Static Detection of DoS Vulnerabilities in Programs that use Regular Expressions (Extended Version)

Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig
The University of Texas at Austin
{valentin, olivo, marijn, isil}@cs.utexas.edu

Abstract. In an *algorithmic complexity attack*, a malicious party takes advantage of the worst-case behavior of an algorithm to cause denial-of-service. A prominent algorithmic complexity attack is *regular expression denial-of-service* (*ReDoS*), in which the attacker exploits a vulnerable regular expression by providing a carefully-crafted input string that triggers worst-case behavior of the matching algorithm. This paper proposes a technique for automatically finding ReDoS vulnerabilities in programs. Specifically, our approach automatically identifies *vulnerable regular expressions* in the program and determines whether an "evil" input string can be matched against a vulnerable regular expression. We have implemented our proposed approach in a tool called Rexploiter and found 41 exploitable security vulnerabilities in Java web applications.

1 Introduction

Regular expressions provide a versatile mechanism for parsing and validating input data. Due to their flexibility, many developers use regular expressions to validate passwords or to extract substrings that match a given pattern. Hence, many languages provide extensive support for regular expression matching.

While there are several algorithms for determining membership in a regular language, a common technique is to construct a non-deterministic finite automaton (NFA) and perform backtracking search over all possible runs of this NFA. Although simple and flexible, this strategy has super-linear (in fact, exponential) complexity and is prone to a class of *algorithmic complexity attacks* [14]. For some regular expressions (e.g., `(a|b)*(a|c)*`), it is possible to craft input strings that could cause the matching algorithm to take quadratic time (or worse) in the size of the input. For some regular expressions (e.g, `(a+)+`), one can even generate input strings that could cause the matching algorithm to take exponential time. Hence, attackers exploit the presence of vulnerable regular expressions to launch so-called *regular expression denial-of-service (ReDoS)* attacks.

ReDoS attacks have been shown to severely impact the responsiveness and availability of applications. For example, the .NET framework was shown to be vulnerable to a ReDoS attack that paralyzed applications using .NET's default validation mechanism [2]. Furthermore, unlike other DoS attacks that require thousands of machines to bring down critical infrastructure, ReDoS attacks can

be triggered by a single malicious user input. Consequently, developers are responsible for protecting their code against such attacks, either by avoiding the use of vulnerable regular expressions or by *sanitizing* user input.

Unfortunately, protecting an application against ReDoS attacks can be non-trivial in practice. Often, developers do not know which regular expressions are vulnerable or how to rewrite them in a way that avoids super-linear complexity. In addition, it is difficult to implement a suitable sanitizer without understanding the class of input strings that trigger worst-case behavior. Even though some libraries (e.g., the .NET framework) allow developers to set a time limit for regular expression matching, existing solutions do not address the root cause of the problem. As a result, ReDoS vulnerabilities are still being uncovered in many important applications. For instance, according to the National Vulnerability Database (NVD), there are over 150 acknowledged ReDoS vulnerabilities, some of which are caused by exponential matching complexity (e.g., [2,3]) and some of which are characterized by super-linear behavior (e.g., [1,4,5]).

In this paper, we propose a static technique for automatically uncovering DoS vulnerabilities in programs that use regular expressions. There are two main technical challenges that make this problem difficult: First, given a regular expression $\mathcal{E}$, we need to statically determine the worst-case complexity of matching $\mathcal{E}$ against an arbitrary input string. Second, given an application A that contains a vulnerable regular expression $\mathcal{E}$, we must statically determine whether there can exist an execution of A in which $\mathcal{E}$ can be matched against an input string that could cause super-linear behavior.

We solve these challenges by developing a two-tier algorithm that combines (a) static analysis of regular expressions with (b) sanitization-aware taint analysis at the source code level. Our technique can identify both *vulnerable* regular expressions that have super-linear complexity (quadratic or worse), as well as *hyper-vulnerable* ones that have exponential complexity. In addition and, most importantly, our technique can also construct an *attack automaton* that captures all possible attack strings. The construction of attack automata is crucial for reasoning about input sanitization at the source-code level.

To summarize, this paper makes the following contributions:

- We present algorithms for reasoning about worst-case complexity of NFAs. Given an NFA $\mathcal{A}$, our algorithm can identify whether $\mathcal{A}$ has linear, super-linear, or exponential time complexity and can construct an *attack automaton* that accepts input strings that could cause worst-case behavior for $\mathcal{A}$.
- We describe a program analysis to automatically identify ReDoS vulnerabilities. Our technique uses the results of the regular expression analysis to identify *sinks* and reason about input sanitization using attack automata.
- We use these ideas to build an end-to-end tool called REXPLOITER for finding vulnerabilities in Java. In our evaluation, we find 41 security vulnerabilities in 150 Java programs collected from Github with a 11% false positive rate.

```
1 public class RegExValidator {
2   boolean validEmail(String t) { return t.matches(".+@.+\\.[a-z]+"); }
3   boolean validComment(String t) {
4     return !t.matches("(\\p{Blank}*(\\r?\\n)\\p{Blank}*)+"); }
5   boolean safeComment(String t) { return t.matches("([^\/<>])+"); }
6   boolean validUrl(String t) {
7     return t.matches("www\\.shoppers\\.com/.+/.+/.+/.+/"); }
8 }
9 public class CommentFormValidator implements Validator {
10   private Admin admin;
11   public void validate(CommentForm form, Errors errors) {
12     String senderEmail = form.getSenderEmail();
13     String productUrl = form.getProductUrl();
14     String comment = form.getComment();
15     if (!RegExValidator.validEmail(admin.getEmail())) return;
16     if (senderEmail.length() <= 254) {
17       if (RegExValidator.validEmail(senderEmail)) ... }
18     if (productUrl.split("/").length == 5) {
19       if (RegExValidator.validUrl(productUrl)) ... }
20     if (RegExValidator.safeComment(comment)) {
21       if (RegExValidator.validComment(comment)) ... }
22   }
```

Fig. 1: Motivating example containing ReDoS vulnerabilities

2 Overview

We illustrate our technique using the code snippet shown in Fig. 1, which shows two relevant classes, namely `RegExValidator`, that is used to validate that certain strings match a given regular expression, and `CommentFormValidator`, that checks the validity of a comment form filled out by a user. In particular, the comment form submitted by the user includes the user's email address, the URL of the product about which the user wishes to submit a comment[1], and the text containing the comment itself. We now explain how our technique can determine whether this program contains a denial-of-service vulnerability.

Regular expression analysis. For each regular expression in the program, we construct its corresponding NFA and statically analyze it to determine whether its worst-case complexity is linear, super-linear, or exponential. For our running example, the NFA complexity analysis finds instances of each category. In particular, the regular expression used at line 5 has linear matching complexity, while the one from line 4 has exponential complexity. The regular expressions from lines 2 and 7 have super-linear (but not exponential) complexity. Fig. 2 plots input size against running time for the regular expressions from lines 2 and 4 respectively. For the super-linear and exponential regular expressions, our technique also constructs an attack automaton that recognizes all strings that cause worst-case behavior. In addition, for each regular expression, we determine a lower bound on the length of any possible attack string using dynamic analysis.

Program analysis. The presence of a vulnerable regular expression does not necessarily mean that the program itself is vulnerable. For instance, the vulnerable regular expression may not be matched against an attacker-controlled string,

[1] Due to the store's organization, the URL is expected to be of the form `www.shoppers.com/Dept/Category/Subcategory/product-id/`

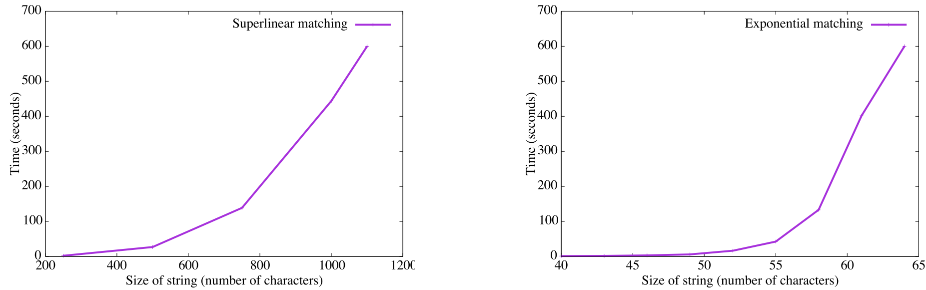

Fig. 2: Matching time against malicious string size for vulnerable (left) and hyper-vulnerable (right) regular expressions from Fig. 1.

or the program may take measures to prevent the user from supplying a string that is an instance of the attack pattern. Hence, we also perform static analysis at the source code level to determine if the program is actually vulnerable.

Going back to our example, the `validate` procedure (lines 11–22) calls `validEmail` to check whether the website administrator's email address is valid. Even though `validEmail` contains a super-linear regular expression, line 15 does not contain a vulnerability because the administrator's email is not supplied by the user. Since our analysis tracks taint information, it does not report line 15 as being vulnerable. Now, consider the second call to `validEmail` at line 17, which matches the vulnerable regular expression against user input. However, since the program bounds the size of the input string to be at most 254 (which is smaller than the lower bound identified by our analysis), line 17 is also not vulnerable.

Next, consider the call to `validUrl` at line 19, where `productUrl` is a user input. At first glance, this appears to be a vulnerability because the matching time of the regular expression from line 4 against a malicious input string grows quite rapidly with input size (see Fig. 2). However, the check at line 18 actually prevents calling `validUrl` with an attack string: Specifically, our analysis determines that attack strings must be of the form `www.shoppers.com`$\cdot$`/`$^b\cdot$`/`$^+\cdot$`x`, where `x` denotes any character and b is a constant inferred by our analysis (in this case, much greater than 5). Since our program analysis also reasons about input sanitization, it can establish that line 19 is safe.

Finally, consider the call to `validComment` at line 21, where `comment` is again a user input and is matched against a regular expression with exponential complexity. Now, the question is whether the condition at line 20 prevents `comment` from conforming to the attack pattern `\n\t\n\t( \t\n\t)`^{k}`a`. Since this is not the case, line 21 actually contains a serious DoS vulnerability.

