Static Analysis for Regular Expression Denial-of-Service Attacks

James Kirrage, Asiri Rathnayake, and Hayo Thielecke

University of Birmingham, UK

Abstract. Regular expressions are a concise yet expressive language for expressing patterns. For instance, in networked software, they are used for input validation and intrusion detection. Yet some widely deployed regular expression matchers based on backtracking are themselves vulnerable to denial-of-service attacks, since their runtime can be exponential for certain input strings. This paper presents a static analysis for detecting such vulnerable regular expressions. The running time of the analysis compares favourably with tools based on fuzzing, that is, randomly generating inputs and measuring how long matching them takes. Unlike fuzzers, the analysis pinpoints the source of the vulnerability and generates possible malicious inputs for programmers to use in security testing. Moreover, the analysis has a firm theoretical foundation in abstract machines. Testing the analysis on two large repositories of regular expressions shows that the analysis is able to find significant numbers of vulnerable regular expressions in a matter of seconds.

1 Introduction

Regular expression matching is a ubiquitous technique for reading and validating input, particularly in web software. While pattern matchers are among the standard techniques for defending against malicious input, they are themselves vulnerable. The root cause of the vulnerability is that widely deployed regular expression matchers, like the one in the Java libraries, are based on *backtracking* algorithms, rather than the construction of a Deterministic Finite Automaton (DFA), as used for lexers in compiler construction [13,2]. One reason for relying on backtracking rather than a DFA construction is to support a more expressive pattern specification language commonly referred to as "regexes". Constructs such as back-references supported by such regex languages go beyond regular and even context-free languages and are known to be computationally expensive [1]. However, even if restricted to purely regular constructs, backtracking matchers may have a running time that is exponential in the size of the input [6], potentially causing a regular expression denial-of-service (ReDoS) attack [19]. It is this potentially exponential runtime on pure regular expressions (without backreferences) that we are concerned about in this paper. Part of our motivation is that, for purely regular expressions, the attack could be defended against by avoiding backtracking matchers and using more efficient techniques [7,26] instead.

J. Lopez, X. Huang, and R. Sandhu (Eds.): NSS 2013, LNCS 7873, pp. 135–148, 2013.

For a minimalistic example [6], consider matching the regular expression `a**` against the input string `a`...`a b`, with n repetitions of `a`. A backtracking matcher takes an exponential time [6] in n when trying to find a match; all matching attempts fail in the end due to the trailing `b`. For such vulnerable regular expressions, an attacker can craft an input of moderate size which causes the matcher to take so long that for all practical purposes the matcher fails to terminate, leading to a denial-of-service attack. Here we assume that the regular expression itself cannot be manipulated by the attacker but that it is matched against a string that is user-malleable.

While the regular expression `a**` as above is contrived, one of the questions we set out to answer is how prevalent such vulnerable expressions are in the real world. As finding vulnerabilities manually in code is time consuming and error-prone, there is growing interest in automated tools for static analysis for security [14,5], motivating us to design an analysis for ReDoS.

Educating and warning programmers is crucial to defending against attacks on software. The standard coverage of regular expressions in the computer science curriculum, covering DFAs in courses on computability [13] or compiler construction [2], is not necessarily sufficient to raise awareness about the possibility of ReDoS. Our analysis constructs a series of attack strings, so that developers can confirm the exponential runtime for themselves.

This paper makes the following contributions:

1. We present an efficient static analysis for DoS on pure regular expressions.
2. The design of the tool has a firm theoretical foundation based on abstract machines [20] and derivatives [4] for regular expressions.
3. We report finding vulnerable regular expressions in the wild.

In Section 2, we describe backtracking regular expression matchers as abstract machines, so that we have a precise model of what it means for a matching attempt to take an exponential number of steps. We build on the abstract machine in designing our static analysis in Section 3, which we have implemented in OCaml as described in Section 4. Experimental results in testing the analysis on two large corpora of regular expressions are reported in Section 5. Finally, Section 6 concludes with a discussion of related work and directions of further research. The code of the tool and data sets are available at this URL:
`http://www.cs.bham.ac.uk/~hxt/research/rxxr.shtml`

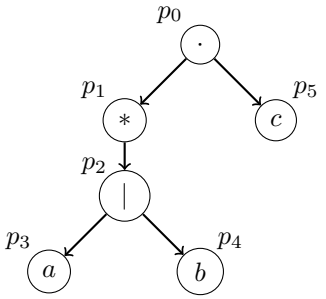
2 Regular Expression Matching by Backtracking

This and the next section present the theoretical basis for our analysis. Readers primarily interested in the results may wish to skim them.

