A Novel Algorithm for Pattern Matching with Back References

Liu Yang
Baidu, Inc.
Shenzhen, China 518000
Email: yangliu11@baidu.com

Vinod Ganapathy
Rutgers University
Piscataway, NJ 08854, USA
Email: vinodg@cs.rutgers.edu

Pratyusa Manadhata
Hewlett Packard Laboratories
Princeton, NJ 085421
Email: manadhata@hpe.com

Ye Wu
Baidu, Inc.
Shenzhen, China 518000
Email: wuye01@baidu.com

***Abstract*—Modern network security applications, such as network-based intrusion detection systems (NIDS) and firewalls, routinely employ deep packet inspection to identify malicious traffic. In deep packet inspection, the contents of network traffic are matched against patterns of malicious traffic to identify attack-carrying packets. The pattern matching algorithms employed for deep packet inspection must be fast, as the algorithms are often implemented on middle-boxes residing on high-speed gigabits per second links. The majority of patterns employed in network security applications are regular languages. However, regular language-based patterns have limited expressive power and are not capable of describing some complex features in network payload. Back reference is an important feature provided by many pattern matching tools, e.g., PCRE, the regular expression libraries of Java, Perl, and Python. Back references are used to identify repeated patterns in input strings. Patterns containing back-references are non-regular languages. Very little work has been done to improve the time-efficiency of back reference-based pattern matching. The de facto algorithm to implement back reference is recursive backtracking, but it is vulnerable to algorithmic complexity attacks. In this paper, we present a novel approach to implement back references. The basic idea of our approach is to transform a back reference problem to a conditional submatch problem, and represent it with a Non-deterministic Finite Automata (NFA)-like machine subject to some constraints. Our experimental results show that our approach resists known algorithmic complexity attacks, and is faster than PCRE by up to three orders of magnitude for certain types of patterns.**

***Index Terms*—Pattern matching; Back reference; Finite automaton; Network-based Intrusion Detection System.**

I. Introduction

Network security applications, e.g., network-based intrusion detection systems (NIDS) and firewalls, perform deep packet inspection to identify malicious traffic. In deep packet inspection, the contents of network traffic are matched against patterns of malicious traffic to identify attack-carrying packets. In the past, patterns were represented by keywords that could be efficiently matched using string matching algorithms, e.g., KMP [11], Boyer-Moore [4], Wu-Manber [24], and Aho-Corasick [2]. The increasing complexity of network attacks has lead the community to employ more expressive representations, which require the full power of regular expressions.

Strictly speaking, regular expressions denote patterns that can be described by regular languages. However, this term has been extended to represent patterns that have non-regular language features. Among these features, *capturing groups* and *back references* are two important ones. A capturing group is used to specify a sub-expression of a regular expression, and a back reference denotes a repeated sub-expression in a regular expression. Many pattern matching tools, e.g., PCRE [16], the regular expression libraries of Java, Perl, and Python, support capturing groups and back references. Patterns containing back references are non-regular languages [7].

Patterns with back references are more expressive than regular languages. For example, suppose we want to match a pair of XML tags and the text in between. It will be hard to represent this pattern if we are only allowed to use regular languages because tags in an XML file may be unknown beforehand. In this case, a back reference can easily describe the pattern. For example, "`<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>`" can be used to match a pair of XML tags and the text in between, where the first capturing group (subexpression within the pair of parentheses) is used to capture an XML tag, and the "`\1`" denotes that the captured tag will be reused at the end (before the '>' symbol) of the pattern. A pattern can have multiple back references, where each of them refers to a different capturing group. Multiple back references can be sequentially named by a '`\`' followed by different numbers. For example, three back references can be named as "`\1`", " `\2`", and "`\3`". One back reference can also appears multiple times in a pattern, e.g., "`([a-c])x\1x\1`". Back references are also employed by modern NIDS to represent attack signatures. For example, the HTTP rule set of Snort 2012 has 167 patterns containing back references [21].

Since patterns containing back references are non-regular languages, they cannot be represented by finite automata, i.e., non-deterministic finite automata (NFAs) or deterministic finite automata (DFAs), as finite automata are equivalent representations of regular languages. Thus, prior approaches on NFAs or DFAs could not be applied to back references. In fact, very little work has been done for patterns containing back references. As pointed out by Cox [6], "No one knows how to implement pattern with back references efficiently, though no one can prove that it's impossible either". Specifically, the back reference problem is NP-complete [1]. The de facto algorithm for back references is recursive backtracking.

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

However, recursive backtracking is vulnerable to algorithmic complexity attacks [18]. For example, the throughput of PCRE quickly decreases to nearly zero mega-byte/second for patterns in the form of "`(a?{n})a{n}\1`" ($n = 5, 10, 15, 20, 25, 30$) with input strings in the form of a^n (i.e., a is repeated n times). In fact, we observed that PCRE fails to return correct results for $n \geq 25$ on a Linux machine with a typical hardware configuration. Can we find an approach that can address back references but resist known algorithmic complexity attacks? In this paper, we explore the answer to this question.