Summary of challenges. This example illustrates several challenges we must address: First, given a regular expression $\mathcal{E}$, we must reason about the worst-case time complexity of its corresponding NFA. Second, given vulnerable regular expression $\mathcal{E}$, we must determine whether the program allows $\mathcal{E}$ to be matched against a string that is (a) controlled by the user, (b) is an instance of the attack pattern for regular expression $\mathcal{E}$, and (c) is large enough to cause the matching algorithm to take significant time.

Our approach solves these challenges by combining complexity analysis of NFAs with sanitization-aware taint analysis. The key idea that makes this combination possible is to produce an attack automaton for each vulnerable NFA. Without such an attack automaton, the program analyzer cannot effectively determine whether an input string can correspond to an attack string.

As shown in Fig. 3, the REXPLOITER toolchain incorporates both static and dynamic regular expression analysis. The static analysis creates attack patterns $s_0 \cdot s^k \cdot s_1$ and dynamic analysis infers a lower bound b on the number of occurrences of s in order to exceed a minimum runtime threshold. The program analysis uses both the attack automaton and the lower bound b to reason about input sanitization.

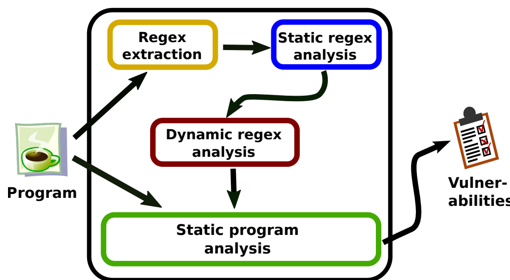

Fig. 3: Overview of our approach

3 Preliminaries

This section presents some useful background and terminology.

Definition 1. (NFA) *An NFA $\mathcal{A}$ 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 $q' \in \Delta(q, l)$.*

An NFA $\mathcal{A}$ accepts a string $s = a_0 a_1 \dots a_n$ iff there exists a sequence of states $q_0, q_1, ..., q_n$ such that $q_n \in F$ and $q_{i+1} \in \Delta(q_i, a_i)$. The language of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of all strings that are accepted by $\mathcal{A}$. Conversion from a regular expression to an NFA is sometimes referred to as *compilation* and can be achieved using well-known techniques, such as Thompson's algorithm [25].

In this paper, we assume that membership in a regular language $\mathcal{L}(\mathcal{E})$ is decided through a worst-case exponential algorithm that performs backtracking search over possible runs of the NFA representing $\mathcal{E}$. While there exist linear-time matching algorithms (e.g., based on DFAs), many real-world libraries employ backtracking search for two key reasons: First, the compilation of a regular expression is much faster using NFAs and uses much less memory (DFA's can be exponentially larger). Second, the backtracking search approach can handle regular expressions containing extra features like backreferences and lookarounds. Thus, many widely-used libraries (e.g., `java.util.regex`, Python's standard library) employ backtracking search for regular expression matching.

In the remainder of this paper, we will use the notation $\mathcal{A}^*$ and $\mathcal{A}^\emptyset$ to denote the NFA that accepts Σ^* and the empty language respectively. Given two NFAs

$\mathcal{A}_1$ and $\mathcal{A}_2$, we write $\mathcal{A}_1 \cap \mathcal{A}_2$, $\mathcal{A}_1 \cup \mathcal{A}_2$, and $\mathcal{A}_1 \cdot \mathcal{A}_2$ to denote automata intersection, union, and concatenation. Finally, given an automaton $\mathcal{A}$, we write $\overline{\mathcal{A}}$ to represent its complement, and we use the notation $\mathcal{A}^+$ to represent the NFA that recognizes exactly the language $\{s^k \mid k \geq 1 \wedge s \in \mathcal{L}(\mathcal{A})\}$.

Definition 2. (Path) *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, a path π of $\mathcal{A}$ is a sequence of transitions $(q_1, \ell_1, q_2), \ldots, (q_{m-1}, \ell_{m-1}, q_m)$ where $q_i \in Q$, $\ell_i \in \Sigma$, and $q_{i+1} \in \Delta(q_i, \ell_i)$. We say that π starts in q_i and ends at q_m, and we write* labels(π) *to denote the sequence of labels $(\ell_1, \ldots, \ell_{m-1})$.*

4 Detecting Hyper-Vulnerable NFAs

In this section, we explain our technique for determining if an NFA is *hyper-vulnerable* and show how to generate an *attack automaton* that recognizes exactly the set of attack strings.

Definition 3. (Hyper-Vulnerable NFA) *An NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is hyper-vulnerable iff there exists a backtracking search algorithm* Match *over the paths of $\mathcal{A}$ such that the worst-case complexity of* Match *is exponential in the length of the input string.*

We will demonstrate that an NFA $\mathcal{A}$ is hyper-vulnerable by showing that there exists a string s such that the number of distinct matching paths π_i from state q_0 to a rejecting state q_r with $labels(\pi_i) = s$ is exponential in the length of s. Clearly, if s is rejected by $\mathcal{A}$, then Match will need to explore each of these exponentially many paths. Furthermore, even if s is accepted by $\mathcal{A}$, there exists a backtracking search algorithm (namely, the one that explores all rejecting paths first) that results in exponential worst-case behavior.

Theorem 1. *An NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is* hyper-vulnerable *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(π_1) = labels(π_2)*, and (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 $q_r \notin F$.*

Proof. The sufficiency argument is laid out below, and the necessity argument can be found in the appendix.

To gain intuition about hyper-vulnerable NFAs, consider Fig. 4 illustrating the conditions of Theorem 1. First, a hyper-vulnerable NFA must contain a *pivot state* q, such that, starting at q, there are two different ways (namely, π_1, π_2) of getting back to q on the same input string s (i.e., *labels*(π_1)). Second, the pivot state q should be reachable from the initial state q_0, and there must be a way of reaching a rejecting state q_r from q.

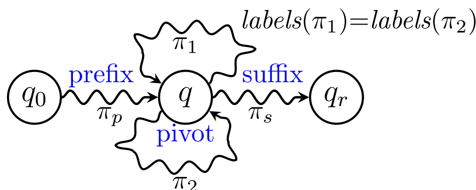

Fig. 4: Hyper-vulnerable NFA pattern

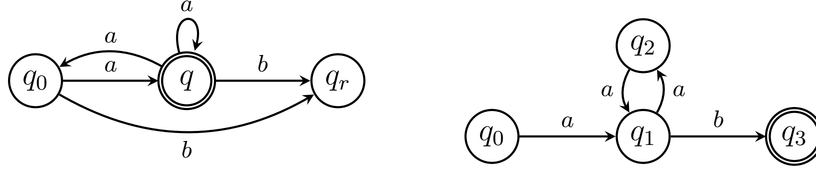

Fig. 5: A hyper-vulnerable NFA (left) and an attack automaton (right).

To understand why these conditions cause exponential behavior, consider a string of the form $s_0 \cdot s^k \cdot s_1$, where s_0 is the *attack prefix* given by $\mathit{labels}(\pi_p)$, s_1 is the *attack suffix* given by $\mathit{labels}(\pi_s)$, and s is the *attack core* given by $\mathit{labels}(\pi_1)$. Clearly, there is an execution path of $\mathcal{A}$ in which the string $s_0 \cdot s^k \cdot s_1$ will be rejected. For example, $\pi_p \cdot \pi_1^k \cdot \pi_s$ is exactly such a path.

Now, consider a string $s_0 \cdot s^{k+1} \cdot s_1$ that has an additional instance of the attack core s in the middle, and suppose that there are n possible executions of $\mathcal{A}$ on the prefix $s_0 \cdot s^k$ that end in q. Now, for each of these n executions, there are two ways to get back to q after reading s: one that takes path π_1 and another that takes path π_2. Therefore, there are $2n$ possible executions of $\mathcal{A}$ that end in q. Furthermore, the matching algorithm will (in the worst case) end up exploring all of these $2n$ executions since there is a way to reach the rejecting state q_r. Hence, we end up doubling the running time of the algorithm every time we add an instance of the attack core s to the middle of the input string.

Algorithm 1 Hyper-vulnerable NFA

1: **function** AttackAutomaton($\mathcal{A}$)
2: assume $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$
3: $\mathcal{A}^{\Psi} \leftarrow \mathcal{A}^{\emptyset}$
4: **for** $q_i \in Q$ **do**
5: $\mathcal{A}_i^{\Psi} \leftarrow$ AttackForPivot($\mathcal{A}, q_i$)
6: $\mathcal{A}^{\Psi} \leftarrow \mathcal{A}^{\Psi} \cup \mathcal{A}_i^{\Psi}$
7: **return** $\mathcal{A}^{\Psi}$
8: **function** AttackForPivot($\mathcal{A}, q$)
9: assume $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$
10: $\mathcal{A}^{\Psi} \leftarrow \mathcal{A}^{\emptyset}$
11: **for** $(q, l, q_1), (q, l, q_2) \in \Delta \wedge q_1 \neq q_2$ **do**
12: $\mathcal{A}_1 \leftarrow$ LoopBack($\mathcal{A}, q, l, q_1$)
13: $\mathcal{A}_2 \leftarrow$ LoopBack($\mathcal{A}, q, l, q_2$)
14: $\mathcal{A}_p \leftarrow (Q, \Sigma, \Delta, q_0, \{q\})$
15: $\mathcal{A}_s \leftarrow (Q, \Sigma, \Delta, q, F)$
16: $\mathcal{A}^{\Psi} \leftarrow \mathcal{A}^{\Psi} \cup (\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2)^+ \cdot \overline{\mathcal{A}_s})$
17: **return** $\mathcal{A}^{\Psi}$
18: **function** LoopBack($\mathcal{A}, q, l, q'$)
19: assume $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$
20: $q^{\star} \leftarrow$ NewState(Q)
21: $Q' \leftarrow Q \cup q^{\star}$; $\Delta' \leftarrow \Delta \cup (q^{\star}, l, q')$
22: **return** $(Q', \Sigma, \Delta', q^{\star}, \{q\})$

Example 1. The NFA in Fig. 5 (left) is hyper-vulnerable because there exist two different paths $\pi_1 = (q, a, q), (q, a, q)$ and $\pi_2 = (q, a, q_0), (q_0, a, q)$ that contain the same labels and that start and end in q. Also, q is reachable from q_0, and the rejecting state q_r is reachable from q. Attack strings for this NFA are of the form $a \cdot (a \cdot a)^k \cdot b$, and the attack automaton is shown in Fig. 5 (right).