We start with the following minimal syntax for regular expressions:

$e ::= \varepsilon$		Empty expression
	`a`	Constant, where `a` is an input symbol
	$e_1 \cdot e_2$	Concatenation
	$e_1 \mid e_2$	Alternation
	e^*	Kleene star

The $\cdot$ in concatenation $e_1 \cdot e_2$ is usually omitted, except when it is useful for emphasis, as in a syntax tree. Following the usual parser construction methods [2], we can define a parser which is capable of transforming (parsing) a given regular expression into an AST (abstract syntax tree) which complies with the above grammar. As an example, the AST constructed by such a parser for the regular expression $(a \mid b)^*c$ can be visualized in the following manner:



p	$\pi(p)$
p_0	$p_1 \cdot p_5$
p_1	${p_2}^*$
p_2	$p_3 \mid p_4$
p_3	a
p_4	b
p_5	c

Notice that we have employed a pointer notation to illustrate the AST structure; this is quite natural given that in most programming languages, such an AST would be defined using a similar pointer-based structure definition. Each node of this AST corresponds to a unique sub-expression of the original regular expression, the relationships among these nodes are given on the table to the right. We have used the notation $\pi(p)$ to signify the dereferencing of the pointer p with respect to the heap π in which the above AST is constructed. A formal definition of π was avoided in order to keep the notational clutter to a minimum, interested readers may refer [20] for a more precise definition of π.

Having parsed the regular expression into an AST, the next step is to construct an NFA structure that allows us to define a backtracking pattern matcher. While there are several standard NFA construction techniques [2], we opt for a slightly different construction which greatly simplifies the rest of the discussion. The idea is to associate a continuation pointer `cont` with each of the nodes in the AST such that `cont` points to the *following* (continuation) expression for each of the sub-expressions in the AST. In other words, `cont` identifies the "next sub-expression" which must be matched after matching the given sub-expression. More formally, `cont` is defined as follows:

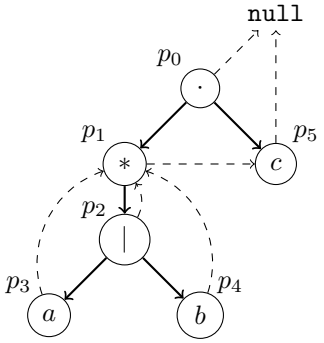
Definition 1. *Let* `cont` *be a function*

$$\texttt{cont} : \mathrm{dom}(\pi) \to (\mathrm{dom}(\pi) \cup \{\texttt{null}\})$$

Such that,

- *If* $\pi(p) = (p_1 \mid p_2)$*, then* $\texttt{cont}\, p_1 = \texttt{cont}\, p$ *and* $\texttt{cont}\, p_2 = \texttt{cont}\, p$
- *If* $\pi(p) = (p_1 \cdot p_2)$*, then* $\texttt{cont}\, p_1 = p_2$ *and* $\texttt{cont}\, p_2 = \texttt{cont}\, p$
- *If* $\pi(p) = (p_1)^*$*, then* $\texttt{cont}\, p_1 = p$
- $\texttt{cont}\, p_0 = \texttt{null}$*, where* p_0 *is the pointer to the root of the AST.*

The following example illustrates the NFA constructed this way for the regular expression $(a \mid b)^*c$:



p	$\pi(p)$	$\texttt{cont}\, p$
p_0	$p_1 \cdot p_5$	`null`
p_1	${p_2}^*$	p_5
p_2	$p_3 \mid p_4$	p_1
p_3	a	p_1
p_4	b	p_1
p_5	c	`null`

Here the dashed arrows identify the `cont` pointer for each of the AST nodes. Readers familiar with Thompson's construction [26,2] will realize that the resulting NFA is a slightly pessimized version of that resulting from Thompson's algorithm. The reason for this pessimization is purely of presentational nature; it helps to visualize the NFA as an AST with an overlay of a `cont` pointer mesh so that the structure of the original regular expression is still available in the AST portion. Furthermore, this presentation allows the definitions and proofs to be presented in an inductive fashion with respect to the structure of the expressions.

