Robust and Fast Pattern Matching for Intrusion Detection

Kedar Namjoshi
Bell Laboratories
Alcatel-Lucent
kedar@research.bell-labs.com

Girija Narlikar
Bell Laboratories
Alcatel-Lucent
girija@research.bell-labs.com

***Abstract*—The rule language of an Intrusion Detection System (IDS) plays a critical role in its effectiveness. A rule language must be expressive, in order to describe attack patterns as precisely as possible. It must also allow for a matching algorithm with predictable and low complexity, in order to ensure robustness against denial-of-service attacks.**

Unfortunately, these requirements often conflict. We show, for instance, that a single rule, when coupled with a backtracking matching algorithm, can bring the processing rate down to nearly ONE packet per second. Performance vulnerabilities of this type are known for patterns described using regular expressions, and can be avoided by using a deterministic matching algorithm. Increasingly, however, rules are being written using the more powerful regex syntax, which includes non-regular features such as back-references. The matching algorithm for general regex's is based on backtracking, and is thus vulnerable to attacks.

The main contribution of this paper is a deterministic algorithm for the full regex syntax, which builds upon the deterministic algorithm for regular expressions. We provide a (rough) complexity bound on the worst-case performance, and show that this bound can be tightened through compile-time analysis of the regex structure. These bounds can be used as an admissibility check, to isolate expressions that require further analysis. Finally, we present an implementation of these algorithms in the context of the Snort IDS, and experimental results on several packet traces which show substantial improvement over the backtracking algorithm.

I. Introduction

A signature-based intrusion detection system (IDS) or intrusion prevention system (IPS) protects a network by examining headers and contents of all packets entering or leaving it. It raises alerts or drops packets (in the case of an IPS) when it sees suspicious headers or payloads. Suspicious packets are detected by matching every incoming packet against a database of rules; each rule represents the signature of a security exploit.

The IDS rule language must be sufficiently powerful to represent current and future security exploits as accurately and precisely as possible. Otherwise, a large number of good packets may be incorrectly marked as harmful, or harmful packets may go undetected. Moreover, the packet processing rate must keep up with high line speeds without dropping packets or allowing bad packets through. These two goals often conflict, as there typically is a direct relationship between the expressiveness and complexity of the rule language and the packet processing time. Unless rules are written with care and the underlying pattern matching is implemented carefully, the packet processing algorithm may take a long time to complete. The resulting performance vulnerability can be exploited by an attacker to generate a low-bandwidth denial-of-service (DoS) attack on the IDS itself. For example, Figure 1(a) shows how a single packet trace brought the performance of the popular Snort IDS/IPS to a grinding halt. A detailed look at individual packet processing times in the problematic trace (number 7) shows some packets requiring over 3 orders of magnitude more time to process (Figure 1(b)). If an attacker were to send a very low-bandwidth trace composed of such "bad" packets to the IDS, its performance would be reduced to 1 packet per second.

This vulnerability can be traced back to a *backtracking*-based pattern matching algorithm for regular expressions implemented in the PCRE library used by Snort. It is known that these algorithms may exhibit *exponential* worst-case complexity (cf. [1], [2], [3]); the results described above provide a dramatic confirmation of the importance of this vulnerability in practice. The vulnerability can be avoided through determinization: either converting the regular expression to a deterministic automaton, or through an "on-the-fly" determinization method due to Thompson [4], used in the standard Unix tool `grep`.

Increasingly, however, as shown in Figure 3, rules in Snort are being written using a powerful extension of regular expressions with back-references, known as the *regex* language. A number of exploits such as buffer overflow attacks (for example, see [5]) can be expressed more precisely using back-references. Figure 2 shows a simplified regex of this kind. Back-references are a non-regular feature, so the known determinization methods are not applicable to regex's in general. This leaves the backtracking algorithm as the only option for matching general regex's. However, regular expressions which cause the backtracking algorithm to enter into its exponential behavior could be embedded inside expressions with back-references, re-exposing this serious vulnerability. Moreover, as we show, compile-time detection of potential exponential behavior is an NP-hard problem.

The main contribution of this paper is an on-the-fly determinization algorithm for the full regex syntax. Towards this result, we also give a construction of an extended non-deterministic finite automaton from a regex. The matching

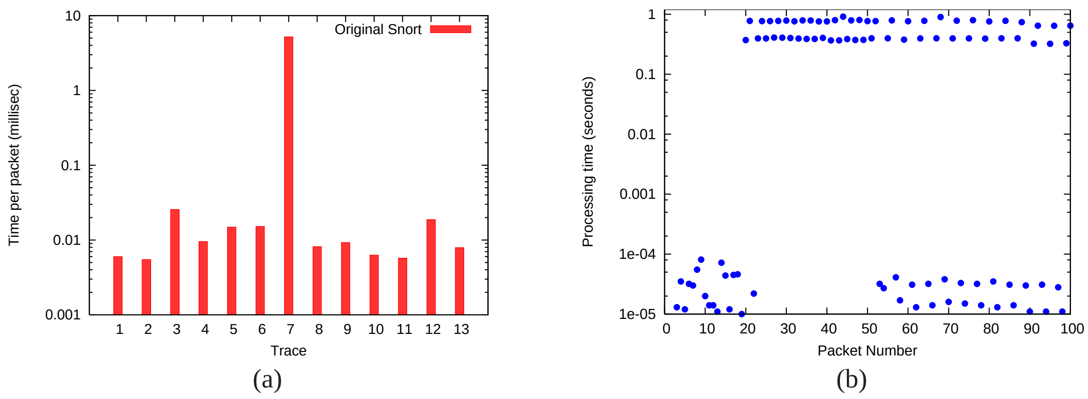

Fig. 1. (a) Average packet processing times for 13 different packet traces for the Snort IDS/IPS. Trace number 7 had some problematic TCP flows. (b) Packet processing times for individual packets from one such flow. Y-axes shown in log scale.