We now use Theorem 1 to devise Algorithm 1 for constructing the attack automaton $\mathcal{A}^{\Psi}$ for a given NFA. The key idea of our algorithm is to search for all possible pivot states q_i and construct the attack automaton $\mathcal{A}_i^{\Psi}$ for state q_i. The full attack automaton is then obtained as the union of all $\mathcal{A}_i^{\Psi}$. Note that

Algorithm 1 can be used to determine if automaton $\mathcal{A}$ is vulnerable: $\mathcal{A}$ exhibits worst-case exponential behavior iff the language accepted by $\mathcal{A}^\Psi$ is non-empty.

In Algorithm 1, most of the real work is done by the AttackForPivot procedure, which constructs the attack automaton for a specific state q: Given a pivot state q, we want to find two different paths π_1, π_2 that loop back to q and that have the same set of labels. Towards this goal, line 11 of Algorithm 1 considers all pairs of transitions from q that have the same label (since we must have $labels(\pi_1) = labels(\pi_2)$).

Now, let us consider a pair of transitions $\tau_1 = (q, l, q_1)$ and $\tau_2 = (q, l, q_2)$. For each q_i ($i \in \{1, 2\}$), we want to find all strings that start in q, take transition τ_i, and then loop back to q. In order to find all such strings $\mathcal{S}$, Algorithm 1 invokes the LoopBack function (lines 18–22), which constructs an automaton $\mathcal{A}'$ that recognizes exactly $\mathcal{S}$. Specifically, the final state of $\mathcal{A}'$ is q because we want to loop back to state q. Furthermore, $\mathcal{A}'$ contains a new initial state q^* (where $q^* \notin Q$) and a single outgoing transition (q^*, l, q_i) out of q^* because we only want to consider paths that take the transition to q_i first. Hence, each $\mathcal{A}_i$ in lines 12–13 of the AttackForPivot procedure corresponds to a set of paths that loop back to q through state q_i. Observe that, if a string s is accepted by $\mathcal{A}_1 \cap \mathcal{A}_2$, then s is an attack core for pivot state q.

We now turn to the problem of computing the set of all attack prefixes and suffixes for pivot state q: In line 14 of Algorithm 1, $\mathcal{A}_p$ is the same as the original NFA $\mathcal{A}$ except that its only accepting state is q. Hence, $\mathcal{A}_p$ accepts all attack prefixes for pivot q. Similarly, A_s is the same as $\mathcal{A}$ except that its initial state is q instead of q_0; thus, $\overline{A_s}$ accepts all attack suffixes for q.

Finally, let us consider how to construct the full attack automaton $\mathcal{A}^\Psi$ for q. As explained earlier, all attack strings are of the form $s_1 \cdot s^k \cdot s_2$ where s_1 is the attack prefix, s is the attack core, and s_2 is the attack suffix. Since $\mathcal{A}_p$, $\mathcal{A}_1 \cap \mathcal{A}_2$, and $\overline{\mathcal{A}_s}$ recognize attack prefixes, cores, and suffixes respectively, any string that is accepted by $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2)^+ \cdot \overline{\mathcal{A}_s}$ is an attack string for the original NFA $\mathcal{A}$.

Theorem 2. (Correctness of Algorithm 1)[2] *Let $\mathcal{A}^\Psi$ be the result of calling* AttackAutomaton($\mathcal{A}$) *for NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. For every $s \in \mathcal{L}(\mathcal{A}^\Psi)$, there exists a rejecting state $q_r \in Q \setminus F$ s.t. the number of distinct paths π_i from q_0 to q_r with $labels(\pi_i) = s$ is exponential in the number of repetitions of the attack core in s.*

5 Detecting Vulnerable NFAs

So far, we only considered the problem of identifying NFAs whose worst-case running time is exponential. However, in practice, even NFAs with super-linear complexity can cause catastrophic backtracking. In fact, many acknowledged ReDoS vulnerabilities (e.g., [1,4,5]) involve regular expressions whose matching complexity is "only" quadratic. Based on this observation, we extend the techniques from the previous section to statically detect NFAs with super-linear time complexity. Our solution builds on insights from Section 4 to construct an attack automaton for this larger class of vulnerable regular expressions.

[2] The proofs of Theorems 2 and 4 are given in the appendix.

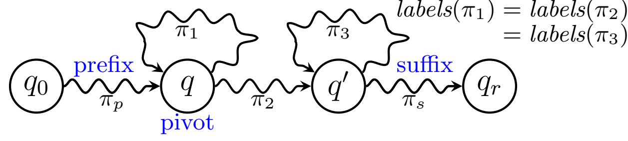

Fig. 6: General pattern characterizing vulnerable NFAs

5.1 Understanding Super-Linear NFAs

Before we present the algorithm for detecting super-linear NFAs, we provide a theorem that explains the correctness of our solution.

Definition 4. (Vulnerable NFA) *An NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is vulnerable iff there exists a backtracking search algorithm* MATCH *over the paths of $\mathcal{A}$ such that the worst-case complexity of* MATCH *is at least quadratic in the length of the input string.*

Theorem 3. *An NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is* vulnerable *iff there exist two states $q \in Q$ (*the pivot*), $q' \in Q$, 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)* $\text{labels}(\pi_1) = \text{labels}(\pi_2) = \text{labels}(\pi_3)$*, and (v) there is a path π_p from q_0 to q, (vi) there is a path π_s from q' to a state $q_r \notin F$.*

Proof. The necessity argument can be found in the appendix. The sufficiency argument is in the following text.

Fig. 6 illustrates the intuition behind the conditions above. The distinguishing characteristic of a super-linear NFA is that it contains two states q, q' such that q' is reachable from q on input string s, and it is possible to loop back from q and q' to the same state on string s. In addition, just like in Theorem 1, the pivot state q needs to be reachable from the initial state, and a rejecting state q_r must be reachable from q'. Observe that any automaton that is hyper-vulnerable according to Theorem 1 is also vulnerable according to Theorem 3. Specifically, consider an automaton $\mathcal{A}$ with two distinct paths π_1, π_2 that loop around q. In this case, if we take q' to be q and π_3 to be π_1, we immediately see that $\mathcal{A}$ also satisfies the conditions of Theorem 3.

To understand why the conditions of Theorem 3 imply super-linear time complexity, let us consider a string of the form $s_0 \cdot s^k \cdot s_1$ 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 $labels(\pi_1)$. Just like in the previous section, the path $\pi_p\, \pi_1^k\, \pi_s$ describes an execution for rejecting the string $s_0 \cdot s^k \cdot s_1$ in automaton $\mathcal{A}$. Now, let $T_q(k)$ represent the running time of rejecting the string $s^k s_1$ starting from q, and suppose that it takes 1 unit of time to read string s. We can write the following recurrence relation for $T_q(k)$:

$$T_q(k) = (1 + T_q(k-1)) + (1 + T_{q'}(k-1))$$

To understand where this recurrence is coming from, observe that there are two ways to process the first occurence of s:

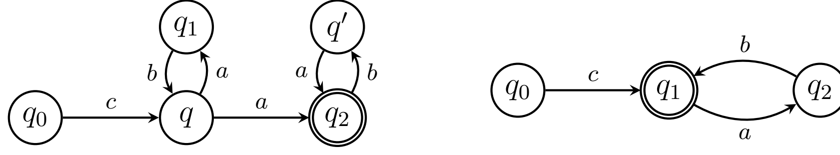

Fig. 7: A vulnerable NFA (left) and its attack automaton (right).

– Take path π_1 and come back to q, consuming 1 unit of time to process string s. Since we are back at q, we still have $T_q(k-1)$ units of work to perform.
– Take path π_2 and proceed to q', also consuming 1 unit of time to process string s. Since we are now at q', we have $T_{q'}(k-1)$ units of work to perform.

Now, observe that a lower bound on $T_{q'}(k)$ is k since one way to reach q_r is $\pi_3^k \pi_s$, which requires us to read the entire input string. This observation allows us to obtain the following recurrence relation:

$$T_q(k) \geq T_q(k-1) + k + 1$$

Thus, the running time of $\mathcal{A}$ on the input string $s_0 \cdot s^k \cdot s_1$ is at least k^2.

Example 2. The NFA shown in Fig. 7 (left) exhibits super-linear complexity because we can get from q to q' on input string ab, and for both q and q', we loop back to the same state when reading input string ab. Specifically, we have:

$$\pi_1 : (q, a, q_1), (q_1, b, q) \quad \pi_2 : (q, a, q_2), (q_2, b, q') \quad \pi_3 : (q', a, q_2), (q_2, b, q')$$

Furthermore, q is reachable from q_0, and there exists a rejecting state, namely q' itself, that is reachable from q'. The attack strings are of the form $c(ab)^k$, and Fig. 7 (right) shows the attack automaton.

5.2 Algorithm for Detecting Vulnerable NFAs

Based on the observations from the previous subsection, we can now formulate an algorithm that constructs an attack automaton $\mathcal{A}^\Psi$ for a given automaton $\mathcal{A}$. Just like in Algorithm 1, we construct an attack automaton $\mathcal{A}_i^\Psi$ for each state in $\mathcal{A}$ by invoking the AttackForPivot procedure. We then take the union of all such $\mathcal{A}_i^\Psi$'s to obtain an automaton $\mathcal{A}^\Psi$ whose language consists of strings that cause super-linear running time for $\mathcal{A}$.

Algorithm 2 describes the AttackForPivot procedure for the super-linear case. Just like in Algorithm 1, we consider all pairs of transitions from q with the same label (line 11). Furthermore, as in Algorithm 1, we construct an automaton $\mathcal{A}_p$ that recognizes attack prefixes for q (line 13) as well as an automaton $\mathcal{A}_1$ that recognizes non-empty strings that start and end at q (line 12).