A. Our Contribution

We present a novel approach to implement pattern matching with back references. The basic idea of our approach is to transform a back reference problem to a *conditional submatch problem*, and represent a conditional submatch problem using an NFA-like machine subject to some constraints. We evaluate the feasibility of our approach with a software-based implementation, using both synthetic patterns and patterns from real-world NIDS. Our experimental results show that our approach resists known algorithmic complexity attacks and is faster than PCRE by three orders of magnitude for certain types of patterns.

The remainder of this paper is organized as follows. Section II provides some background of the problem. Section III presents the design of our algorithm for patterns with back references. Sections IV and V present the implementation and the experimental evaluation of our approach, and Section VI discusses the related work. Section VII concludes our work.

II. Background

A. Finite Automata and Regular Expressions

Finite automata are natural representations for regular expressions. It is known that regular expressions, deterministic finite automata (DFAs), and non-deterministic finite automata (NFAs) are equivalent in terms of expressive power. Therefore, regular expression matching can be performed by operating the corresponding NFAs or DFAs. Given a regular expression, we can use Thompson's algorithm [23] to construct an NFA that recognizes the same language as the given regular expression. An NFA can be converted to a DFA that recognizes the same language using the subset construction algorithm [9]. For a regular expression of length m, with an input string of length n, the time complexities of the DFA-based algorithm and Thompson's NFA-based algorithm are $O(n)$ and $O(m \times n)$ respectively. However, their space complexities are $O(2^m)$ and $O(m)$. In other words, DFA-based algorithms are time efficient but space inefficient; NFA-based algorithms are space efficient, but often much slower than DFA-based algorithms [7].

B. Recursive Backtracking-based Matching

Another way to simulate an NFA is using recursive backtracking. The algorithm operates in a depth-first-search style. For a current state with the ith symbol in an input string, the algorithm processes all states in the next set of states in a depth-first-search way. A recursive backtracking algorithm may have to scan an input string multiple times before it finds a match. Tools like PCRE and the regular expression libraries in many high level languages such as Java, Perl, and Python implement pattern matching using recursive backtracking. As it was pointed out by Cox, recursive backtracking based matching can be extremely slow in some cases [7].

C. Algorithmic Complexity Attack

As we described in Section I, recursive backtracking is the de facto implementation of back references. However, a recursive backtracking matching algorithm can be extremely slow in certain cases, as is shown by an example below.

Figure 1 shows the process of using recursive backtracking algorithm to match the pattern "`host.*com.*uuid=.*wv=.*cargo`" with the following string that has 45 characters:

"`hostcomhostcomhostcomuuid=uuid=uuid=wv=wv=wv=`"

We denote the five parts separated by "`.*`" in the pattern by P1, P2, P3, P4, and P5 respectively, i.e., P1="`host`", P2="`com`", etc. A number on an edge between two nodes in the figure denotes an offset where a subexpression Pi($i = 1 \ldots 5$) is matched in the input string. For example, the leftmost edge between P1 and P2 is labeled by 3, which means that "`host`" is matched by the input string at offset 3. The above pattern is matched by an input string if and only if P1, P2, P3, P4, and P5 are sequentially matched by the input string. It can be observed that a backtracking approach needs to try 45 paths for the input string before it can claim that the example pattern is not matched by the example input string. In general, for a pattern that has k parts separated by wildcard characters "`.*`", the running time of a backtracking algorithm can be close to $O(n^k)$ [18], where n is the length of the input string. Such a behavior that triggers a backtracking algorithm to exhaustively try all execution paths for input strings is called the *Algorithmic Complexity Attack*. Researchers have demonstrated that the throughput of a NIDS employing recursive backtracking for pattern matching can be slowed down by several orders of magnitude under Algorithmic Complexity Attacks [18].

III. Design of Our Algorithm

The basic idea of our approach to address back reference is to transform a back reference problem to a conditional submatch problem, and represent the conditional submatch problem using an NFA-like machine subject to some constraints. Our approach includes two phases: *compilation* and *execution*. During the compilation phase, patterns with back references are compiled to tagged-NFAs subjected to some constraints. During the execution phase, pattern matching is performed by operating the tagged-NFAs generated at the compilation phase with input strings.

