expressions with extended features such that REVEALER can locate the vulnerable subexpressions of a regex. REVEALER then dynamically simulates regex matching to generate attack strings to trigger the worst case matching of a potential vulnerable regex. We thoroughly evaluated the effectiveness and efficiency of REVEALER on a benchmark dataset and on real-world popular applications. We demonstrated that REVEALER can significantly outperform the state-of-the-art ReDoS detection tools, and can effectively and efficiently detect unknown ReDoS vulnerabilities.

Acknowledgment

The authors would like to thank Andrej Bogdanov, Siu On Chan and the anonymous reviewers for their helpful feedback. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14210219).

References

[1] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, Apr. 2010.

[2] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of the 26th IEEE International Conference on Computer Communications (INFOCOM)*, Barcelona, Spain, May 2007.

[3] M. Berglund, F. Drewes, and B. van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. *Electronic Proceedings in Theoretical Computer Science*, 151:109–123, may 2014. doi: 10.4204/eptcs.151.7. URL https://doi.org/10.4204%2Feptcs.151.7.

[4] M. Berglund, W. Bester, and B. van der Merwe. Formalising boost posix regular expression matching. In B. Fischer and T. Uustalu, editors, *Theoretical Aspects of Computing – ICTAC 2018*, pages 99–115, Cham, 2018. Springer International Publishing.

[5] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[6] 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 (ISSTA)*, Saarbrücken, Germany, July 2016.

[7] 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*, GECCO '17, pages 1280–1287, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349208. doi: 10.1145/3071178.3071196. URL https://doi.org/10.1145/3071178.3071196.

[8] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[9] C. CÂMPEANU, K. SALOMAA, and S. YU. A formal study of practical regular expressions. *International Journal of Foundations of Computer Science*, 14(06):1007–1018, 2003. doi: 10.1142/S012905410300214X.

[10] J. C. Davis. Rethinking regex engines to address redos. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, Aug. 2019.

[11] 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 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista, FL, Nov. 2018.

[12] J. C. Davis, E. R. Williamson, and D. Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[13] J. C. Davis, L. G. M. 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 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, Aug. 2019.

[14] H. M. Demoulin, T. Vaidya, I. Pedisich, B. DiMaiolo, J. Qian, C. Shah, Y. Zhang, A. Chen, A. Haeberlen, B. T. Loo, L. T. X. Phan, M. Sherr, C. Shields, and W. Zhou. DeDoS:defusing dos with dispersion oriented software. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, San Juan, PR, Dec. 2018.

[15] M. Docs. Backtracking in .net regular expressions. "https://docs.microsoft.com/en-us/dotnet/standard/base-types/backtracking-in-regular-expressions", 2020.

[16] M. J. Dominus. Perl regular expression matching is np-hard, 2020. URL https://perl.plover.com/NPC/.

[17] J. Friedl. *Mastering Regular Expressions: Understand Your Data and Be More Productive*. O'Reilly Media, 2006. ISBN 9781449332532. URL https://books.google.com.hk/books?id=sshKXlr32-AC.

[18] J. Kirrage, A. Rathnayake, and H. Thielecke. Static analysis for regular expression denial-of-service attacks. In J. Lopez, X. Huang, and R. Sandhu, editors, *Network and System Security*, pages 135–148, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] E. Larson and A. Kirk. Generating evil test strings for regular expressions. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2016. doi: 10.1109/icst.2016.29. URL https://doi.org/10.1109%2Ficst.2016.29.

[20] N. Li, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Reggae: Automated test generation for programs using complex regular expressions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Auckland, New Zealand, Nov. 2009.

[21] C. Lin, C. Liu, and S. Chang. Accelerating regular expression matching using hierarchical parallel machines on gpu. In *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011*, pages 1–5, 2011.

[22] 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 (PLDI)*, New York, NY, June 2019.

[23] 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 *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2019.

[24] R. Mitkov. *The Oxford Handbook of Computational Linguistics*. Oxford University Press, 2003. ISBN 9780199276349. URL https://books.google.com/books?id=yl6AnaKtVAkC&pg=PA754.