The key difference of Algorithm 2 is that we also need to consider all states that could be instantiated as q' from Fig. 6 (lines 15–19). For each of these candidate q''s, we construct automata $\mathcal{A}_2, \mathcal{A}_3$ that correspond to paths π_2, π_3

Algorithm 2 Construct super-linear attack automaton $\mathcal{A}^{\Psi}$ for $\mathcal{A}$ and pivot q

```
 1: function AnyLoopBack(𝒜, q′)
 2:     assume 𝒜 = (Q, Σ, Δ, q₀, F)
 3:     q⋆ ← NewState(Q);   Q′ ← Q ∪ q⋆;   Δ′ ← Δ
 4:     for (q′, l, qᵢ) ∈ Δ do
 5:         Δ′ ← Δ′ ∪ (q⋆, l, qᵢ)
 6:     𝒜′ ← (Q′, Σ, Δ′, q⋆, {q′})
 7:     return 𝒜′
 8: function AttackForPivot(𝒜, q)
 9:     assume 𝒜 = (Q, Σ, Δ, q₀, F)
10:     𝒜^Ψ ← 𝒜^∅
11:     for (q, l, q₁) ∈ Δ ∧ (q, l, q₂) ∈ Δ ∧ q₁ ≠ q₂ do
12:         𝒜₁ ← LoopBack(𝒜, q, l, q₁)
13:         𝒜ₚ ← (Q, Σ, Δ, q₀, {q})
14:         for q′ ∈ Q do
15:             qᵢ ← NewState(Q)
16:             𝒜₂ ← (Q ∪ {qᵢ}, Σ, Δ ∪ {(qᵢ, l, q₂)}, qᵢ, {q′})
17:             𝒜₃ ← AnyLoopBack(𝒜, q′)
18:             𝒜ₛ ← (Q, Σ, Δ, q′, F)
19:             𝒜^Ψ ← 𝒜^Ψ ∪ (𝒜ₚ · (𝒜₁ ∩ 𝒜₂ ∩ 𝒜₃)⁺ · ‾𝒜ₛ)
20:     return 𝒜^Ψ
```

from Fig. 6 (lines 16–17). Specifically, we construct $\mathcal{A}_2$ by introducing a new initial state q_i with transition (q_i, l, q_2) and making its accepting state q'. Hence, $\mathcal{A}_2$ accepts strings that start in q, transition to q_2, and end in q'.

The construction of automaton $\mathcal{A}_3$, which should accept all non-empty words that start and end in q', is described in the AnyLoopBack procedure. First, since we do not want $\mathcal{A}_3$ to accept empty strings, we introduce a new initial state $q^\star$ and add a transition from $q^\star$ to all successor states q_i of q'. Second, the final state of $\mathcal{A}'$ is q' since we want to consider paths that loop back to q'.

The final missing piece of the algorithm is the construction of $\mathcal{A}_s$ (line 19), whose complement accepts all attack suffixes for state q'. As expected, $\mathcal{A}_s$ is the same as the original automaton $\mathcal{A}$, except that its initial state is q'. Finally, similar to Algorithm 1, the attack automaton for states q, q' is obtained as $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3)^+ \cdot \overline{\mathcal{A}_s}$.

Theorem 4. (Correctness of Algorithm 2) *Let NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ and $\mathcal{A}^{\Psi}$ be the result of calling* AttackAutomaton($\mathcal{A}$). *For every $s \in \mathcal{L}(\mathcal{A}^{\Psi})$, there exists a rejecting state $q_r \in Q \setminus F$ s.t. the number of distinct paths π_i from q_0 to q_r with $labels(\pi_i) = s$ is super-linear in the number of repetitions of the attack core in s.*

6 Dynamic Regular Expression Analysis

Algorithms 1 and 2 allow us to determine whether a given NFA is vulnerable. Even though our static analyses are sound and complete at the NFA level, differ-

ent regular expression matching algorithms construct NFAs in different ways and use different backtracking search algorithms. Furthermore, some matching algorithms may determinize the NFA (either lazily or eagerly) in order to guarantee linear complexity. Since our analysis does not perform such partial determinization of the NFA for a given regular expression, it can, in practice, generate false positives. In addition, even if a regular expression is indeed vulnerable, the input string must still exceed a certain minimum size to cause denial-of-service.

In order to overcome these challenges in practice, we also perform dynamic analysis to (a) confirm that a regular expression $\mathcal{E}$ is indeed vulnerable *for Java's matching algorithm*, and (b) infer a minimum bound on the size of the input string. Given the original regular expression $\mathcal{E}$, a user-provided time limit t, and the attack automaton $\mathcal{A}^{\Psi}$ (computed by static regular expression analysis), our dynamic analysis produces a refined attack automaton as well as a number b such that there exists an input string of length greater than b for which Java's matching algorithm takes more than t seconds. Note that, as usual, this dynamic analysis trades soundness for completeness to avoid too many false positives.

In more detail, given an attack automaton $\mathcal{A}^{\Psi}$ of the form $\mathcal{A}_p \cdot \mathcal{A}_c^+ \cdot \mathcal{A}_s$, the dynamic analysis finds the smallest k where the shortest string $s \in \mathcal{L}(\mathcal{A}_p \cdot \mathcal{A}_c^k \cdot \mathcal{A}_s)$ exceeds the time limit t. In practice, this process does not require more than a few iterations because we use the complexity of the NFA to predict the number of repetitions that should be necessary based on previous runs. The minimum required input length b is determined based on the length of the found string s. In addition, the value k is used to refine the attack automaton: in particular, given the original attack automaton $\mathcal{A}_p \cdot \mathcal{A}_c^+ \cdot \mathcal{A}_s$, the dynamic analysis refines it to be $\mathcal{A}_p \cdot \mathcal{A}_c^k \cdot \mathcal{A}_c^* \cdot \mathcal{A}_s$.

7 Static Program Analysis

As explained in Section 2, the presence of a vulnerable regular expression does not necessarily mean that the program is vulnerable. In particular, there are three necessary conditions for the program to contain a ReDoS vulnerability: First, a variable x that stores user input must be matched against a vulnerable regular expression $\mathcal{E}$. Second, it must be possible for x to store an attack string that triggers worst-case behavior for $\mathcal{E}$; and, third, the length of the string stored in x must exceed the minimum threshold determined using dynamic analysis.

To determine if the program actually contains a ReDoS vulnerability, our approach also performs static analysis of source code. Specifically, our program analysis employs the Cartesian product [7] of the following abstract domains:

- The *taint abstract domain* [6,26] tracks taint information for each variable. In particular, a variable is considered *tainted* if it may store user input.
- The *automaton abstract domain* [34,33,12] overapproximates the contents of string variables using finite automata. In particular, if string s is in the language of automaton $\mathcal{A}$ representing x's contents, then x *may* store string s.
- The *interval domain* [13] is used to reason about string lengths. Specifically, we introduce a ghost variable l_x representing the length of string x and use the interval abstract domain to infer upper and lower bounds for each l_x.

Since these abstract domains are fairly standard, we only explain how to use this information to detect ReDoS vulnerabilities. Consider a statement match$(x, \mathcal{E})$ that checks if string variable x matches regular expression $\mathcal{E}$, and suppose that the attack automaton for $\mathcal{E}$ is $\mathcal{A}^\Psi$. Now, our program analysis considers the statement match$(x, \mathcal{E})$ to be vulnerable if the following three conditions hold:

1. $\mathcal{E}$ is vulnerable and variable x is tainted;
2. The intersection of $\mathcal{A}^\Psi$ and the automaton abstraction of x is non-empty;
3. The upper bound on ghost variable l_x representing x's length exceeds the minimum bound b computed using dynamic analysis for $\mathcal{A}^\Psi$ and a user-provided time limit t.

Appendix D offers a more rigorous formalization of the analysis.

8 Experimental Evaluation

To assess the usefulness of the techniques presented in this paper, we performed an evaluation in which our goal is to answer the following questions:

Q1: Do real-world Java web applications use vulnerable regular expressions?
Q2: Can Rexploiter detect ReDoS vulnerabilities in web applications and how serious are these vulnerabilities?

Results for Q1. In order to assess if real-world Java programs contain vulnerabilities, we scraped the top 150 Java web applications (by number of stars) that contain at least one regular expression from GitHub repositories (all projects have between 10 and 2,000 stars and at least 50 commits) and collected a total of 2,864 regular expressions. In this pool of regular expressions, Rexploiter found 37 that have worst-case exponential complexity and 522 that have super-linear (but not exponential) complexity. Thus, we observe that approximately 20% of the regular expressions in the analyzed programs are vulnerable. We believe this statistic highlights the need for more tools like Rexploiter that can help programmers reason about the complexity of regular expression matching.

Results for Q2. To evaluate the effectiveness of Rexploiter in finding ReDoS vulnerabilities, we used Rexploiter to statically analyze all Java applications that contain at least one vulnerable regular expression. These programs include both web applications and frameworks, and cover a broad range of application domains. The average running time of Rexploiter is approximately 14 minutes per program, including the time to dynamically analyze regular expressions. The average size of analyzed programs is about 58,000 lines of code.

Our main result is that Rexploiter found exploitable vulnerabilities in 27 applications (including from popular projects, such as the Google Web Toolkit and Apache Wicket) and reported a total of 46 warnings. We manually inspected each warning and confirmed that 41 out of the 46 vulnerabilities are exploitable, with 5 of the exploitable vulnerabilities involving hyper-vulnerable regular expressions and the rest being super-linear ones. *Furthermore, for each of these*

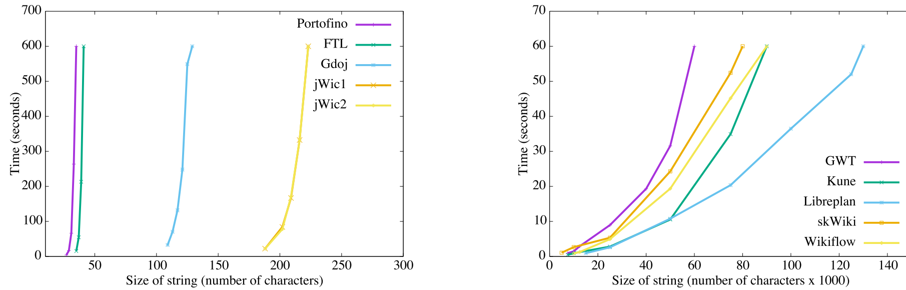

Fig. 8: Running times for exponential vulnerabilities (left) and super-linear vulnerabilities (right) for different input sizes.

41 vulnerabilities (including super-linear ones), we were able to come up with a full, end-to-end exploit that causes the server to hang for more than 10 minutes.