With the NFA defined, we present a simple non-deterministic regular expression matcher in the form of an abstract-machine called the PWπ machine:

Definition 2. *A configuration of the PWπ machine consists of two components:*

$$\langle p\,;\, w\rangle$$

The p component represents the current sub-expression (similar to a code pointer) while w corresponds to the rest of the input string that remains to be matched. The transitions of this machine are as follows:

$$\begin{aligned}
\langle p\,;\, w\rangle &\to \langle p_1\,;\, w\rangle \;\; \textit{if } \pi(p) = (p_1 \mid p_2)\\
\langle p\,;\, w\rangle &\to \langle p_2\,;\, w\rangle \;\; \textit{if } \pi(p) = (p_1 \mid p_2)\\
\langle p\,;\, w\rangle &\to \langle q\,;\, w\rangle \;\; \textit{if } \pi(p) = {p_1}^* \wedge \texttt{cont}\, p = q\\
\langle p\,;\, w\rangle &\to \langle p_1\,;\, w\rangle \;\; \textit{if } \pi(p) = {p_1}^*\\
\langle p\,;\, w\rangle &\to \langle p_1\,;\, w\rangle \;\; \textit{if } \pi(p) = (p_1 \cdot p_2)\\
\langle p\,;\, aw\rangle &\to \langle q\,;\, w\rangle \;\; \textit{if } \pi(p) = a \wedge \texttt{cont}\, p = q\\
\langle p\,;\, w\rangle &\to \langle q\,;\, w\rangle \;\; \textit{if } \pi(p) = \varepsilon \wedge \texttt{cont}\, p = q
\end{aligned}$$

The initial state of the PWπ machine is $\langle p_0\,;\, w\rangle$, where p_0 is the root of the AST corresponding to the input expression and w is the input string. The machine may terminate in the state $\langle \texttt{null}\,;\, w''\rangle$ where it has matched the original regular

expression against some prefix w' of the original input string w such that $w = w'w''$. Apart from the successful termination, the machine may also terminate if it enters into a configuration where none of the above transitions apply.

The PWπ machine searches for a matching prefix by non-deterministically making a choice whenever it has to branch at alternation or Kleene nodes. While this machine is not very useful in practice, it allows us to arrive at a precise model for backtracking regular expression matchers. Backtracking matchers operate by attempting all the possible search paths in order; this allows us to model them with a stack of PWπ machines. We call the resulting machine the PWFπ machine:

Definition 3. *The PWFπ machine consists of a stack of PWπ machines. The transitions of the PWFπ machine are given below:*

$$\frac{\langle p\ ;\ w\rangle \to \langle q\ ;\ w'\rangle}{\langle p\ ;\ w\rangle :: f \to \langle q\ ;\ w'\rangle :: f} \qquad \frac{\langle p\ ;\ w\rangle \not\to}{\langle p\ ;\ w\rangle :: f \to f}$$

$$\frac{\langle p\ ;\ w\rangle \to \langle q_1\ ;\ w\rangle \qquad \langle p\ ;\ w\rangle \to \langle q_2\ ;\ w\rangle}{\langle p\ ;\ w\rangle :: f \to \langle q_1\ ;\ w\rangle :: \langle q_2\ ;\ w\rangle :: f}$$

The initial state of the PWFπ machine is $[\langle p_0\ ;\ w\rangle]$. The machine may terminate if one of the PWπ machines locates a match or if none of them succeeds in finding a match. In the latter case the PWFπ machine has exhausted the entire search space and determined that the input string cannot be matched by the regular expression in question.

The PWFπ machine allows us to analyze backtracking regular expression matchers at an abstract level without concerning ourselves about any implementation specific details. More importantly, it gives an accurate cost model of backtracking matchers; the number of steps executed by the PWFπ machine corresponds to the amount of work a backtracking matcher has to perform when searching for a match. In the following sections we employ these ideas to develop and implement our static analysis.

3 Static Analysis for Exponential Blowup

The problem we are aiming to solve is this: given a regular expression e, represented as in Section 2, are there input strings x, y, and z, such that:

1. Reading x takes the machine to a pointer p_0 that is the root of a Kleene star expression.
2. Reading the input w takes the machine from p_0 back to p_0, and in at least two different ways, that is, along two different paths in the NFA.
3. Reading the input z when starting from p_0 causes the match to fail.