Attack	$<v>=<longstring>;\ \ldots;$ F($<v>$)
Regex	**(\w+)**="[^ "]{1024, }"; .* ; F(**1**)

Fig. 2. Use of a back reference to represent a (simplified) buffer overflow security exploit on function F. The back-reference, '\1', refers to the term **(\w+)**, which matches a variable name appearing earlier in the packet.

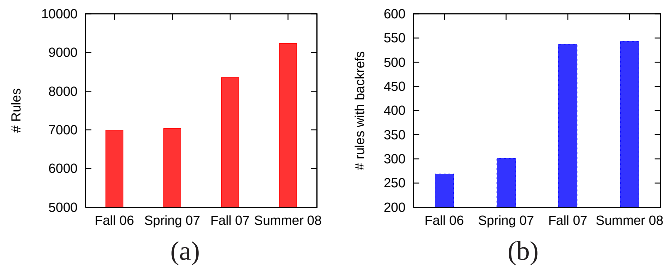

Fig. 3. Growth in the number of rules with regex's in Snort. (a) Total number of rules, and (b) Rules with regex's containing back references

algorithm has worst-case space and time complexity that is linear in the size of this automaton, but exponential in the number of back-references in the given expression. This is unavoidable, as the pattern matching problem for regex's is known to be NP-complete in the number of back-references [6].

A detailed examination of regex's from the Snort rule set suggests, however, that the worst-case complexity is unlikely in practice. We show that compile-time analysis of the "liveness" of back references, as well as of the number of possible positions in the input where a back reference can match, results in significantly tighter worst-case bounds. The compile-time analysis results can also be used to limit state space growth at run-time.

A backtracking approach may have an advantage over deterministic algorithms if the input packet is "bad"—i.e., if it matches the regex describing an attack pattern. For "good" packets, we show that the backtracking algorithm always does at least as much work as the deterministic algorithm; hence, the deterministic algorithm has the advantage. Normal traffic is overwhelmingly composed of good packets; thus, one might expect the performance of the deterministic algorithm to be good on average. Experimental results with a modified Snort implementation confirm this hypothesis.

The main advantage of a deterministic algorithm is that its performance is predictable. (We show that predicting the worst-case performance of the backtracking algorithm is NP-hard even for regular expressions.) The complexity bounds for the deterministic algorithm which are computed by the compile-time analysis can be used to separate out regex patterns that look vulnerable to attack, for more detailed manual analysis. Thus, the combination of the new algorithm, with the compile-time analysis of potential vulnerability results in significantly improved robustness to performance attacks.

We demonstrate our algorithm's benefits by adding it to the Snort IDS. Although backtracking is fast in the common case, it results in performance as low as 1–2 packets/sec in the face of malicious inputs. Our deterministic algorithm, on the other hand, has performance similar to the backtracking algorithm in the common case, while avoiding the orders of magnitude slowdown with malicious inputs.

The main contributions of this work can be summarized as follows.

- A new on-the-fly deterministic matching algorithm for general regex's, with detailed complexity analysis and predictable, robust behavior in practice,
- New compile-time analysis resulting in much tighter worst-case matching costs, and use of this analysis to improve run-time space usage and performance,
- Rule-by-rule analysis of the worst case matching cost for our deterministic algorithm to allow automatic detection and isolation of potentially "bad" rules, and
- Extensive experimental results which show significantly more robust performance for the new algorithm in the face of malicious packets.

II. Related Work

A number of hardware (FPGA/ASIC/CAM) and software solutions have been proposed to optimize running time of IDS/IPS systems (e.g., [7], [8], [9], [10], [11], [12]). While

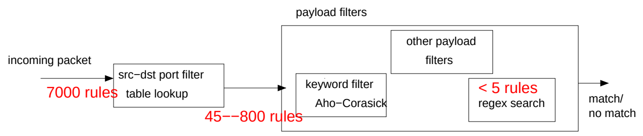

Fig. 4. Basic packet processing pipeline in the Snort IDS. Intelligent filtering significantly reduces the number of rules that need to be matched in full with a packet. The numbers denote the average number of rules remaining at each stage when processing the 25 packet traces. Over all traces, on average less than 5 rules (and often close to zero) rules need to be checked for a regexp match.

these solutions significantly accelerate the average packet processing time, they do not aim to minimize the worst case running time. Given the critical functionality of an IDS/IPS, it is important to test its survivability or robustness in the face of "bad" inputs and detect any performance vulnerabilities [13]. Therefore, we focus on the most complex and time-consuming operation in an IDS, which is to match packets against arbitrary regex's, including those with back references.

Performance problems with regular expression matching are well known [14], [15], [16], [17]. Vulnerabilities in various stages of regular expression processing have been demonstrated across operating systems and applications [14]. Exponential behavior of NFA-based backtracking algorithms for regular expression matching has also been demonstrated [1], [2], [3]. Regular expressions are known to be complex to write well [14], and these papers have provided certain guidelines to minimize their performance vulnerabilities. These guidelines include avoiding regex's (back references), avoiding backtracking, using a memory-efficient deterministic algorithm for regular expressions such as that used in grep [18], using only well-tested regular expressions, avoiding known patterns that incur exponential behavior [19], [1], and limiting time and memory requirements of the matching phase.

While all these suggestions are very practical, only some are applicable in the context of an IDS. IDS patterns are continually added and updated, mostly by network managers or security professionals, who are not cognizant of the underlying pattern matching algorithms. Further, some exploits are most accurately expressed with complex syntax like regex's—the Snort ruleset, for example, contains several rules which rely on back references to accurately express buffer overflow attacks. Limiting time or memory [20] may result in failure to detect bad packets or dropping of harmless packets. Therefore limiting the IDS rule syntax, time or memory restrictions, or relying on the prudence of the rule writers are not appropriate for IDS applications. Instead, we propose to allow a powerful rule language, but also to accompany it with a static check for all the IDS rules to detect worst case performance problems. We can then guarantee that an adversary cannot craft malicious packets to attack vulnerable IDS rules.