In Fig. 8, we explore a subset of the vulnerabilities uncovered by REXPLOITER in more detail. Specifically, Fig. 8 (left) plots input size against running time for the exponential vulnerabilities, and Fig. 8 (right) shows the same information for a subset of the super-linear vulnerabilities.

Possible fixes. We now briefly discuss some possible ways to fix the vulnerabilities uncovered by REXPLOITER. The most direct fix is to rewrite the regular expression so that it no longer exhibits super-linear complexity. Alternatively, the problem can also be fixed by ensuring that the user input cannot contain instances of the attack core. Since our technique provides the full attack automaton, we believe REXPLOITER can be helpful for implementing suitable sanitizers. Another possible fix (which typically only works for super-linear regular expressions) is to bound input size. However, for most vulnerabilities found by REXPLOITER, the input string can legitimately be very large (e.g., review). Hence, there may not be an obvious upper bound, or the bound may still be too large to prevent a ReDoS attack. For example, Amazon imposes an upper bound of 5000 words (∼25,000 characters) on product reviews, but matching a super-linear regular expression against a string of that size may still take significant time.

9 Related Work

To the best of our knowledge, we are the first to present an end-to-end solution for detecting ReDoS vulnerabilities by combining regular expression and program analysis. However, there is prior work on static analysis of regular expressions and, separately, on program analysis for finding security vulnerabilities.

Static analysis of regular expressions. Since vulnerable regular expressions are known to be a significant problem, previous work has studied static analysis techniques for identifying regular expressions with worst-case exponential complexity [9,18,22,24]. Recent work by Weideman et al. [30] has also proposed an analysis for identifying super-linear regular expressions. However, no previous technique can construct attack automata that capture all malicious strings. Since

attack automata are crucial for reasoning about sanitization, the algorithms we propose in this paper are necessary for performing sanitization-aware program analysis. Furthermore, we believe that the attack automata produced by our tool can help programmers write suitable sanitizers (especially in cases where the regular expression is difficult to rewrite).

Program analysis for vulnerability detection. There is a large body of work on statically detecting security vulnerabilities in programs. Many of these techniques focus on detecting cross-site scripting (XSS) or code injection vulnerabilities [8,11,12,15,17,19,20,23,27,28,29,32,33,34,35]. There has also been recent work on static detection of specific classes of denial-of-service vulnerabilities. For instance, Chang et al. [10] and Huang et al. [16] statically detect attacker-controlled loop bounds, and Olivo et al. [21] detect so-called *second-order DoS vulnerabilities*, in which the size of a database query result is controlled by the attacker. However, as far as we know, there is no prior work that uses program analysis for detecting DoS vulnerabilities due to regular expression matching.

Time-outs to prevent ReDoS. As mentioned earlier, some libraries (e.g., the .NET framework) allow developers to set a time-limit for regular expression matching. While such libraries may help *mitigate* the problem through a band-aid solution, they do not address the root cause of the problem. For instance, they neither prevent against stack overflows nor do they prevent DoS attacks in which the attacker triggers the regular expression matcher many times.

10 Conclusions and Future Work

We have presented an end-to-end solution for statically detecting regular expression denial-of-service vulnerabilities in programs. Our key idea is to combine complexity analysis of regular expressions with safety analysis of programs. Specifically, our regular expression analysis constructs an attack automaton that recognizes all strings that trigger worst-case super-linear or exponential behavior. The program analysis component takes this information as input and performs a combination of taint and string analysis to determine whether an attack string could be matched against a vulnerable regular expression.

We have used our tool to analyze thousands of regular expressions in the wild and we show that 20% of regular expressions in the analyzed programs are actually vulnerable. We also use REXPLOITER to analyze Java web applications collected from Github repositories and find 41 exploitable security vulnerabilities in 27 applications. Each of these vulnerabilities can be exploited to make the web server unresponsive for more than 10 minutes.

There are two main directions that we would like to explore in future work: First, we are interested in the problem of automatically *repairing* vulnerable regular expressions. Since it is often difficult for humans to reason about the complexity of regular expression matching, we believe there is a real need for techniques that can automatically synthesize equivalent regular expressions with linear complexity. Second, we also plan to investigate the problem of automat-

ically generating sanitizers from the attack automata produced by our regular expression analysis.

References

1. CVE-2013-2009. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2099
2. CVE-2015-2525. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2526
3. CVE-2015-2525. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3275
4. CVE-2016-2515. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2515
5. CVE-2016-2537. cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2537
6. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: PLDI. pp. 259–269. ACM (2014)
7. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstraction for model checking c programs. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 268–283. Springer (2001)
8. Bandhakavi, S., Tiku, N., Pittman, W., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with VEX. Commun. ACM 54(9), 91–99 (2011)
9. Berglund, M., Drewes, F., van der Merwe, B.: Analyzing catastrophic backtracking behavior in practical regular expression matching. In: AFL. EPTCS, vol. 151, pp. 109–123 (2014)
10. Chang, R.M., Jiang, G., Ivancic, F., Sankaranarayanan, S., Shmatikov, V.: Inputs of coma: Static detection of denial-of-service vulnerabilities. In: CSF. pp. 186–199. IEEE Computer Society (2009)
11. Chaudhuri, A., Foster, J.S.: Symbolic security analysis of ruby-on-rails web applications. In: CCS. pp. 585–594. ACM (2010)
12. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: SAS. LNCS, vol. 2694, pp. 1–18. Springer (2003)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
14. Crosby, S.A., Wallach, D.S.: Denial of service via algorithmic complexity attacks. In: USENIX Security Symposium. USENIX Association (2003)
15. Dahse, J., Holz, T.: Static detection of second-order vulnerabilities in web applications. In: USENIX Security Symposium. pp. 989–1003. USENIX Association (2014)
16. Huang, H., Zhu, S., Chen, K., Liu, P.: From system services freezing to system server shutdown in android: All you need is a loop in an app. In: CCS. pp. 1236–1247. ACM (2015)
17. Kiezun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: ICSE. pp. 199–209. IEEE (2009)
18. Kirrage, J., Rathnayake, A., Thielecke, H.: Static analysis for regular expression denial-of-service attacks. In: NSS. LNCS, vol. 7873, pp. 135–148. Springer (2013)
19. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security Symposium. USENIX Association (2005)
20. Martin, M.C., Livshits, V.B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPSLA. pp. 365–383. ACM (2005)

21. Olivo, O., Dillig, I., Lin, C.: Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In: CCS. pp. 616–628. ACM (2015)
22. Rathnayake, A., Thielecke, H.: Static analysis for regular expression exponential runtime via substructural logics. CoRR abs/1405.7058 (2014)
23. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: POPL. pp. 372–382. ACM (2006)
24. Sugiyama, S., Minamide, Y.: Checking time linearity of regular expression matching based on backtracking. IPSJ Online Transactions 7, 82–92 (2014)
25. Thompson, K.: Programming techniques: Regular expression search algorithm. Communications of the ACM 11(6), 419–422 (1968)
26. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: TAJ: effective taint analysis of web applications. In: PLDI. pp. 87–97. ACM (2009)
27. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI. pp. 32–41. ACM (2007)
28. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: ICSE. pp. 171–180. ACM (2008)
29. Wassermann, G., Yu, D., Chander, A., Dhurjati, D., Inamura, H., Su, Z.: Dynamic test input generation for web applications. In: ISSTA. pp. 249–260. ACM (2008)
30. Weideman, N., van Der Merwe, B., Berglund, M., Watson, B.: Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In: CIAA (2016), to appear
31. Wüstholz, V., Olivo, O., Heule, M.J.H., Dillig, I.: Static detection of dos vulnerabilities in programs that use regular expressions (extended version). CoRR abs/1109.6926 (2017)
32. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium. USENIX Association (2006)
33. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: TACAS. LNCS, vol. 6015, pp. 154–157. Springer (2010)
34. Yu, F., Alkhalaf, M., Bultan, T., Ibarra, O.H.: Automata-based symbolic string analysis for vulnerability detection. FMSD 44(1), 44–70 (2014)
35. Yu, F., Bultan, T., Hardekopf, B.: String abstractions for string verification. In: SPIN. LNCS, vol. 6823, pp. 20–37. Springer (2011)

Appendix A: Proof about Algorithm 1

Proof. (sketch) We first show that any attack string s is accepted by $\mathcal{A}^\Psi$. Based on Section 4, we know that attack strings that cause exponential behavior are of the form $s_0 \cdot s_c^k \cdot s_1$ where $s_0 = \mathit{labels}(\pi_p)$, $s_c = \mathit{labels}(\pi_1) = \mathit{labels}(\pi_2)$, $s_1 = \mathit{labels}(\pi_s)$ for some pivot state q. Now, we argue that s will be accepted by the attack automaton $\mathcal{A}_q^\Psi$ for q, which implies that s is also accepted by $\mathcal{A}^\Psi$ since $\mathcal{A}_q^\Psi \subseteq \mathcal{A}^\Psi$. Since AttackForPivot is invoked for each state q, we will consider the two distinct transitions (q, l, q_1) and (q, l, q_2) that start paths π_1 and π_2. Furthermore, by the construction in the LoopBack procedure, $\mathit{labels}(\pi_1)$ and $\mathit{labels}(\pi_2)$ will be accepted by $\mathcal{A}_1$ and $\mathcal{A}_2$. Thus, string s_c will be accepted by $(\mathcal{A}_1 \cap \mathcal{A}_2)$. Similarly, by the construction at lines 14–15, $\mathcal{A}_p$ and $\overline{\mathcal{A}_s}$ will accept s_0 and s_1 respectively. Hence, the attack string $s = s_0 s_c^k s_1$ will be recognized by $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2)^+ \cdot \overline{\mathcal{A}_s}$.

For the other direction, we show that if a string s is accepted by $\mathcal{A}^{\Psi}$, then it is indeed an attack string. The attack automaton constructed by the algorithm is a union of automata $\mathcal{A}^{\Psi}_{q}$ of the form $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2)^+ \cdot \overline{\mathcal{A}_s}$, one of which must accept the string s. As a consequence, there must exist strings $s_0 \in \mathcal{A}_p$, $s_c^k \in (\mathcal{A}_1 \cap \mathcal{A}_2)^+$, and $s_1 \in \overline{\mathcal{A}_s}$ such that $s = s_0 \cdot s_c^k \cdot s_1$. Due to the construction of $\mathcal{A}^{\Psi}_{q}$ in function AttackForPivot, there must exist corresponding paths π_1, π_2 (distinct from π_1), π_p, and π_s such that $s_0 = \mathit{labels}(\pi_p)$, $s_c = \mathit{labels}(\pi_1) = \mathit{labels}(\pi_2)$, and $s_1 = \mathit{labels}(\pi_s)$. Based on Section 4, any such string s constitutes an attack string.