We call x the *prefix*, w the *pumpable* string by analogy with pumping lemmas in automata theory [13], and z the *failure suffix*.

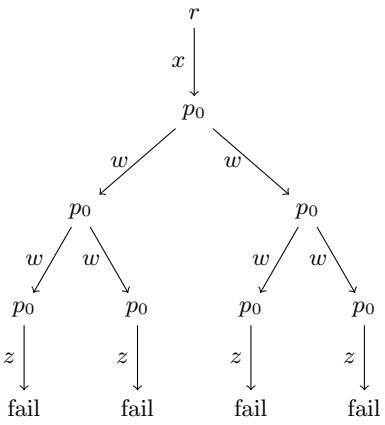


Fig. 1. The search tree for $x\,w\,w\,y$

From these three strings, malicious inputs can be constructed: the n-th malicious input is $x\,w^n\,z$. Figure 1 illustrates the search tree that a backtracking matcher has to explore when w is pumped twice. Because w can be matched in two different ways, the tree branches every time a w is read from the input. All branches fail in the end due to the trailing z, so that the matcher must explore the whole tree.

To state the analysis more formally, we will need to define paths in the matcher.

Definition 4. *A path of pointers, $t : p \xrightarrow{w} q$ is defined according to the following inductive rules:*

- *For each pointer p, $[p] : p \xrightarrow{\varepsilon} p$ is a path (identity).*
- *If $t : p \xrightarrow{w} q$ is a path and there exists a PWπ transition such that:*

$$\langle q\,;\, w'w_1\rangle \to \langle q'\,;\, w_1\rangle$$

Then $t \cdot [q'] : p \xrightarrow{ww'} q'$ is also a path.

Lemma 1. *The path $t : p \xrightarrow{w} q$ $(q \neq p)$ exists if and only if a PWπ run exists such that:*

$$\langle p\,;\, ww'\rangle \to \cdots \to \langle q\,;\, w'\rangle$$

Lemma 1 associates a unique string w with each path of pointers (the sub-string matched by the corresponding PWπ run). However, note that the inverse of this implication does not hold; there can be input strings for which we may find more than one PWπ run. In fact, it is this property of paths that leads us to the main theorem of this paper:

Theorem 1. *For a given Kleene expression p_0 where $\pi(p_0) = {p_1}^*$, if at least two paths exist such that $t_1 : p_1 \xrightarrow{w} p_0$, $t_2 : p_1 \xrightarrow{w} p_0$ and $t_1 \neq t_2$, then a regular expression involving p_0 exhibits $o(2^n)$ runtime on a backtracking regular expression matcher for input strings of the form xw^nz where x is a sub-string matching the prefix of p_0 and z is such that xw^nz fails to match the overall expression.*

While a formal proof of Theorem 1 is outside of the scope of this paper, we sketch its proof with reference to Figure 1. The prefix x causes the PWFπ machine to advance into a state where it has to match p_0 against the remainder of the input string, which leads to the branching of the search tree. Finally, the suffix z at the end of the input causes each search path to fail, which in turns forces the PWFπ machine to backtrack and explore the entire search tree before concluding that a match cannot be found. For the complexity, note that each additional pumping increases the size of the input by a constant (the length of w) whereas it doubles the size of the binary subtree given by the w branches, as well as the number of failed attempts to match z at the end. If there are more than 2 ways to match the pumpable string, say b, then b rather than 2 becomes the base of the exponent, but 2 is still a lower bound. The matching of the prefix x at the beginning contributes a constant to the runtime, which can be disregarded relative to the exponential growth. Thus the lower bound for the number of steps is exponential.

3.1 Generating the Pumpable String

The most important step in generating an attack string for a vulnerable regular expression is to generate the pumpable string w in xw^nz (for some Kleene sub-expression). In order to arrive at the machine for building the pumpable string, we must first introduce several utility definitions. Note that in the remainder of this discussion, p_0 refers to a Kleene expression such that $\pi(p_0) = {p_1}^*$.

Definition 5. *For a given pointer p, the operation $\Box p$ (called evolve) is defined as:*

$$\Box p = [q \mid \exists t.t : p \xrightarrow{\varepsilon} q \wedge \exists a.\pi(q) = a]$$

Notice that the result of $\Box p$ is a list of pointers.