[25] Y. Noller, R. Kersten, and C. S. Păsăreanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*, Amsterdam, Netherlands, July 2018.

[26] A. Ojamaa and K. Düüna. Security assessment of node.js platform. In *Information Systems Security: 8th International Conference, ICISS 2012, Guwahati, India, December 15-19, 2012, Proceedings*, page 40. Springer Science & Business Media, 2012. URL https://books.google.com.hk/books?id=xOrfxw3OcTUC.

[27] OWASP. Regex denial of service. "http://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS", 2010.

[28] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct.–Nov. 2017.

[29] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics (extended). *arXiv preprint arXiv:1405.7058*, 2014.

[30] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, Sept. 2018.

[31] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a nids. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[32] C.-A. Staicu and M. Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[33] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. *IPSJ Online Transactions*, 7(0):82–92, 2014. doi: 10.2197/ipsjtrans.7.82. URL https://doi.org/10.2197%2Fipsjtrans.7.82.

[34] 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 on - SAICSIT '17*. ACM Press, 2017. doi: 10.1145/3129416.3129440. URL https://doi.org/10.1145%2F3129416.3129440.

[35] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010. doi: 10.1109/icst.2010.15. URL https://doi.org/10.1109%2Ficst.2010.15.

[36] P. Wang and K. T. Stolee. How well are regular expressions tested in the wild? In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Lake Buena Vista, FL, Nov. 2018.

[37] 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*, pages 322–334. Springer International Publishing, 2016. doi: 10.1007/978-3-319-40946-7_27. URL https://doi.org/10.1007%2F978-3-319-40946-7_27.

[38] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Uppsala, Sweden, Apr. 2017.

[39] S. Yu. *Regular Languages*. Springer Berlin Heidelberg, 1997. doi: 10.1007/978-3-642-59136-5_2. URL https://doi.org/10.1007%2F978-3-642-59136-5_2.

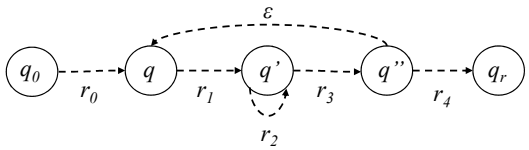


Fig. 10: NFA-like structure transformed from the "Loop in Loop" vulnerable e-NFA structure. r_0 to r_4 are the same subexpressions as those in the vulnerable e-NFA structure.

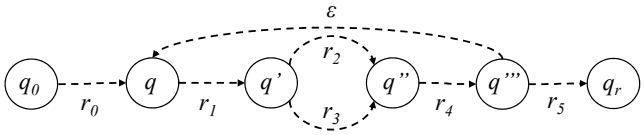


Fig. 11: NFA-like structure transformed from the "Branch in Loop" vulnerable structure.

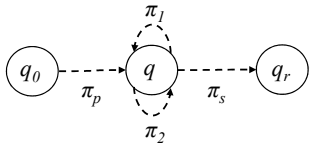


Fig. 8: The structure of hyper-vulnerable NFA pattern.

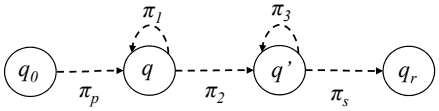


Fig. 9: The structure of vulnerable NFA pattern.

APPENDIX

A. Additional Proofs for Modeling ReDoS Vulnerability

We provide the proofs for the four theorems we defined in §IV. We first introduce the existing theorems about vulnerable NFA patterns (§A1) that inspired our modeling. We then prove the theorem about the crucial states in an e-NFA that may lead to super-linear matching behavior (§A2). Finally we provide proofs for our proposed vulnerable e-NFA patterns (§A3).

1) Vulnerable NFA Patterns: Before we introduce our theory in vulnerable e-NFA patterns, we review necessary background of vulnerable NFA patterns and corresponding attack string patterns in [38].