Appendix B: Proof about Algorithm 2

Proof. (sketch) We first show that any attack string s is accepted by $\mathcal{A}^{\Psi}$. Based on Section 5.1, we know that attack strings that cause super-linear behavior are of the form $s_0 \cdot s_c^k \cdot s_1$ where $s_0 = \mathit{labels}(\pi_p)$, $s_c = \mathit{labels}(\pi_1) = \mathit{labels}(\pi_2) = \mathit{labels}(\pi_3)$, $s_1 = \mathit{labels}(\pi_s)$ for some pivot state q and a state q'. Now, we argue that s will be accepted by the attack automaton $\mathcal{A}^{\Psi}_{q}$ for q, which implies that s is also accepted by $\mathcal{A}^{\Psi}$ since $\mathcal{A}^{\Psi}_{q} \subseteq \mathcal{A}^{\Psi}$. Since AttackForPivot is invoked for each state q, we will consider the two distinct transitions (q, l, q_1) and (q, l, q_2) that start paths π_1 and π_2 and any state q'. Furthermore, by the construction in the LoopBack procedure, $\mathit{labels}(\pi_1)$ will be accepted by $\mathcal{A}_1$. By the construction on lines 15–16, $\mathit{labels}(\pi_2)$ will be accepted by $\mathcal{A}_2$. By the construction in the AnyLoopBack procedure, $\mathit{labels}(\pi_3)$ will be accepted by $\mathcal{A}_3$. Thus, string s_c will be accepted by $(\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3)$. Similarly, by the construction at lines 13 and 18, $\mathcal{A}_p$ and $\overline{\mathcal{A}_s}$ will accept s_0 and s_1 respectively. Hence, the attack string $s = s_0 s_c^k s_1$ will be recognized by $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3)^+ \cdot \overline{\mathcal{A}_s}$.

For the other direction, we show that if a string s is accepted by $\mathcal{A}^{\Psi}$, then it is indeed an attack string. The attack automaton constructed by the algorithm is a union of automata $\mathcal{A}^{\Psi}_{q}$ of the form $\mathcal{A}_p \cdot (\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3)^+ \cdot \overline{\mathcal{A}_s}$, one of which must accept the string s. As a consequence, there must exist strings $s_0 \in \mathcal{A}_p$, $s_c^k \in (\mathcal{A}_1 \cap \mathcal{A}_2 \cap \mathcal{A}_3)^+$, and $s_1 \in \overline{\mathcal{A}_s}$ such that $s = s_0 \cdot s_c^k \cdot s_1$. Due to the construction of $\mathcal{A}^{\Psi}_{q}$ in function AttackForPivot, there must exist corresponding paths π_1, π_2 (distinct from π_1), π_3, π_p, and π_s such that $s_0 = \mathit{labels}(\pi_p)$, $s_c = \mathit{labels}(\pi_1) = \mathit{labels}(\pi_2) = \mathit{labels}(\pi_3)$, and $s_1 = \mathit{labels}(\pi_s)$. Based on Section 5.1 any such string s constitutes an attack string.

Appendix C: Necessity Proofs (Theorems 1 and 3)

This section shows that the conditions in Theorems 1 and 3 are not only sufficient, but also necessary for the NFA to exhibit exponential and super-linear complexity respectively. In the rest of this section, we use the term *vulnerable NFA* to mean an NFA that satisfies the conditions of Theorem 1 and *hyper-vulnerable NFA* to mean an NFA satisfying conditions of Theorem 3.

Our proof uses the concept of *strongly-connected component pair path* (SPP). Given an NFA with $|Q|$ states, there are at most $\mathcal{O}(|Q|^{|Q|})$ such SPPs. Given

a string s, if the NFA is not hyper-vulnerable, then there are at most $|s|^{|Q|}$ possible matchings per SPP. The complexity of a worst-case backtracking search algorithm is thus polynomial in length of s: $\mathcal{O}(|Q|^{|Q|}|s|^{|Q|})$.

If the NFA is not vulnerable, then there are at most $|Q|^{|Q|}$ possible matchings per SPP. A backtracking search algorithm could match all possible substrings of s adding a of factor $|s|$ and a factor of $|\Sigma|^{|Q|}$, resulting in the complexity: $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|}|s|)$.

Definition 5. (Strongly-Connected Component) *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. Two states $q, q' \in Q$ with $q \neq q'$ are in the same strongly-connected component if and only if there exist a path from q to q' and from q' to q. A partition of Q in strongly connected components is unique.*

Lemma 1. *Given a non-hyper-vulnerable NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, a string s, and two states $q, q' \in Q$ occurring in the same strongly-connected component. The path π from q to q' such that* $\text{labels}(\pi) = s$ *is unique.*

Proof. Assume that there are two paths π_1 and π_2 from q to q' such that $\pi_1 \neq \pi_2$ and $\mathit{labels}(\pi_1) = \mathit{labels}(\pi_2) = s$. Since q and q' occur in the same strongly-connected component, there must be a path π_3 from q' to q, because in a strongly-connected component there exists a path from every state to every state. Now we have two cycles from q to q, $\pi_1\pi_3$ and $\pi_2\pi_3$ such that $\mathit{labels}(\pi_1\pi_3) = \mathit{labels}(\pi_2\pi_3)$. This violates the assumption that $\mathcal{A}$ is not hyper-vulnerable.

Notice that Lemma 1 also holds for non-vulnerable NFAs as each non-vulnerable NFA is also non-hyper-vulnerable.

Definition 6. (Strongly-Connected Component Pair Path) *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. A strongly-connected component pair path (SPP) of $\mathcal{A}$ is a sequence of state pairs $(q_{\text{in}}, q_{\text{out}})$, with q_{in} and q_{out} occurring in the same strongly-connected component. Moreover, for any two consecutive (i.e., occurring in different, but connected strongly-connected components) state pairs $(q_{\text{in}}, q_{\text{out}})$ and $(q'_{\text{in}}, q'_{\text{out}})$ there must exist a transition $(q_{\text{out}}, l, q'_{\text{in}}) \in \Delta$ for some label l.*

Lemma 2. *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. There are at most $\mathcal{O}(|Q|^{|Q|})$ different strongly-connected component pair paths.*

Proof. $\mathcal{A}$ has at most $|Q|$ strongly-connected components. Consequently, a SPP consists of at most $|Q|$ pairs. Each strongly-connected component consists of at most $|Q|$ states. Hence there are at most $|Q|^2$ different state pairs per strongly-connected components. The number of SPPs for $\mathcal{A}$ is thus at most $|Q|^{2|Q|}$ or $\mathcal{O}(|Q|^{|Q|})$.

Definition 7. (Path Partition) *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, a string s and a strongly-connected component pair path Π of $\mathcal{A}$. A* path partition *of s and Π is a partition of s into $2|\Pi| - 1$ substrings s_i with $i \in \{0, \ldots, 2|\Pi| - 2\}$ in such a way that each s_i with odd i consists of exactly one symbol. The substrings s_i with i even can be arbitrarily long (or short, even empty). Notice that the choice of the s_i with odd i define the s_i with the even i: Let i be odd, s_{i+1} are all the symbols in s that occur between s_i and s_{i+2}.*

Lemma 3. *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, a string s and a strongly-connected component pair path Π of $\mathcal{A}$. There exist at most $|s|^{|Q|}$ different path partitions of s and Π.*

Proof. There exists at most $\binom{|s|}{|\Pi|-1}$ different path partitions of s and Π, i.e., all possible $|\Pi| - 1$ choices of s_i's with odd i. Notice that $|\Pi| - 1 < |Q|$, because a SPP has at most length $|Q|$. Therefore the number of different path partitions is less than $|s|^{|Q|}$.

Theorem 5. *Let $\mathcal{A}$ be the NFA $(Q, \Sigma, \Delta, q_0, F)$, which is not hyper-vulnerable. The runtime to determine if a string s is accepted by $\mathcal{A}$ is at most $\mathcal{O}(|Q|^{|Q|}|s|^{|Q|})$.*

Proof. Below we assume that s consists of at least $|Q|$ symbols. In case $|s| < |Q|$, then the number of steps is limited by $|Q|^{|s|}$ and thus $|Q|^{|Q|}$ even for hyper-vulnerable NFAs: From each state we could potentially go to each other state and repeat that $|s|$ times.

From Lemma 1 we know that there is a path from state q to q' with q and q' occurring in the same strongly connected component is unique. Given path partition P of s and Π, we can now deduce that the path of s from q_0 to the last q_{out} in Π is unique: each s_i with even i uniquely force the path within a strongly-connected component, while each s_i with odd i uniquely define the path in between strongly-connected components.

Furthermore, from Lemma 3, we know that the number of part partitions is at most $|s|^{|Q|}$.

Consequently, a backtracking search algorithm for s will require at most $\mathcal{O}(|Q|^{|Q|}|s|^{|Q|})$ steps. Notice that we did not discuss that a backtrack search algorithm also matches strings that are the first n symbols of s. This adds another factor of $|s|$, which can be ignored given the above complexity result.

Definition 8. (Labelled Strongly-Connected Component Pair Path) *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. A labelled strongly-connected component pair path (LSPP) of $\mathcal{A}$ is a strongly-connected component pair path with has a specific label l in between two consecutive state pairs $(q_{\text{in}}, q_{\text{out}})$ and $(q'_{\text{in}}, q'_{\text{out}})$ that describes the transition from q_{out} to q'_{in}.*

Lemma 4. *Given an NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. There are at most $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|})$ different labelled strongly-connected component pair paths.*

Proof. Let Π be a strongly-connected component pair path. There are at most $\mathcal{O}(|Q|^{|Q|})$ strongly-connected component pair paths (Lemma 2). There are $|\Pi|-1$ consecutive state pairs in Π. For each of them a label $l \in \Sigma$ can be selected. Hence there are $|\Sigma|^{|\Pi|-1}$ different labelled strongly-connected component pair path that have the same state pairs as Π. This results in $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|})$ different LSSPs.