Definition 6. *The function $\mathcal{D}_a(P)$, (called derive) is defined on a list of pointers P and an input symbol a according to the following rules:*

$$\mathcal{D}_a([\,]) = [\,]$$

$$\mathcal{D}_a(h :: t) = \begin{cases} \mathcal{D}_a(t) & \text{if } \pi(h) = b, b \neq a \\ q :: \mathcal{D}_a(t) & \text{if } \pi(h) = a \wedge \texttt{cont}\, h = q \\ \mathcal{D}_a(\Box h \cdot t) & \text{otherwise.} \end{cases}$$

The definition $\mathcal{D}_a(P)$ is analogous to Brzozowski's derivatives of regular expressions [4]. In essence, the analysis computes derivatives of a Kleene expression in order to find two different matcher states for the same input string.

Definition 7. *A wP frame is defined as a pair (w, P) where w is a string and P is a list of pointers. A non-deterministic transition relation is defined on wP frames as follows:*

$$\frac{\mathcal{D}_a(P) \neq [\,]}{(w, P) \to (w\,a, \mathcal{D}_a(P))}$$

Definition 8. *The HFπ machine has configurations of the following form:*

$$\langle H \,;\, f \rangle$$

Here H (history) represents a set of (sorted) pointer lists and f is a list of wP frames. A deterministic transition relation defines the behavior of this machine as follows:

$$\frac{(w, P) \to (wx_0, P_0) \quad \ldots \quad (w, P) \to (wx_n, P_n) \quad \forall i. x_i \in \Sigma \quad P_i \notin H}{\langle H \,;\, (w, P) :: f \rangle \to \langle H \cup \{P_0, \ldots, P_n\} \,;\, f \cdot [(wx_0, P_0), \ldots, (wx_n, P_n)] \rangle}$$

The initial configuration of the HFπ machine is $\langle \emptyset \,;\, [(\varepsilon, [p_1])] \rangle$ and the machine can terminate in either of the following two configurations:

$$\langle H \,;\, [\,] \rangle$$

$$\langle H \,;\, (w, P) :: f \rangle \text{ where } \exists p', p'' \in P.\ \exists t', t''.\ t' : p' \xrightarrow{\varepsilon} p_0 \wedge t'' : p'' \xrightarrow{\varepsilon} p_0$$

In the former configuration the machine has determined the Kleene expression in question to be non-vulnerable while in the latter it has derived the pumpable string w.

3.2 Generating the Prefix and the Suffix

For a regular expression of the form $e_1\,(e_2{}^*)\,e_3$, apart from a pumpable string w, we must also generate a prefix x and a suffix z. The intention is that x would lead the matcher to the point where it has to match $e_2{}^*$, after which we can pump many copies of w to increase the search space of the matcher. However, a successful exploit also needs a suffix z that forces the matcher to fail and so to traverse the entire search tree.

Generating the prefix and the suffix might at first appear to be straightforward, since x and z can be generated from e_1 and e_3 such that x is in the language of e_1 and z is not in the language of e_3. However, upon closer inspection we realize that the choice of x and z can have un-intended effects on the final outcome of the match, as it is possible that e_1 could match part of the pumped string w^n in addition to the intended sub-string x. A similar situation could occur with e_2 and z. In other words, x, w and z are dependent on each other in complicated ways. Writing $e \downarrow y$ for e matches y, we have the following conditions:

$$e_1 \downarrow x \qquad e_2 \downarrow w \text{ (with multiple traces)} \qquad e_1\,e_2{}^*\,e_3 \not\downarrow x\,w^n\,z$$

At present, we have chosen not to solve this problem in full generality, but resolve to employ heuristics that find prefixes and suffixes for many practical expressions, as illustrated in the results section.

4 Implementation of the Static Analysis

We have implemented the HFπ machine described in Section 3 using the OCaml programming language. OCaml is well suited to programming abstract syntax, and hence a popular choice for writing static analyses. One of the major obstacles faced with the implementation is that in order to be able to analyze real-world regular expressions, it was necessary to build a sophisticated parser. In this regard, we decided to support the most common elements of the Perl / PCRE standards, as these seem to be the most commonly used (and adapted) syntaxes. It should be noted that the current implementation does not support back-references or look-around expressions due to their inherent complexity; it remains to be seen if the static analysis proposed in this work can be adapted to handle such "regexes". However, as it was explained earlier, exponential vulnerabilities in pattern specifications are not necessarily dependent on the use of back-references or other advanced constructs (although one would expect such constructs to further increase the search space of a backtracking matcher). A detailed description of the pattern specification syntax currently supported by the implementation has been included in the resources accompanying this paper.