Recent work has a proposal for a deterministic algorithm to handle back references [21]. However, their algorithm does not handle expressions with multiple occurrences of a back-reference, and the paper does not include methods for obtaining tight worst-case bounds, or experimental evaluation with real traces. Since determinism is useful for high performance, much recent work has focused on the DFA state explosion caused by combining regular expressions from multiple rules. Space and time overheads of DFAs created for regular expression matching have been optimized by grouping or rewriting of rule sets [10], or by creating compact new data structures [22], [23], [24]. Hybrid NFA/DFA approaches [25], or extensions of DFAs with scratch memory [26] can help reduce run-time and state explosion due to the combination of a large number of regular expressions into a single structure. While these approaches solve other important problems, they do not efficiently handle regex's—the solutions are complementary to ours.

As pointed out in prior work [8], [27] and confirmed in our experiments (see Figure 4), IDSs like Snort perform very fast and extensive pre-filtering of the rules at runtime based on packet headers, along with fast multiple keyword-based pattern matching [28]. As a result, full regular expressions matching for each packet must be performed for only a handful of rules; this set of rules is determined at runtime based on the packet contents. Therefore, combining hundreds or thousands of regular expressions into a single (optimized) DFA is not necessarily the best strategy for rule-based IDS systems. Moreover, this earlier work does not address the case of regex's with back-references.

Several other types of attacks on IDS's have been studied in the literature. For example, hash tables that exhibit quadratic behavior due to collisions can slow down systems [29]. Another kind of backtracking performance vulnerability was discovered in the overall payload filtering stage of Snort [30]. The authors used memoization to protect against the vulnerability, resulting in significant performance improvement for certain packet traces. While memoization works well for the overall rule evaluation (which consists of several individual regex matches), it is not a good strategy for pattern matching against packets. For good packets (those that do not match the pattern), it requires storing the entire search tree, rather than the horizontal "slice" needed by an on-the-fly method.

III. Pattern Matching Algorithms for IDSs

In this section, we examine methods for pattern matching in detail, and give examples of regex's where the backtracking method (also called the "NFA" method) exhibits exponential cost. We show that it is NP-hard to predict in advance whether the NFA method will incur exponential cost for a given regular expression. We present a DFA-style method for full regex's, including arbitrary uses of backreferences, and examine its worst-case time and space complexity. As we show, more precise bounds can be obtained through static analysis of the computed automaton. The implementation of this method, and experimental results are described in the following section.

A. Packet Classification in an IDS

IDS's, such as Snort, usually apply several pattern matching filters to a packet. The goal is to classify good packets quickly, and thus reduce the average matching time. An incoming packet is first classified according to header information; then, by whether it matches a set of representative keywords taken from the full rules—typically using the Aho-Corasick algorithm [28]. This usually eliminates most rules from consideration. If some rules remain, the packet is then matched against these rules, which usually contain regex's. In all our tests with Snort, the initial filters proved to be very effective, reducing the number of regex matches required to at most 10's of rules (and on average well under 5 rules), contrasted with the 1000's of potential rules.

The most complex pattern matching task in an IDS/IPS is regular expression matching. Snort allows unrestricted use of regex's, making it challenging to predict worst-case performance, and therefore, to guard in advance against performance attacks.

B. Regular Expression and Regex Matching

Matching for regular expressions can be implemented using either an NFA-style algorithm or a DFA-style algorithm. To understand these algorithms, recall that a regular expression can be converted in linear time to a nondeterministic finite-state automaton ("nfa") (cf. [31]).

The NFA-style algorithm is a depth-first search of this nfa and the input packet. The depth-first nature and the use of backtracking ensures that the memory required is low; however, there are simple cases where it requires exponential running time [1]. An example is the regular expression $((a|b)^*)^*b$. If the packet ends in a rather than b, the NFA-style algorithm will backtrack, attempting to partition the packet in alternative ways. The number of such ways is roughly 2^n, where n is the length of the packet. The slowdown in Figure 1 is due to a similar expression in one of the Snort rules.

Backtracking can be replaced with memoization—i.e., remembering the (nfa state, input position) combinations that occur in the search—, but this results in a large space consumption when the input does not match the pattern. That being the common case for IDS applications, memoization is typically not used.

An nfa can be turned into a dfa (deterministic finite-state automaton) by the subset construction. Each state of the dfa is a subset of nfa states; after reading a prefix σ of the input, the subset-dfa state represents precisely the states that the nfa could be in after reading σ. The translation from an nfa to a dfa requires, in general, exponential space in the size of the regular expression.

Instead of constructing the entire dfa, the DFA-style algorithm [4] keeps track of the current subset-dfa state while scanning the input left-to-right. Thus, the DFA-style algorithm *always* takes time linear in the packet length, requiring space at most the number of nfa states, which is proportional to the length of the regular expression.

For inputs which match the regular expression, the backtracking method has a potential of detecting a match with less work than that done by the DFA-method. However, if there is no match, the backtracking method must examine all possible runs of the automaton on the input, and thus incur cost that is at least that of the DFA method, which can be viewed as checking all runs together. As noted in the introduction, the expected common case for IDS applications is for nearly all packets to be benign, on which the NFA-method has no time advantage over the DFA-method. For those packets that do match, the NFA method has high variability in its performance, ranging from polynomial to exponential time in the length of the packet. Thus, in both cases the DFA approach seems preferable.

A question that might occur to the reader at this point is whether it is possible to determine in advance the worst-case behavior of the NFA-method over all possible inputs. This turns out to be a hard problem, specifically, NP-hard.