A path partition P of string s and strongly-connected component pair path Π is called *valid*, if there exists a path π that follows the states described in Π such that $labels(\pi) = s$. Lemma 3 states that for a non-hyper-vulnerable NFA

there are at most $|s|^{|Q|}$ path partitions. All of them could be valid. However, below we will show that for a non-vulnerable NFA that the number of valid path partitions is at most $|Q|^{|Q|}$, thereby removing the factor $|s|^{|Q|}$ from the complexity.

Lemma 5. *Given a non-vulnerable NFA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, a string s, and a labelled strongly-connected component pair path Π. Let (q_{in}, q_{out}) and $(q'_{\text{in}}, q'_{\text{out}})$ be two consecutive state pairs of Π and let l be the label in between the state pairs: $(q_{\text{out}}, l, q'_{\text{in}}) \in \Delta$. There are at most $|Q|$ possible choices of l in s, such that there exists a valid path partition P of s and Π.*

Proof. (sketch) Let q_{end} be the last state in Π, i.e., the second state in the last state pair. Let Q_0 be all the states for which a path exists to q_{out} including q_{out} and let Q_1 be all states for which a path exists starting from q'_{in} including q'_{in}. Notice that the intersection of Q_0 and Q_1 is empty.

Consider $|Q_0|$ different choices of l in s such that exists a path from q_0 to q_{out} following the states described in Π and a path from q'_{in} to q_{end} following the states described in Π. On any such path that can be at most $|Q_0| - 1$ different l transitions. Hence there must exists a chosen transition (q, l, q') such that the path from q_0 to q_{out} uses that transition at least twice. For this q, there exists a cycle π_1 from q to q and a path π_2 starting at q that includes the transition $(q_{\text{out}}, l, q'_{\text{in}})$ such that $labels(\pi_1) = labels(\pi_2)$.

We can apply the same reason for Q_1: Consider $|Q_1|$ different picks of l in s such that exists a path from q_0 to q_{out} following the states described in Π and a path from q'_{in} to q_{end} following the states described in Π. On any such path that can be at most $|Q_1| - 1$ different l transitions. Hence there must exists a transition (q'', l, q''') such that the path from q'_{in} to q_{end} uses that transition at least twice. For this q'', there exists a cycle π_3 from q'' to q'' and a path π_4 ending at q'' that includes the transition $(q_{\text{out}}, l, q'_{\text{in}})$ such that $labels(\pi_3) = labels(\pi_4)$.

Let π_5 be a path from q to q''. Now we can change the cycles π_1 and π_3 to π'_1 and π'_3 by extending them with loops such that $labels(\pi'_1) = labels(\pi'_3) = labels(\pi_5)$. The existence of such paths is in conflict with the assumption that $\mathcal{A}$ is not vulnerable. Consequently, there must be less than $|Q_0| + |Q_1|$ choices for l. Since the intersection of Q_0 and Q_1 is empty, $|Q_0| + |Q_1| \leq |Q|$.

Theorem 6. *Let $\mathcal{A}$ be the NFA $(Q, \Sigma, \Delta, q_0, F)$, which is not vulnerable. The runtime to determine if a string s is accepted by $\mathcal{A}$ is at most $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|}|s|)$.*

Proof. From Lemma 4 we know that there are at most $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|})$ labelled strongly-connected component pair path of s. Let Π be one of these LSPPs. There are $|\Pi| - 1$ labels between consecutive state pairs. For each of them there are at lost $|Q|$ choices from s, such that the path partition is valid (Lemma 5). Consequently, there are at most $|Q|^{|\Pi|-1}$ valid path partitions for Π. Since $|\Pi| - 1 < |Q|$, the number of valid path partitions is less than $|Q|^{|Q|}$. Hence we can ignore the number of valid path partitions in the complexity, because it does not alter $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|})$.

Given a valid path partition, a backtrack search algorithm may take $|s|$ steps from q_0 to the last state. This adds a factor of $|s|$ to the complexity resulting in $\mathcal{O}(|Q|^{|Q|}|\Sigma|^{|Q|}|s|)$.

Appendix D: Formal Analysis

$$
\begin{array}{lll}
\text{Statement } S & := & x := e \mid \text{getInput}(x) \\
 & & \mid \text{match}(x, \mathcal{E}) \mid S_1; S_2 \\
 & & \mid \text{assume}(x \in \mathcal{R}) \\
 & & \mid \text{assume}(\mathit{len}(x) \leq \nu) \\
 & & \mid \text{if}(\star) \text{ then } S_1 \text{ else } S_2 \\
 & & \mid \text{while}(\star) \text{ do } S \\
 & & \\
\text{String exp } e & := & x \mid ? \\
\text{Int exp } \nu & := & \text{int} \mid \mathit{len}(x) \mid \nu_1 + \nu_2 \mid \nu_1 - \nu_2 \\
\text{Pure regex } \mathcal{E} & := & a \in \Sigma \mid \mathcal{E}^* \mid \mathcal{E}_1 + \mathcal{E}_2 \mid \mathcal{E}_1 \cdot \mathcal{E}_2 \\
\text{Impure regex } \mathcal{R} & := & \mathcal{E} \mid x \mid \mathcal{R}^* \mid \mathcal{R}_1 + \mathcal{R}_2 \mid \mathcal{R}_1 \cdot \mathcal{R}_2
\end{array}
$$

Fig. 9: The STRIMP intermediate language

Intermediate language. We formalize our program analysis using the intermediate language shown in Fig. 9. This language, which we refer to as STRIMP, is suitable for describing our analysis because it models the effects of different string manipulation functions in a uniform way using *assume* statements.

In STRIMP, all variables have type string. In addition to the standard assignment, sequence, conditional, and loop constructs, STRIMP contains a function *getInput(x)*, which binds variable x to a string supplied by the user. Another function, *match*$(x, \mathcal{E})$, models matching string x against regular expression $\mathcal{E}$.

The STRIMP language contains two kinds of *assume* statements that allow us to model the effect of string manipulation procedures (e.g., provided by `java.lang.String`). First, the statement *assume*$(x \in \mathcal{R})$ states that x belongs to the language given by *impure regular expression* $\mathcal{R}$. Here, we refer to $\mathcal{R}$ as *impure* because the regular expression can refer to program variables. For example, the statement *assume*$(x \in y \cdot a)$ models that the value stored in x is the value stored in y concatanated with the character a. Thus, if y can be any string, then this annotation expresses that x is a string ending in a. The use of such impure regular expressions in STRIMP allows us to model string operations in a uniform way.

The second form of annotation in STRIMP is of the form *assume*$(\mathit{len}(x) \leq \nu)$ and allows us to express constraints on the size of strings. Here, the integer expression ν can refer to the length of other strings and can contain arithmetic operators $(+, -)$.

One final point worth noting is that string expressions include a special symbol ?, which represents an unknown string constant. Hence, an assignment of

$$
\begin{aligned}
[\![\mathcal{E}]\!]_\Lambda &= \mathcal{A}(\mathcal{E}) \\
[\![x]\!]_\Lambda &= snd(\Lambda(x)) \\
[\![\mathcal{R}^*]\!]_\Lambda &= ([\![\mathcal{R}]\!]_\Lambda)^* \\
[\![\mathcal{R}_1\mathcal{R}_2]\!]_\Lambda &= [\![\mathcal{R}_1]\!]_\Lambda \cdot [\![\mathcal{R}_2]\!]_\Lambda \\
[\![\mathcal{R}_1 + \mathcal{R}_2]\!]_\Lambda &= [\![\mathcal{R}_1]\!]_\Lambda + [\![\mathcal{R}_2]\!]_\Lambda
\end{aligned}
$$

Fig. 10: Helper rules for evaluating impure regular expressions. We use $\mathcal{A}(\mathcal{E})$ to denote an NFA that accepts the same language as regular expression $\mathcal{E}$.

$$
\begin{aligned}
[\![\text{int}]\!]_\Lambda &= \langle \text{int}, \text{int} \rangle \\
[\![len(x)]\!]_\Lambda &= fst(\Lambda(x)) \\
[\![\nu_1 + \nu_2]\!]_\Lambda &= [\![\nu_1]\!]_\Lambda \oplus [\![\nu_2]\!]_\Lambda \\
[\![\nu_1 - \nu_2]\!]_\Lambda &= [\![\nu_1]\!]_\Lambda \ominus [\![\nu_2]\!]_\Lambda
\end{aligned}
$$

Fig. 11: Helper rules for evaluating arithmetic expressions

the form `x = "abc"` is easily expressible in our language using the code snippet:

$$x :=?;\ \text{assume}(x \in abc)$$

Program abstraction. As mentioned earlier, our program analysis needs to track taint information as well as information about string lengths and contents. Towards this goal, our analysis employs three kinds of program abstractions:

- The *taint abstraction* Φ is a set of variables such that $x \in \Phi$ indicates that x may be tainted.
- The *string abstraction* Λ is a mapping from each program variable x to a pair $(\mathbb{I}, \mathcal{A})$, where $\mathbb{I}$ is an interval $\langle l, u \rangle$ such that $l \leq len(s) \leq u$ and $\mathcal{A}$ is an NFA representing x's contents. In particular, if a string s is in the language of $\mathcal{A}$, this indicates that x can store string s.
- The *attack abstraction* Ψ maps each regular expression $\mathcal{E}$ in the program to a pair $(b, \mathcal{A}^\Psi)$. Here, b is a minimum bound on the length of the input string s such that, if $len(s) < b$, matching s against $\mathcal{E}$ takes negligible time [3]. The NFA $\mathcal{A}^\Psi$ is the attack automaton for regular expression $\mathcal{E}$ and is pre-computed using the analyses from Sections 4 and 5.

Since our program abstractions involve pairs (e.g., $(\mathbb{I}, \mathcal{A})$) , we use the notation *fst(p)* and *snd(p)* to retrieve the first and second components of pair p respectively.

Analysis rules. We describe our static analysis using judgments of the form $\Psi, \Phi, \Lambda \vdash S : \Phi', \Lambda'$ which state that, if we execute S in a state that satisfies program abstractions Ψ, Φ, Λ, we obtain a new taint abstraction Φ' and new string abstraction Λ'. The inference rules describing our analysis are shown in Fig. 13.