The implementation closely follows the description of the HFπ machine presented in Section 3. The history component H is implemented as a set of sorted integer lists, where a single sorted integer list corresponds to a list of nodes pointed by the pointer list P of a wP frame (w, P). This representation allows for quick elimination of looping wP frames. While the size of H is potentially exponential in the number of nodes of a given Kleene expression, for practical regular expressions we found this size to be well within manageable levels (as evidenced in the results section).

A technical complication not addressed in the current work is that the PWFπ machine (and naive backtracking matching algorithms in general) can enter into infinite loops for Kleene expressions where the enclosed sub-expression can match the empty string, i.e., where the sub-expression is nullable [9,12]. A similar problem occurs in the HFπ machine during the $\Box\, p$ operation. We have incorporated a variation of the technique proposed by Danvy and Nielsen [9] for detecting and terminating such infinite loops into the OCaml code for the $\Box\, p$ function, so that it terminates in all cases.

5 Experimental Results

The analysis was tested on two corpora of regexes (Figure 1). The first of these was extracted from an online regex library called *RegExLib* [21], which is a community-maintained regex archive; programmers from various disciplines submit their solutions to various pattern matching tasks, so that other developers can reuse these expressions for their own pattern matching needs. The second corpus was extracted from the popular intrusion detection and prevention system *Snort* [25], which contains regex-based pattern matching rules for inspecting

Table 1. Experimental results with RegExLib and Snort

	RegExLib	Snort
Total patterns	2994	12499
Analyzable (only regular constructs)	2213	9408
Uses Kleene star	1103	2741
Pumpable Kleene and suffix found	127	15
Pumpable Kleene only	20	4
No pumpable Kleene	2066	9389
Max HFπ steps	509	256
Total classification time (Intel Core 2 Duo 1.8 MHz, 4 GB RAM)	40 s	10 s

IP packets across network boundaries. The contrasting purposes of these two corpora allow us to get a better view of the seriousness of exponential vulnerabilities in practical regular expressions.

The regex archive for RegExLib was only available through the corresponding website [21]. Therefore, as the first step the expressions had to be scraped from their web source and adapted so that they can be fed into our tool. These adaptations include removing unnecessary white-space, comments and spurious line breaks. A detailed description of these adjustments as well as copies of both adjusted and un-adjusted data sets have been included with the resources accompanying this paper (also including the Python script used for scraping). The regexes for Snort, on the other hand, are embedded within plain text files that define the Snort rule set. A Python script (also included in the accompanying resources) allowed the extraction of these regexes, and no further processing was necessary.

The results of the HFπ static analysis on these two corpora of regexes are presented in Table 1. The figures show that we can process around 75% of each of the corpora with the current level of syntax support. Out of these analyzable amounts, it is notable that regular expressions from the RegExLib archive use the Kleene operator more frequently (about 50% of the analyzable expressions) than those from the Snort rule set (close to 30%). About 11.5% of the Kleene-based RegExLib expressions were found to have a pumpable Kleene expression as well as a suitable suffix, whereas for Snort this figure stands around 0.55%.

The vulnerabilities reported range from trivial programming errors to more complicated cases. For an example, the following regular expression is meant to validate time values in 24-hour format (from RegExLib):

```
^(([01][0-9]|[012][0-3]):([0-5][0-9]))*$
```

Here the author has mistakenly used the Kleene operator instead of the `?` operator to suggest the presence or non-presence of the value. This pattern works perfectly for all intended inputs. However, our analysis reports that this expression is

vulnerable with the pumpable string "`13:59`" and the suffix "`/`". This result gives the programmer a warning that the regular expression presents a DoS security risk if exposed to user-malleable input strings to match.

For a moderately complicated example, consider the following regular expression (again from RegExLib):

```
^([a-zA-z]:((\\([-*\.*\w+\s+\d+]+)|(\w+)\\)+)(\w+.zip)|(\w+.ZIP))$
```

This expression is meant to validate file paths to zip archives. Our tool identifies this expression as vulnerable and generates the prefix "`z:\ `", the pumpable string "`\zzz\`" and the empty string as the suffix. This is probably an unexpected input in the author's eye, and this is another way in which our tool can be useful in that it can point out potential mis-interpretations which may have materialized as vulnerabilities.