A. Pattern Compilation

We introduce a *relax plus constrain* approach to tackle the back reference problem. The compilation process is shown in

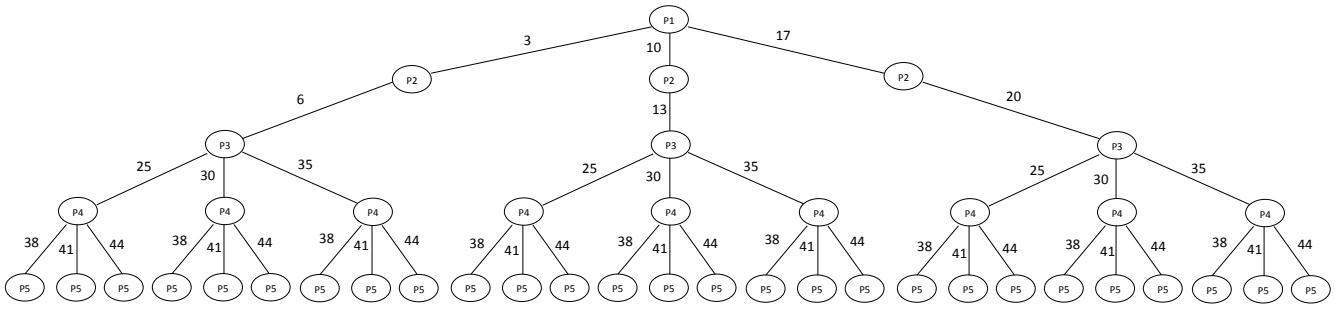


Fig. 1: An example path tree traversed by the recursive backtracking agorithm.

Algorithm 1. Relax refers to re-writing a regular expression with back references to a regular expression that only contains capturing groups. During re-writing, a back reference part is replaced by the capturing group that is referred by the back reference. By doing this, a back reference and its referred capturing group become a pair of capturing groups in the re-written regular expression. To make the re-written pattern be equivalent to the original pattern, we add a constraint to the accept condition such that the submatches returned by the capturing group pair are equal. The re-writing operation is shown in line 1 of Algorithm 1, where p is the re-written expression and C denotes the constraint added to the accept condition. For example, pattern "`(a*)aa\1`" can be re-written as "`(a*)aa(a*)`" with the constraint such that "`$1=$2`", where "`$1`" and "`$2`" denote the first and second submatches captured by the two capturing groups.

Once a pattern with one or more back references is transformed to a pattern of conditional submatch extraction, we can construct an NFA-like machine (a tagged-NFA) that is equivalent to the converted pattern using a Thompson's like algorithm. The algorithm starts from three basic cases: tagged-NFAs of ϵ expression, empty expression, and a symbol wrapped by a capturing group. More complex tagged-NFAs can be constructed using the union, concatenation, and closure constructs. The construction process bears some similarity to constructing a tagged-NFA described in [27]. As we will see soon, the difference lies in how to operate a tagged NFA. Recall that we add a equal substrings constraint to a re-written pattern in order to make it equivalent to its original pattern. Thus, we need a mechanism to maintain substrings matched by the capturing groups in a re-written pattern. To do so, a tagged NFA needs to have a data structure allowing for bookkeeping of captured substrings. The data structure we use is to associate each state with a pair of substrings (multiple pairs of substrings are needed if there are multiple back references). For transitions within a capturing group, we add the corresponding input symbols into captured substrings. For transitions that are not within any capturing group, we just carry over the captured substrings from state to state. When a tagged-NFA reaches a final state, we check whether there

Algorithm: Pattern Compilation
Input : A pattern r
Output : A tagged-NFA with constraints
1 (p, C)=re-write(r);
2 $(Q, \Sigma, T, \delta, q_0, Fin)$ = compile(p);
3 return $((Q, \Sigma, T, \delta, q_0, Fin), C)$;

Algorithm 1: **An algorithm to compile a pattern with back references.**

exists a pair of equal captured strings. If so, an input string is matched by the tagged-NFA. To be more formal, we denote a tagged NFA as a 6-tuple $(Q, \Sigma, T, \delta, q_0, Fin)$, where Q is the state set, Σ is an alphabet set, T is a tag set, q_0 is the start state, Fin is a set of final states, and δ is a transition function $\delta : Q \times \Sigma^* \times \Sigma^* \times \cdots \rightarrow Q \times \Sigma^* \times \Sigma^* \times \ldots$ that maps a current state with the captured substrings to a next state with updated captured strings. The Thompson's-like process to construct a tagged-NFA is described in line 2 of Algorithm 1.

We now demonstrate the compilation process using the example pattern "`(a*)aa(a*)`". After adding tags, the pattern is denoted as "$(a*)_{t_1}aa(a*)_{t_2}$", where t_1 to t_2 are used to label the two capturing groups. Figure 2 shows a tagged-NFA constructed from pattern "$(a*)_{t_1}aa(a*)_{t_2}$" such that "`$1=$2`". It can be observed that the transition from state 1 to itself with input symbol '`a`' is within the first capturing group, and the transition from state 3 to itself with input symbol '`a`' is within the second capturing group. All other transitions are not in any capturing group.