Theorem 0: The questions below are NP-hard.

1) Given a regular expression and numbers L and K, is there a word of length L on which the backtracking NFA search examines at least K states?
2) Given a regular expression, is there a family of words on which the backtracking NFA search has exponential time complexity in the length of the word?

C. Regex's: Extending Regular Expressions

The standard regular expression syntax is as follows. A regular expression R has the following forms: ϵ (the empty string), a (a single character), $(R1|R2)$ (union), $R1; R2$ (concatenation), and $R*$ (reflexive transitive closure).

To this, regex's allow the addition of back-references to strings matched in brackets. An expression such as "$(a|b)*\backslash 1$" matches precisely those input words of the form ww (such as aa or $abab$), as the $\backslash 1$ has to refer to a word identical to that matched by the bracketed expression.

The addition of backreferences results in non-regular expressions; for instance, the above language is well-known to be strictly context-sensitive. It is shown in [6] that the matching problem (does an input match a regex?) is NP-complete. Moreover, most interesting questions about context-sensitive languages are undecidable (cf. [32]); hence, such representations are hard to analyze.

The NFA-algorithm for full regex's is susceptible to the same problems as for regular expressions. *In particular, it is possible for a rule to have a bad regular expression embedded in a regex which has backreferences.* Such expressions, by the hardness results above, are difficult or impossible to analyze a priori. Therefore, we look for a deterministic, DFA-like algorithm with predictable performance guarantees. This algorithm builds on the classical algorithm by Ken Thompson [4], used in Unix tools such as `grep`, for matching input words against a standard regular expression. We briefly describe the classical algorithm before describing the necessary extensions.

D. Classical matching algorithm for regular expressions

The classical algorithm works in two phases:

1) A regular expression is compiled into a non-deterministic finite automaton (NFA), via recursion on the structure of the regular expression

2) On a given input word, the matching algorithm scans the word from left to right, maintaining a *set* of active states of the NFA. If, on reaching the end of the word, an accepting state of the NFA appears in the current active set, the word is accepted; otherwise, it is rejected.

This algorithm can be seen as determinizing the NFA "on-the-fly". The algorithm has the important property that it scans each character of the input word at most once. The worst-case space complexity is $O(K)$, where K is the number of states in the NFA. The worst-case time complexity is $O(K * m)$, where m is the length of the input word. However, caching the computed state transitions appears to reduce the time requirement in practice to $O(K + m)$, as described in [31].

Formally, a *non-deterministic finite automaton* (NFA) is represented by a tuple $(Q, Q_0, \delta, \Sigma, F)$, where

- Q is a finite set of *states*,
- Q_0 is a subset of Q, defining the *initial states*,
- Σ is the *input alphabet*, the set of legal symbols in a word,
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is the *transition relation*. Here ϵ is a special symbol not in Σ; transitions on ϵ do not "consume" an input character,
- F, a subset of Q, is a set of *final states*

We can consider δ as defining a labeled transition graph with Q as the set of vertices, and $(\Sigma \cup \{\epsilon\})$ as the set of edge labels.

A *run* of the NFA $A = (Q, Q_0, \delta, \Sigma, F)$ on an input word w with length (i.e., number of characters) m is given by a sequence $r = (q_0, i_0); (q_1, i_1); \ldots$ which associates each position of the word with a state of Q. The sequence must satisfy the following constraints:

- the first position is 0, and the first state is initial. I.e., $i_0 = 0$ and $q_0 \in Q_0$.
- successive pairs are related by automaton transitions. I.e., for each k:
  - (epsilon-transition) $i_{k+1} = i_k$ and $q_{k+1} \in \delta(q_k, \epsilon)$, or
  - (input-transition) $i_{k+1} = i_k + 1$ and $q_{k+1} \in \delta(q_k, w(i_k))$

A run r is an *accepting run* for the input word w of length m if the final pair of the run, (q_n, i_n), is such that q_n is in F and $i_n = m$.

The Thompson algorithm can be seen as following all the runs of the NFA simultaneously on the given input word.

E. The matching algorithm for back-references

As described above, a regular expression with back-references may not describe a regular language. Thus, in general, regex's need not be representable by NFA's. Hence, one has to extend the NFA structure in some form. The extension we propose allows additional labels on the edges of the NFA. The labels indicate the start and end positions of a matching bracket, and the presence of a back-reference such as $\backslash 1$.

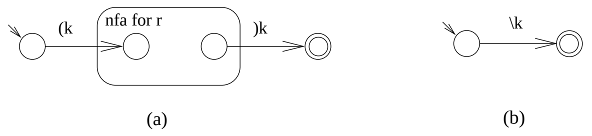

Fig. 5. Extended nfa for (a) the k'th bracket, (r) and (b) back-reference $\backslash k$.

Formally, a *backref-NFA* (NFA) is represented by a tuple $(B, Q, Q_0, \delta, \Sigma, F)$, where

- B is the number of back-references, numbered $1..B$,
- Q is a finite set of *states*,
- Q_0 is a subset of Q, defining the *initial states*,
- Σ is the *input alphabet*, the set of legal symbols in a word,
- $\delta : Q \times (\Sigma \cup \{\epsilon\} \cup \{(_k,)_k, \backslash k : 1 \leq k \leq B\}) \rightarrow 2^Q$ is the *transition relation*.
- F, a subset of Q, is a set of *final states*

The new transition symbols are $(_k$, which indicates the start of matching bracket k; $)_k$, which indicates the end of matching bracket k; and $\backslash k$, which indicates a back-reference k.

The construction of a backref-NFA from a regex follows Thompson's construction for NFA's, as described in [31]. The new cases are those for brackets and backreferences, and are given in Figure 5. A bracketed term, (r) generates an nfa with transition edges with "notes" that indicate the start and end of a bracket. A backreference term, $\backslash k$, generates a transition edge labeled by a "match" note to a unique successor. The construction results in an extended nfa of size linear in the length of the regular expression.