Definition 5 (NFA)**.** *An NFA A is a 5-tuple $(Q, \Sigma, \Delta, q_0, F)$ where Q is a finite set of states, Σ is a finite alphabet of symbols, and $\Delta : Q \times \Sigma \to 2^Q$ is the transition function. Here, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. We say that (q, l, q') is a transition via label l if $q' \in \Delta(q, l)$.*

The notation π denotes an NFA path, which includes a sequence of NFA transitions $(q_1, l_1, q_2), \ldots, (q_{m-1}, l_{m-1}, q_m)$ that starts at q_1 and ends at q_m. $labels(\pi)$ denotes the sequence of labels $(l_1, \ldots, l_{m-1})$ for following the transitions in path π.

Theorem 5 (Hyper-vulnerable NFA pattern)**.** *An NFA is hyper-vulnerable (has exponential complexity) **iff** there exists a pivot state $q \in Q$ and two distinct paths π_1, π_2 such that **(i)** both π_1, π_2 start and end at q, **(ii)** $labels(\pi_1) = labels(\pi_2)$, **(iii)** there is a path π_p from initial state q_0 to q, and **(iv)** there is a path π_s from q to a state $q_r \notin F$, as shown in Figure 8.*

Theorem 6 (Vulnerable NFA pattern)**.** *An NFA is vulnerable (has super-linear complexity) **iff** there exists two states $q \in Q$, $q' \in Q$, and three paths π_1, π_2, and π_3 (where $\pi_1 \neq \pi_2$) such that **(i)** π_1 starts and ends at q, **(ii)** π_2 starts at q and ends at q', **(iii)** π_3 starts and ends at q', **(iv)** $labels(\pi_1) = labels(\pi_2) = labels(\pi_3)$, **(v)** there is a path π_p from initial state q_0 to q, and **(vi)** there is a path π_s from q' to a state $q_r \notin F$, as shown in Figure 9.*

The ReDoS attack string pattern against the vulnerable NFA patterns is proposed as $s_0.s^k.s_1$, where s_0 is the *attack prefix* given by $labels(\pi_p)$, s_1 is the *attack suffix* given by $labels(\pi_s)$, and s is the *attack core* given by $labels(\pi_1)$. For example, regex `(ab|a|b)*` has a hyper-vulnerable NFA pattern, whose attack core is `'ab'`. The attack core has two distinct matching paths—`(ab){1}` for π_1, and `(a|b){2}` for π_2 which is a repetition of the subexpression `(a|b)`. Both $labels(\pi_1)$ and $labels(\pi_2)$ are equal to the attack core `'ab'`.

2) Crucial States in e-NFA: We provide the proof for Theorem 1 that the e-NFA match process runs in linear time if an e-NFA has neither loop states nor branch states.

Proof of Theorem 1: For a state v that is neither a loop state nor a branch state, it has at most one outgoing inclusion transition, and $c_v^{max} \leq 1$, which means it can match a symbol at most once and the e-NFA has to take a connection transition after the match. Since a state can have at most one outgoing connection transition as described in §III-A4, the e-NFA can transit from v to only one state v' through the only connection transition t. Let $P_v(s)$ denote the number of possible matching paths from the start state v_0 to v before matching the next substring s, and s' denote the remaining unmatched substring after following the transition t from v to v'. We have $P_{v'}(s') = P_v(s)$. If the e-NFA contains only such states, then all states would be visited at most once, and the possible matching paths count keeps the same from the start state to the accepting state, *i.e.*, $P_{v_f}(\epsilon) = P_{v_0}(s_0) = 1$, where s_0 is the entire input string. Since the maximum length for each matching path L_{max} is less than $M \times N$, where M is the number of states in e-NFA, and N is the length of s_0, the overall matching time would be less than $P_{v_f}(\epsilon) \times L_{max}$, and is linear with N. ■

3) The Sufficiency of Vulnerable e-NFA Patterns: In this section, we prove the sufficiency of vulnerable e-NFA patterns.

We first discuss two vulnerable e-NFA patterns with exponential complexity: "Loop in Loop" and "Branch in Loop".