[3] Here, what constitutes *negligible time* is an input parameter of our analysis and can be customized by the user.

$$
\begin{aligned}
\langle l_1, u_1 \rangle \oplus \langle l_2, u_2 \rangle &= \langle l_1 + l_2, u_1 + u_2 \rangle \\
\langle l_1, u_1 \rangle \ominus \langle l_2, u_2 \rangle &= \langle l_1 - l_2, u_1 - u_2 \rangle \\
\langle l_1, u_1 \rangle \sqcup \langle l_2, u_2 \rangle &= \langle min(l_1, l_2), max(u_1, u_2) \rangle \\
(\Lambda_1 \sqcup \Lambda_2)(x) &= (fst(\Lambda_1(x)) \sqcup fst(\Lambda_2(x)), \\
&\quad snd(\Lambda_1(x)) \cup snd(\Lambda_2(x)))
\end{aligned}
$$

Fig. 12: Operations on abstract domains

In this figure, rule (1) describes the analysis of sources (i.e., $getInput(x)$). Since variable x is now tainted, we add it to our taint abstraction Φ. Furthermore, since the user is free to supply any string, Λ' abstracts the length of x using the interval $\langle 0, \infty \rangle$ and its contents using the automaton $\mathcal{A}^*$, which accepts any string.

Rule (2) for processing assignments $x_1 := x_2$ is straightforward: In particular, x_1 becomes tainted iff x_2 is tainted, and the string abstraction of x_1 is the same as variable x_2. For assignments of the form $x :=?$ (rule 3), we untaint variable x by removing it from Φ since ? denotes string constants in Strimp. However, since ? represents unknown strings, Λ' maps x to $(\langle 0, \infty \rangle, \mathcal{A}^*)$.

Rule (4) describes the analysis of assumptions of the form $assume(x \in \mathcal{R})$. Because $\mathcal{R}$ can refer to program variables, we must first figure out the regular expressions that are represented by $\mathcal{R}$. For this purpose, Fig. 10 describes the evaluation of impure regular expression $\mathcal{R}$ under string abstraction Λ, denoted as $[\![\mathcal{R}]\!]_\Lambda$. Since the assumption states that the value stored in x must be in the language $[\![\mathcal{R}]\!]_\Lambda$, the new string abstraction Λ' maps x to the automaton $snd(\Lambda(x)) \cap [\![\mathcal{R}]\!]_\Lambda$. [4]

Rule (5), which is quite similar to rule (4), allows us to handle assumptions of the form $assume(len(x) \leq \nu)$. Since integer expression ν can refer to terms of the form $len(y)$, we must evaluate ν under string abstraction Λ. For this purpose, Fig. 11 shows the evaluation of ν under Λ, denoted as $[\![\nu]\!]_\Lambda$. Now, going back to rule (5) of Fig. 13, suppose that $[\![\nu]\!]_\Lambda$ yields the interval $\langle l_2, u_2 \rangle$, and suppose that Λ maps x to the length abstraction $\langle l_1, u_1 \rangle$. Clearly, the assumption $assume(len(x) \leq \nu)$ does not change the lower bound on $len(x)$; hence the lower bound remains l_1. However, if u_2 is less than the previous upper bound u_1, we now have a more precise upper-bound u_1. Hence, the new string abstraction Λ' maps the length component of x to the interval $\langle l_1, min(u_1, u_2) \rangle$.

Rule (6) for *match* statements allows us to detect if the program contains a vulnerability. In particular, the premise of this rule states that either (1) x is *not* tainted ($x \notin \Phi$) or (2) the automaton representing x's contents does not contain any string in the attack automaton for $\mathcal{E}$ (i.e., $\mathcal{A} = \mathcal{A}^\emptyset$), or (3) the length of x cannot exceed the minimum bound given by Ψ (i.e., $fst(\Psi(\mathcal{E})) \notin fst(\Lambda(x))$). If

[4] Observe that Rule (4) does not modify the length abstraction component of Λ. This is clearly sound, but potentially imprecise. However, since we model Java string operations by adding a pair of assumptions, one concerning length and the other concerning content, our analysis does not lead to a loss of precision because of the way assumptions are introduced.

$$\frac{\Phi' = \Phi \cup \{x\} \quad \Lambda' = \Lambda[x \mapsto (\langle 0, \infty \rangle, \mathcal{A}^*)]}{\Psi, \Phi, \Lambda \vdash \text{getInput}(x) : \Phi', \Lambda'} \quad (1)$$

$$\frac{\Lambda' = \Lambda[x_1 \mapsto \Lambda(x_2)] \quad \Phi' = \begin{cases} \Phi \cup \{x_1\} & \text{if } x_2 \in \Phi \\ \Phi & \text{if } x_2 \notin \Phi \end{cases}}{\Psi, \Phi, \Lambda \vdash x_1 := x_2 : \Phi', \Lambda'} \quad (2)$$

$$\frac{\Lambda' = \Lambda[x \mapsto (\langle 0, \infty \rangle, \mathcal{A}^*)]}{\Psi, \Phi, \Lambda \vdash x := \ ? : \Phi \setminus \{x\}, \Lambda'} \quad (3)$$

$$\frac{\mathcal{A} = \mathit{snd}(\Lambda(x)) \cap [\![\mathcal{R}]\!]_\Lambda \quad \Lambda' = \Lambda[x \mapsto (\mathit{fst}(\Lambda(x)), \mathcal{A})]}{\Psi, \Phi, \Lambda \vdash \text{assume}(x \in \mathcal{R}) : \Phi, \Lambda'} \quad (4)$$

$$\frac{\langle l_1, u_1 \rangle = \mathit{fst}(\Lambda(x)) \quad \langle l_2, u_2 \rangle = [\![\nu]\!]_\Lambda \quad \mathbb{I} = \langle l_1, \mathit{min}(u_1, u_2) \rangle \quad \Lambda' = \Lambda[x \mapsto (\mathbb{I}, \mathit{snd}(\Lambda(x)))]}{\Psi, \Phi, \Lambda \vdash \text{assume}(\mathit{len}(x) \leq \nu) : \Phi, \Lambda'} \quad (5)$$

$$\frac{\mathcal{A} = \mathit{snd}(\Psi(\mathcal{E})) \cap \mathit{snd}(\Lambda(x)) \quad x \notin \Phi \vee \mathcal{A} = \mathcal{A}^\emptyset \vee \mathit{fst}(\Psi(\mathcal{E})) \not\in \mathit{fst}(\Lambda(x))}{\Psi, \Phi, \Lambda \vdash \text{match}(x, \mathcal{E}) : \Phi, \Lambda} \quad (6)$$

$$\frac{\Psi, \Phi, \Lambda \vdash S_1 : \Phi_1, \Lambda_1 \quad \Psi, \Phi_1, \Lambda_1 \vdash S_2 : \Phi_2, \Lambda_2}{\Psi, \Phi, \Lambda \vdash S_1; S_2 : \Phi_2, \Lambda_2} \quad (7)$$

$$\frac{\Psi, \Phi, \Lambda \vdash S_1 : \Phi_1, \Lambda_1 \quad \Psi, \Phi, \Lambda \vdash S_2 : \Phi_2, \Lambda_2 \quad \Phi' = \Phi_1 \cup \Phi_2,\ \Lambda' = \Lambda_1 \sqcup \Lambda_2}{\Psi, \Phi, \Lambda \vdash \text{if}(\star) \text{ then } S_1 \text{ else } S_2 : \Phi', \Lambda'} \quad (8)$$

$$\frac{\Phi^* \supseteq \Phi,\ \Lambda^* \sqsupseteq \Lambda \quad \Psi, \Phi^*, \Lambda^* \vdash S : \Phi^*, \Lambda^*}{\Psi, \Phi, \Lambda \vdash \text{while}(\star) \text{ do } S : \Phi^*, \Lambda^*} \quad (9)$$

Fig. 13: Inference rules describing static analysis

Java statement or predicate	Strimp **translation**
`x.contains(s)`	$assume(x \in (\Sigma^* \cdot s \cdot \Sigma^*));\ assume(len(s) \leq len(x))$
`y = x.replaceAll(a, b)`	$(y := x \otimes y :=?);\ assume(y \in (!a)^*);\ assume(len(y) \leq len(x))$
`y = x.substring(c1, c2)`	$(y := x \otimes y :=?);\ assume(len(y) \leq c_2 - c_1)$
`x.length() <= c`	$assume(len(x) \leq c)$
`x.split(a).length() == c`	$assume(x \in ((!a)^* \cdot a \cdot (!a)^*)^c)$
`x.indexOf(s) != -1`	$assume(x \in (\Sigma^* \cdot s \cdot \Sigma^*));\ assume(len(s) \leq len(x))$
`x.endsWith(y)`	$assume(x \in (\Sigma^* \cdot y));\ assume(len(y) \leq len(x))$
`x.equals(y)`	$assume(x \in y); assume(len(x) \leq len(y)); assume(len(y) \leq len(x))$
`x.matches(`$\mathcal{E}$`)`	$assume(x \in \mathcal{E});$
`x.startsWith(y)`	$assume(x \in (y \cdot \Sigma^*));\ assume(len(y) \leq len(x))$

Table 1: Examples illustrating translation from Java string operations to Strimp constructs. Here x, y denote variables, s denotes string constants, a, b represent distinct characters, and c, c_1, c_2 represent integer constants. The notation $!a$ means any character other than a, and $S_1 \otimes S_2$ is syntactic sugar for *if*($\star$) *then* S_1 *else* S_2. Observe that the statement $y := x \otimes y :=?$ has the effect of tainting y if x is tainted but does not introduce any assumptions about the content or size of string y.

these conditions in the premise of the *match* rule are not met, then the program may contain a vulnerability.

The next rules for sequencing (7) and conditionals (8) are fairly standard. Since we take the union of the taint abstractions in rule (7), a variable x becomes tainted if it was tainted in either branch of the conditional. Also, note that the join operator on string abstractions is defined in Fig. 12. Finally, the last rule describes the analysis of loops. In particular, rule (9) states that the abstractions Φ^* and Λ^* overapproximate the behavior of the loop because (a) they subsume the initial abstractions Φ, Λ (first premise), and (b) they are inductive (second premise). While this rule does not describe how to compute Φ^* and Λ^* in an algorithmic way, our implementation performs standard fixed point computation (using widening) to find these loop invariants.