The definition of a run of a backref-NFA on an input word extends the earlier definition as follows. Each position in an input word is now associated with a *configuration*, rather than simply an NFA state. The set of configurations will be referred to as $\mathcal{C}$. Informally, a configuration records the current NFA state, substring information for already scanned brackets, and whether a particular back-reference is in the process of being matched. Formally, a configuration is a triple (q, μ, s), where

- q is a state of the backref-NFA,
- μ is a pair of partial functions, (μ_L, μ_R), both from $[1..B]$ to $[0..m-1]$. For any b, the pair $(\mu_L(b), \mu_R(b))$, if defined, indicates a substring of the input word which matches bracket b. We use $\perp$ to indicate a totally undefined function
- s is either $\mathsf{NOMATCH}$, or $\mathsf{MATCH}(b, i)$, where b is a bracket number in $1..B$, and i is an index.

A *run* of the backref-NFA $A = (B, Q, Q_0, \delta, \Sigma, F)$ on an input word w with length (i.e., number of characters) m is given by a sequence $r = (c_0, i_0); (c_1, i_1); \ldots$ which associates each position of the word with a configuration. The sequence must satisfy the following constraints:

- the first position is 0, and the first configuration is initial. I.e., $i_0 = 0$, and $c_0 = (q, \perp, \mathsf{NOMATCH})$, where $q \in Q_0$.
- successive pairs are related by automaton transitions as follows. If $(c = (q, \mu, s), i)$ and $(c' = (q', \mu', s'), i')$ are

successive pairs, then one of the following constraints must hold

- (epsilon-transition) $q' \in \delta(q, \epsilon)$, and $i' = i$, $\mu' = \mu$, and $s' = s$
- (input-transition) $q' \in \delta(q, w(i))$, $i' = i+1$, $\mu' = \mu$, and $s' = s$
- (start-bracket) This transition records the start of a bracket. For some b, $q' \in \delta(q, (_b)$, $i' = i$, μ'_L is μ_L extended with b mapped to i, and $\mu'_R = \mu_R$.
- (end-bracket) This transition records the end of a bracket. For some b, $q' \in \delta(q,)_b)$, $i' = i$,$\mu_L(b)$ is defined, and $\mu'_L = \mu_L$, μ'_R is μ_R extended with b mapped to i, and $s' = s$
- (begin-matching) $s = \mathsf{NOMATCH}$, $\mu(b)$ is defined, q has a single transition on $\backslash b$ for some k, and $s' = \mathsf{MATCH}(b, 0)$, $\mu' = \mu$, $q' = q$, $i' = i$
- (continue-matching) s has the form $\mathsf{MATCH}(b, p)$. Let $\mu(b) = (l, h)$. If $(l+p) < h$, and $w(l+p) = w(i)$ then $q' = q$, $s' = \mathsf{MATCH}(b, p+1)$, $i' = i, \mu' = \mu$
- (end-matching) s has the form $\mathsf{MATCH}(b, p)$. Let $\mu(b) = (l, h)$. If $(l+p) = h$, then q' is the unique successor of q on $\backslash b$, $s' = \mathsf{NOMATCH}$, and $i' = i, \mu' = \mu$

Matching a back-reference is done by a sequence of transitions. The "begin-matching" transition activates matching a back-reference b. The "continue-matching" transition matches the current input character $w(i)$ against the expected character $w(l+p)$ while keeping the state constant. The "end-matching" transition de-activates matching for b and moves the automaton state forward.

A run r is an *accepting run* for the input word w of length m if the final pair of the run, $(c_n = (q_n, \mu_n, s_n), i_n)$, is such that q_n is in F and $i_n = m$. The language of a backref-NFA is the set of input words for which there is an accepting run.

Theorem 1: Given a regex r, the augmented Thompson construction builds a backref-NFA A of size linear in the size of r, such that the language of A is precisely the set of words that satisfy r.

The deterministic matching algorithm is given in Figure 6. It follows the standard Thompson matching algorithm, with the replacement of NFA states by the more general configurations.

F. Complexity Analysis.

The number of possible configurations is the product of the number of states of the NFA, which we denote by K, and the number of possible maps, which is m^{2B}, where m is the length of the packet and B is the number of backreferences. Hence, the time taken is, at worst, $O(m * K * m^{2B})$. (Note that for the case of regular expressions ($B = 0$), this reduces to the usual $O(m * K)$.)

Although the complexity is polynomial in m and K, it is exponential in the number of back-references. While the number of backreferences is usually small, this expression is a weak worst-case estimate. Moreover, the matching problem for regex's is NP-complete when the number of backreferences

```
1. Convert regex r to backref-NFA A
2. For input word w:
   Let S = empty set
   Scan w from left to right.
   For each non-final position i:
     Let S1=S.
     For each configuration c in S, repeatedly
     add successors due to moves other than
     "input-transition" to S1
     Let S2=empty.
     For each configuration c in S1, add
     successors due to "input-transition"
     move for letter w(i) to S2
     Replace S with S2
3. Compute S1 from S as above. If S1 contains
   a configuration with a final NFA state,
   accept. Otherwise, reject
```

Fig. 6. Deterministic Regex Matching Algorithm.

is considered [6], so this exponential behavior should not be surprising.

G. Compile-Time Analysis

In this section, we describe two simple techniques which perform static analysis on the backref-NFA derived from a regex, to provide better bounds for the worst-case space and time complexity. The computed bounds can be used as an admissibility test, to separate out regex's which are potentially vulnerable to attacks for more detailed manual analysis.