Loop in Loop. The "Loop in Loop" vulnerable structure and the corresponding NFA-like structure are shown in Figure 2 and Figure 10, respectively. We can see similar structures between Figure 8 and Figure 10 by mapping π_p to r_0, π_1 to r_1r_3, π_2 to $r_1r_2r_3$, and π_s to $r_1r_3r_4$. We propose the following proof for the "Loop in loop" vulnerable e-NFA structure.

Proof of Theorem 2: Let s denote the common match string of $r_1r_2r_3$ and r_1r_3, and there are k repetitions of s in the attack string. When the regex engine fails to match r_4 after accepting s^k at v_i, for each s, it can backtrack to v_i by either $r_1r_2r_3$ or r_1r_3. Each backtracking of string s takes constant time $O(1)$. Let $T_v(k)$ denote the running time of backtracking from state v on string s^k. For each repetition of s, the total number of backtracking paths doubles, and the total running

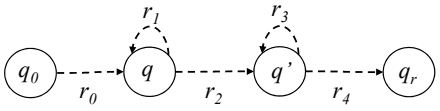


Fig. 12: NFA-like structure transformed from the "Loop after Loop" vulnerable structure.

time becomes: $T_{v_i}(k) = (O(1)+T_{v_i}(k-1))+(O(1)+T_{v_i}(k-1))$. Iteratively, we have $T_{v_i}(k) = O(1) \cdot (2^{k+1}-2) = O(2^k)$. In summary, there are 2^k possible backtracking paths, and the regex engine would end up traversing all possibilities in the worst case, which is of exponential time with k. ■

Branch in Loop. The "Branch in Loop" vulnerable structure and the corresponding NFA-like structure are shown in Figure 3 and Figure 11, respectively. Similarly, paths in Figure 8 can be mapped to subexpressions in Figure 11 as follows: π_p to r_0, π_1 to $r_1r_2r_4$, π_2 to $r_1r_3r_4$, and π_s to $r_1r_3r_4r_5$. We propose the following proof.

Proof of Theorem 3: Let s denote the common match string of $r_1r_2r_4$ and $r_1r_3r_4$, and there are k repetitions of s in the attack string. Similar to Theorem 2, the number of backtracking possibilities doubles on each repetition of s, which leads to exponential time complexity. ■

We then discuss the vulnerable e-NFA pattern "Loop after Loop", which is of polynomial complexity.

Loop after Loop. The "Loop after Loop" vulnerable structure and the corresponding NFA-like structure are shown in Figure 4 and Figure 12, respectively. Paths in Figure 9 and subexpressions in Figure 12 can be mapped as follows: π_p to r_0, π_1 to r_1, π_2 to r_2, π_3 to r_3 , and π_s to r_4. We propose the following proof.

Proof of Theorem 4: Let s denote the common match string of r_1, r_3 (and r_2), and there are k repetitions of s in the attack string. When the regex engine fails to match r_4 after accepting s^k at v_i, for each s, it can backtrack to v_i by either $r_1r_2r_3$ or r_1r_3. Each backtracking step of string s takes constant time $O(1)$. Let $T_v(k)$ denote the running time of backtracking from state v on string s^k. The backtracking algorithm starts from v_{i+1} with s^k. When the engine backtracks the first s, it either goes to v_i or stays at v_{i+1}, which gives: $T_{v_{i+1}}(k) = (O(1) + T_{v_i}(k-1)) + (O(1) + T_{v_{i+1}}(k-1))$. If the engine goes to v_i, it cannot come back to v_{i+1}, so it can only perform backtracking on v_i afterwards. Thus, we have $T_{v_i}(k-1) = (k-1) \cdot O(1) = O(k)$, and then $T_{v_{i+1}}(k) = T_{v_{i+1}}(k-1)+O(k)$. Iteratively we have $T_{v_{i+1}}(k) = k \cdot O(k) = O(k^2)$. In summary, the regex engine would end up running in polynomial time with k in the worst case. ■