It is worth noting that the HFπ machine manages to classify both the corpora (the analyzable portions) in a matter of seconds on modest hardware. This shows that our static analysis is usable for most practical purposes, with the average classification time for an expression in the range of micro-seconds. The two extreme cases for which the machine took several seconds for the classification are given below (only the respective Kleene expressions):

```
([\d\w][-\d\w]{0,253}[\d\w]\.)+

([^\x00]{0,255}\x00)*
```

Here counting expressions `[-\d\w]{0,253}` and `[^\x00]{0,255}` were expanded out during the parsing phase. The expansion produces a large Kleene expression, which naturally requires more analysis during the HFπ simulation. However, it should be noted that such expressions are the exception rather than the norm.

Finally, it should be mentioned that all the vulnerabilities reported above were individually verified using a modified version of the PWFπ machine (which counts the number of steps taken for a particular matching operation). A sample of those vulnerabilities was also tested on the Java regular expression matcher.

6 Conclusions

We have presented a static analysis to help programmers defend against regular expression DoS attacks. Large numbers of regular expressions can be analysed quickly, and developers are given feedback on where in their regular expressions the problem has been identified as well as examples of malicious input.

As illustrated in Section 5, the prefix, pumpable string and failure suffix can be quite short. If their length is, say, 3, 5 and 0 characters, then an attacker only needs to spend a very small amount of effort in providing a malicious input of length 3+5*100 characters to cause a matching time in excess of 2^{100} steps. Even if a matching step takes only a nanosecond, such a running time takes, for all intents and purposes, forever. The attacker can still scale up the

attack by pumping a few times more and thereby correspondingly multiplying the matching time.

The fact that the complexity of checking a regular expression for exponential runtime may be computationally expensive in the worst case does not necessarily imply that such an analysis is futile. Type checking in functional languages like ML and Haskell also has high complexity [16,23], yet works efficiently in practice because the worst cases rarely occur in real-world code. There are even program analyses for undecidable problems like termination [3], so that the worst-case running time is infinite; what matters is that the analysis produces results in enough cases to be useful in practice. It is a common situation in program analysis that tools are not infallible (having false positives and negatives), but they are nonetheless useful for identifying points in code that need attention by a human expert [10].

6.1 Related Work

A general class of DoS attacks based on algorithmic complexities has been explored in [8]. In particular, the exponential runtime behavior of backtracking regular expression matchers has been discussed in [6] and [22]. The seriousness of this issue is further expounded in [24] and [18] where the authors demonstrate the mounting of DoS attacks on an IDS/IPS system (Snort) by exploiting the said vulnerability. The solutions proposed in these two works involve modifying the regular expressions and/or the matching algorithm in order to circumvent the problem in the context of IDS/IPS systems. We consider our work to be quite orthogonal and more general since it is based on a compile-time static analysis of regular expressions. However, it should be noted that both of those works concern of regexes with back-references, which is a feature we are yet to explore (known to be NP-hard [1]).

While the problem of ReDoS has been known for at least a decade, we are not aware of any previous static analysis for defending against it. A handful of tools exist that can assist programmers in finding such vulnerable regexes. Among these tools we found Microsoft's SDL Regex Fuzzer [17] and the RegexBuddy [15] to be the most usable implementations, as other tools were too unstable to be tested with complex expressions.

While RegexBuddy itself is not a security oriented software, it offers a debug mode, which can be used to detect what the authors of the tool refer to as *Catastrophic Backtracking* [11]. Even though such visual debugging methods can assist in detecting potential vulnerabilities, it would only be effective if the attack string is known in advance—this is where a static analysis method like the one presented on this paper has a clear advantage.

SDL Fuzzer, on the other hand, is aimed specifically at analyzing regular expression vulnerabilities. While details of the tool's internal workings are not publicly available, analyzing the associated documentation reveals that it operates fuzzing, i.e., by brute-forcing a sequence of generated strings through the regular expression in question to detect long running times. The main disadvantage of this tool is that it can take a very long time for the tool to classify a

given expression. Tests using some of the regular expressions used in the results section above revealed that it can take up to four minutes for the Fuzzer to classify certain expressions. It is an inherent limitation of fuzzers for exponential runtime DoS attacks that the finding out if something takes a long time by running it takes a long time. By contrast, our analysis statically analyzes an expression without ever running it. It is capable of classifying thousands of regular expressions in a matter of seconds. Furthermore, the output produced by the SDL Fuzzer only reports the fact that the expression in question failed to execute within a given time limit for some input string. Using this generated input string to pin-point the exact problem in the expression would be quite a daunting task. In contrast, our static analysis pin-points the exact Kleene expression that causes the vulnerability and allows programmers to test their matchers with a sequence of malicious inputs.