1) Liveness Analysis: In the bound estimate described above, we implictly assume that each backreference entry in a configuration is useful for further matching. However, this may not be the case. Consider, for instance, the expression $(ab)\backslash 1(cd)\backslash 2$. After the use of $\backslash 1$, it is not needed for further matching. A similar situation is seen in the expression $(ab)\backslash 1|(cd)\backslash 2$, as the backreferences are used along disjoint union expressions.

In each case, while there are syntactically two backreferences in the expression, $\backslash 1$ is "dead" (i.e., unnecessary) after it is used to match for the first time. In expressions such as $(ab)(cd)\backslash 1; R; \backslash 2$, the match for $\backslash 1$ is not used in R, which might be a large and highly ambiguous regular expression. Thus, it is worthwhile, for performance reasons, to determine at compile-time those backreferences that are guaranteed to be "live" (i.e., useful) at each NFA state.

This sort of "liveness" analysis is a standard operation in optimizing compilers (cf. [31]), where it is used to determine which variables are still in use at a program point. We adapt the compiler algorithm to analyze the backref-NFA structure for live backrefs. The adaptation is the following:

- instead of a ”program flow graph”, we use the graph of the NFA
- instead of a ”basic block”, we use the NFA edges
- instead of variables, we track backreferences: An edge labeled with $\backslash b$ denotes a "use" of backreference b, while the edge labeled with $)_b$ forms a definition of backreference b.

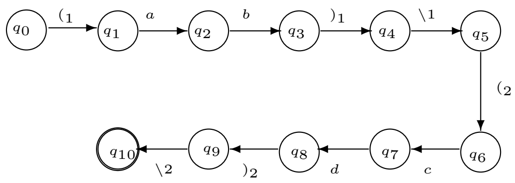

Fig. 7. Liveness for $(ab)\backslash 1(cd)\backslash 2$. Live sets: $q_4 = \{1\}, q_9 = \{2\}$; others are empty.

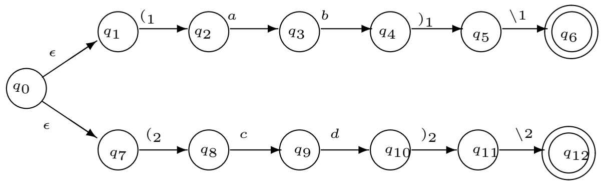

Fig. 8. Liveness for $(ab)\backslash 1|(cd)\backslash 2$. Live sets: $q_4 = \{1\}, q_{10} = \{2\}$; others are empty.

With this adaptation, a standard analysis for liveness generates the set of backreferences live at each NFA state. The result of running this algorithm is shown in Figures 7 and 8.

The computed information can be used in two ways.

- First, the analysis gives a better bound for the time/space consumption. If L is the maximum number of live backreferences (over all NFA states), then the worst-case time complexity can be tightened to $O(m * K * m^{2L})$.
- An optimization during matching. For a configuration (q, μ, s), entries in μ which refer to backreferences that are *NOT* live at q can be dropped to obtain the smaller configuration (q, μ', s). This reduction can result in two configurations which differ only on dead backreferences being merged into a single configuration, reducing the size of the current set.

2) Deterministic match analysis: Another assumption implicit in the bounds estimates is that a backreference can match arbitrary substrings of the input. This can happen if the matching is highly non-deterministic. For instance, the bracket in $a * (a*)a*$ can match anywhere in an input of the form a^n. However, it is sometimes the case that the backref-NFA for a pattern has the structure where the matching is purely deterministic, which can drastically reduce the match complexity bound.

An example is $(a*)b(c*)d; R; \backslash 1 \backslash 2$, where R is an arbitrary regular expression. Here, A_1 represents $(a*)b(c*)d$, and A_2 represents $R; \backslash 1 \backslash 2$. In this pattern, while there are two live back-references, the substring matching each backreference is chosen uniquely, thus, there is a unique map, rather than the potential of m^4 possible maps with two live back-references. Thus, the worst-case space complexity is $O(m * K)$.

3) Experiments on Snort Patterns: We ran the liveness analysis on 120 expressions with backreferences collected from Snort rules (after removing duplicate expressions arising from distinct rules). The resulting backref-NFA's have between 70 and 30,000 states, with most on the lower side.

Nearly all (111/120) of these expressions had a single backreference. For 8 of the expressions, however, the number of back-references is 6, while the maximum number of *live* backreferences at any state is only 2. This gives a drastic reduction in the rough worst-case complexity estimate. The remaining expression has 4 live backreferences, however, it has a deterministic match prefix, as in the case described above. Hence, the worst-case complexity is only linear.

The liveness analysis also shows that there is much variability between the number of live backreferences inside an NFA. For instance, in one of the 6 expressions where the max. live back-references is 2, the distribution is as follows: out of 638 states, 308 have no live back-reference, 192 have one, and the remaining have two. Thus, the actual matching complexity, using the optimization indicated previously, could be even lower than the tighter bound obtained through the liveness analysis.

IV. Experiments with Snort

Snort [33] is an open source network intrusion prevention system for performing real-time traffic analysis and packet logging on IP networks. It utilizes a flexible rule language to perform protocol analysis, content searching/matching and detection of a variety of attacks and probes, such as buffer overflows, stealth port scans, CGI attacks, SMB probes, OS fingerprinting attempts, etc. The number of rules and the complexity of rules has been increasing as more sophisticated attacks need to be expressed. In particular, regex's with back references are being used increasingly to express various attacks more precisely (see Figure 3).

The Snort IDS partitions rules into rule groups at compile time by the rule header, specifically, by the destination and source IP ports. As shown in Figure 4, each incoming packet is matched against one or two such port rule groups. For each rule within a rule group, Snort extracts a fixed string that must appear in a packet for that packet to successfully match the full rule. All such fixed strings are combined into a single DFA for each rule group as part of the Aho-Corasick string matching algorithm. This DFA efficiently determines the subset of rules that need to be fully matched against an incoming packet. At this point Snort invokes full rule matching, including the regular expression engine, for each of these remaining rules.

A. Modifying Snort

We modified the code in Snort that calls the pcre regexp matching library for regexp matching. In its place, we linked in as a library our new single scan algorithm that handles regular expressions as well as regex's. Parts of the new pattern matching algorithm were written in OCaml (http://caml.inria.fr) for quick prototyping, and then compiled into binary form.

We ran Snort on a 2.6GHz Linux PC. We used 25 packet traces collected during the "Capture the Flag" contest held at the hacker convention DefCon [34]: 13 (numbered 1 to 13) from Defcon 10 [35] and 12 (numbered 14 to 25) from Defcon 8 [36]. We also include one trace (numbered 26) which is

No.	Name	Source	# packets	# alerts	No.	Name	Source	# packets	# alerts
1	orange1.5.	Defcon10	21990	507	14	29122836	Defcon8	207631	437
2	orange1.6.	Defcon10	6971	329	15	29123907	Defcon8	202092	484
3	orange2.1.	Defcon10	164738	1051	16	29124449	Defcon8	523330	810
4	orange2.2.	Defcon10	37570	32	17	29132445	Defcon8	524968	277272
5	orange2.3.	Defcon10	84001	2684	18	29133000	Defcon8	18845100	979177
6	orange2.4.	Defcon10	6869	30	19	29142147	Defcon8	268153	833
7	orange2.5.	Defcon10	85156	6777	20	29163144	Defcon8	18467155	565926
8	orange2.6.	Defcon10	119159	1191	21	29170000	Defcon8	12160133	267556
9	orange2.7.	Defcon10	47113	654	22	29180000	Defcon8	22041078	145641
10	orange2.8	Defcon10	17867	74	23	29183001	Defcon8	12967247	186383
11	orange3.1	Defcon10	29989	87	24	29193000	Defcon8	921473 4	396084
12	orange3.2	Defcon10	154531	639	25	29200000	Defcon8	11188876	5859
13	orange3.3	Defcon10	121016	324	26	local	local WAN	2000000	2000

Fig. 9. Traces used in our Snort experiments.

collected locally from the link outside our firewall. The traces ranged from 6,800 to 2M packets.

A closer look at the poorly performing traces revealed that the PCRE NFA algorithm incurred an exponential search time on poorly written regular expressions, and finally returned with an error. Thus, not only was the performance reduced, but the IDS did not even successfully complete the matching. By selecting "bad" packets from these traces, the performance is reduced to nearly *1 packet per second!* When we replaced the backtracking pattern matching algorithm in Snort with our deterministic algorithm, the performance of the poorly performing traces was significantly improved. For the remaining traces, as expected, there was no significant change in performance.

The above traces have only a handful of packets that are matched against expressions with back-references. We isolated those packets, and measured the time to run a regex search using both the deterministic and backtracking algorithms. Figure 11 compares the time required by the two approaches to match backreference rules against packets. We see that the backtracking algorithm is consistently faster by a factor of two. This is partly due to the non-optimized ML implementation, but also because all of these packets match the expressions. Recall that when packets match a rule, the backtracking algorithm may detect the match earlier than the deterministic algorithm. For an IDS, this is the less common case of "bad" packets matching against rules. Also note that none of the patterns matched in Figure 11 caused exponential behavior during backtracking. As mentioned in Section III, it is hard to predict whether or not such behavior is possible in the worst case.

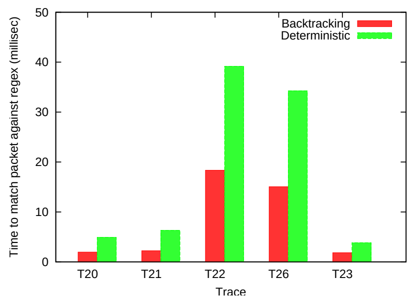

Fig. 11. Times to match a regex (with one or more back references). Traces labeled by the number of the original trace from which packet flows were extracted.

V. Conclusion and Future Work

A powerful and expressive rule language is necessary for an IDS/IPS system to defend the network against complex new attacks. But this power comes at a high cost, especially since the rule writer is typically unaware of the performance consequences of matching packets against complex rules. We have seen a number of poorly written rules in the popular Snort IDS, some of which are simply incorrect, while the rest can be very hard to match efficiently. Therefore, it is important for the IDS/IPS itself to analyze new rules and ensure that there are no resulting performance vulnerabilities, even in the presence of worst-case packet inputs.

We have shown that precise analysis for exponential matching behavior is impossible. Therefore, we propose an alternative approach: use a deterministic matching algorithm, check all rules for worst-case matching performance, and expose the few rules that are suspected to have performance vulnerabilities. We presented static analysis solutions to perform this rule-by-rule analysis. The intent is that rule writers can use this analysis as an admissibility test to check new rules and ensure that their IDS/IPS will not fall prey to a DoS attack.

References

[1] R. Cox, "Regular expression matching can be simple and fast," http://swtch.com/ rsc/regexp/regexp1.html.
[2] ——, "Implementing regular expressions," http://swtch.com/ rsc/regexp/.
[3] S. Crosby, "Denial of service through regular expressions," usenix Security Symposium Work-in-Progress, http://www.usenix.org/events/sec03/wips.html.
[4] K. Thompson, "Regular expression search algorithm," *Commun. ACM*, vol. 11, no. 6, pp. 419–422, 1968.
[5] T. S. S. Alert, "Buffer overflow on procedures of the replication management api packages," http://www.appsecinc.com/resources/alerts/oracle/2004-0001/25.html.
[6] A. V. Aho, "Algorithms for finding patterns in strings," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*. Elsevier, 1990, pp. 255–300.
[7] L. Tan, B. Brotherton, and T. Sherwood, "Bit-split string-matching engines for intrusion detection and prevention," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 1, pp. 3–34, 2006.