6.2 Directions for Further Research

In further work, we aim to broaden the coverage of our tool to include more regexes. Given its basis in our earlier work on abstract machines [20] and derivatives [4], we aim for a formal proof of the correctness of our analysis. We intend to release the source code of the tool as an open source project. More broadly, we hope that raising awareness of the dangers of backtracking matchers will help in the adoption of superior techniques for regular expression matching [7,26,20].

References

1. Aho, A.V.: Algorithms for Finding Patterns in Strings. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. A, pp. 255–300. MIT Press, Cambridge (1990)
2. Aho, A.V., Lam, M., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools, 2nd edn. Addison Wesley (2007)
3. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.W.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
4. Brzozowski, J.A.: Derivatives of Regular Expressions. J. ACM 11(4), 481–494 (1964)
5. Chess, B., McGraw, G.: Static analysis for security. IEEE Security & Privacy 2(6), 76–79 (2004)
6. Cox, R.: Regular Expression Matching Can Be Simple and Fast (but is slow in Java, Perl, Php, Python, Ruby, ...) (January 2007), `http://swtch.com/~rsc/regexp/regexp1.html`
7. Cox, R.: Regular expression matching: the virtual machine approach (December 2009), `http://swtch.com/~rsc/regexp/regexp2.html`
8. Crosby, S.A., Wallach, D.S.: Denial of Service via Algorithmic Complexity Attacks. In: Proceedings of the 12th USENIX Security Symposium, Washington, DC (August 2003)
9. Danvy, O., Nielsen, L.R.: Defunctionalization at Work. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP 2001, pp. 162–174. ACM, New York (2001)

10. Dowd, M., McDonald, J., Schuh, J.: The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Addison Wesley (2006)
11. Goyvaerts, J.: Runaway Regular Expressions: Catastrophic Backtracking (2009), http://www.regular-expressions.info/catastrophic.html
12. Harper, R.: Proof-Directed Debugging. J. Funct. Program. 9(4), 463–469 (1999)
13. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
14. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: Proceedings of the 14th Conference on USENIX Security Symposium, vol. 14, p. 18 (2005)
15. Just Great Software Co. Ltd. RegexBuddy (2012), http://www.regexbuddy.com/
16. Mairson, H.G.: Deciding ML typability is complete for deterministic exponential time. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 382–401. ACM (1989)
17. Microsoft. SDL Regex Fuzzer (2011), http://www.microsoft.com/en-gb/download/details.aspx?id=20095
18. Namjoshi, K., Narlikar, G.: Robust and Fast Pattern Matching for Intrusion Detection. In: Proceedings of the 29th Conference on Information Communications, INFOCOM 2010, pp. 740–748. IEEE Press, Piscataway (2010)
19. The Open Web Application Security Project (OWASP). Regular Expression Denial of Service - ReDoS (2012), https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS
20. Rathnayake, A., Thielecke, H.: Regular Expression Matching and Operational Semantics. In: Structural Operational Semantics (SOS 2011). Electronic Proceedings in Theoretical Computer Science (2011)
21. RegExLib.com. Regular Expression Library (2012), http://regexlib.com/
22. Roichman, A., Weidman, A.: Regular Expression Denial of Service (2012), http://www.checkmarx.com/white_papers/redos-regular-expression-denial-of-service/
23. Seidl, H., et al.: Haskell overloading is DEXPTIME-complete. Information Processing Letters 52(2), 57–60 (1994)
24. Smith, R., Estan, C., Jha, S.: Backtracking Algorithmic Complexity Attacks Against a NIDS. In: Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC 2006, pp. 89–98. IEEE Computer Society, Washington, DC (2006)
25. Sourcefire. Snort, IDS/IPS (2012), http://www.snort.org/
26. Thompson, K.: Programming Techniques: Regular Expression Search Algorithm. Communications of the ACM 11(6), 419–422 (1968)