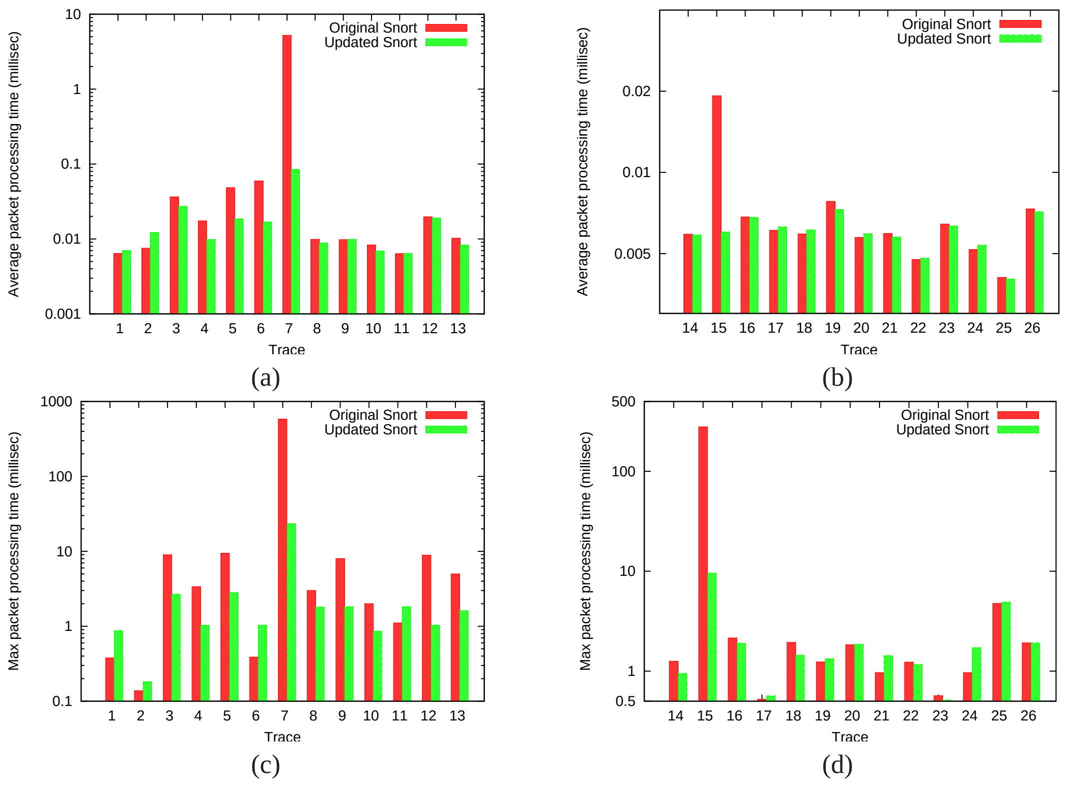

Fig. 10. (a),(b): Average, and (c),(d): maximum packet processing time by Snort for each of the 26 traces. The red bars represent the performance of the original Snort IDS, while the green bars represent the performance of Snort updated with the deterministic matcher.

[8] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, and S. Vassiliadis, "Packet pre-filtering for network intrusion detection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 183–192.

[9] D. L. Schuff and V. S. Pai, "Design alternatives for a high-performance self-securing ethernet network interface," in *IPDPS*, 2007, pp. 1–10.

[10] F. Yu, R. Katz, and T. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *ICNP*, 2004.

[11] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *FCCM*. IEEE Computer Society, 2004, pp. 249–257.

[12] Brodie, Cytron, and Taylor, "A scalable architecture for high-throughput regular-expression pattern matching," *CANEWS: ACM SIGARCH Computer Architecture News*, vol. 34, 2006.

[13] A. Ghosh and J. Voas, "Inoculating software for survivability," *Commun. ACM*, vol. 42, no. 7, pp. 38–44, 1999.

[14] W. Drewry and T. Ormandy, "Insecure context switching: inoculating regular expressions for survivability," in *WOOT'08: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*.

[15] K. Ellul, B. Krawetz, J. Shallit, and M. Wang, "Regular expressions: new results and open problems," *J. Autom. Lang. Comb.*, vol. 9, no. 2-3, pp. 233–256, 2004.

[16] V. Laurikari, "Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions," pp. 181–187, 2000, http://laurikari.net/ville/spire2000-tnfa.ps.

[17] T. M. Corporation, "Can-2005-2491," August 2000, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2005-2491.

[18] grep(1), "In gnu project manual pages," 2002.

[19] J. Friedl, *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[20] W. Lee, J. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang, "Performance adaptation in real-time intrusion detection systems," in *RAID*, 2002, pp. 252–273.

[21] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *CONEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pp. 1–12.

[22] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. SIGCOMM*, 2006, pp. 339–350.

[23] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM/IEEE ANCS*, 2007, pp. 145–154.

[24] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. INFOCOM*, 2007.

[25] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM Conference on Emerging Network Experiment and Technology, CoNEXT*, 2007.

[26] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *Proc. ACM SIGCOMM*, 2008.

[27] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*.

[28] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[29] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proc. of the 12th USENIX Security Symposium*, 2003, pp. 29–44.

[30] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a nids," in *ACSAC*, 2006.

[31] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools, Second Edition*. Addison-Wesley, 2007.

[32] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, Massachusetts: Addison-Wesley, 1979.

[33] I. Sourcefire, "Snort - the de facto standard for intrusion," http://www.snort.org/.

[34] DefCon.org, "Defcon convention," http://www.defcon.org/.

[35] ——, "Defcon 10 convention, capture the flag packet traces," http://cctf.shmoo.com/data/cctf-defcon10/orange.cctf.tar.gz.

[36] ——, "Defcon 8 convention, capture the flag packet traces," http://cctf.shmoo.com/data/cctf-defcon8/.