Similar to traditional NFAs, we can use a transition table to represent the transitions of a tagged-NFA. Instead of having three columns, the transition table of a tagged-NFA has five columns, where the first three columns are same as those in a traditional transition table, the fourth column denotes tags associated with each transition, and the fifth column specifies the actions used for maintaining the substrings matched by capturing groups. In our design, we have three types of actions: *new*, *update*, and *carry over*, where *new* and *update* actions are associated with transitions within capturing groups, and a *carry over* action is associated with transitions not in any

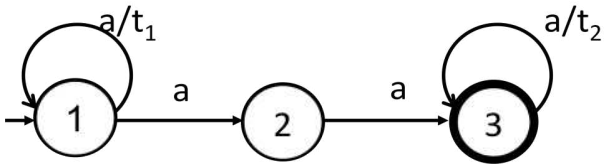


Fig. 2: The tagged-NFA constructed from "(a*)aa(a*)". The start state is labeled by 1, and the accept state is labeled by 3.

Algorithm: Execution
Input : An input string str to be matched, and a tagged-NFA with constraints $((Q, \Sigma, T, \delta, q_0, Fin), C)$ compiled from a pattern.
Output : true or false

1 $current\_states = \{(q_0, \text{“”}, \text{“”}, ...)\}$;
2 **for** i=1 **to** $strlen(str)$ **do**
3 $next\_states$ = {};
4 **foreach** $(s, substr_1, substr_2, \dots) \in current\_states$ **do**
5 $next\_states$ = $next\_states \cup \delta((s, substr_1, substr_2, \dots), str[i])$;
6 **if** (i==$strlen(str)$) and $(\exists(x, substr1, substr2, ...) \in next\_states)$ $s.t.$ $(x \in Fin)$ and $C(substr1, substr2, ...) = true$ **then**
7 return true;
8 $current\_states$ = $next\_states$;
9 return false;

Algorithm 2: **The execution of a compiled pattern.**

capturing group. A *new* action denotes creating a new captured substring using a current input symbol, and an *update* action denotes updating a substring by appending an input symbol to the end of the substring. A *carry over* action denotes that captured substrings are copied over from a current state to a next state. Table I shows the transition table of the tagged-NFA in Figure 2.

B. Execution

We now describe how to match a pattern that has back references with an input string. The matching process is called *execution* of a compiled pattern and it mainly involves two operations: *frontier derivation* and *acceptance checking*. Frontier derivation refers to how to update the current states of a tagged-NFA with an input symbol. Acceptance checking refers to checking whether the tagged-NFA is in an accept state. The matching algorithm is shown in Algorithm 2, where lines 4-5 describe the frontier derivation, and line 6 describes the acceptance checking operation.

1) Frontier Derivation: To allow for the maintenance of captured substrings, we denote an element in a frontier set by a tuple $(x, substr_1, substr_2, \dots)$, where x is a state number, and $substr_1$ and $substr_2$ are substrings matched by the capturing groups. In general, if there are k back references, we need a $(2k+1)$-tuple to represent a frontier element. During a match test of an input string, the frontier set is initially a singleton set $\{(q_0, \text{“”}, \text{“”}, ...)\}$ shown in line 1 of Algorithm 2 (where “” is an empty string denoting that no substring has been captured yet) but may include multiple elements during the operation of a tagged-NFA. For each symbol in the input string, we must process all elements in a frontier set and find a new set of elements by applying the transition functions represented by the transition table (lines 4-5 in Algorithm 2). Applying a transition function to a frontier element $(s, substr_1, substr_2, \dots)$ and an input symbol includes two steps. The first step is a table lookup, i.e., given a state s and symbol $I(i)$, retrieve all states that are reachable from s with symbol $I(i)$. The second step is to apply one or more actions to the captured substrings associated with the state s. In particular, if a transition is a start of a capturing group, then a *new* action is applied; if a transition is within a capturing group then an *update* action is applied; and if a transition is not within any capturing group, then a *carry over* action is applied by just copying around the captured substrings (if there is any) from a current state to a next state. For a pattern with one back reference, the above frontier derivation process can be expressed by the following Boolean formula:

$$\begin{aligned}\mathcal{G}(\boldsymbol{y}, \boldsymbol{s}) = F_0(\exists\, \boldsymbol{x} \cdot \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{t} \cdot (\boldsymbol{t} = \phi \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t}))) \\ \vee F_1(\exists\, \boldsymbol{x} \cdot \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{t} \cdot (\boldsymbol{t} = t_1 \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t}))) \\ \vee F_2(\exists\, \boldsymbol{x} \cdot \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{t} \cdot (\boldsymbol{t} = t_2 \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})))\end{aligned}$$

where

$$\Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t}) = \mathcal{F}(\boldsymbol{x}, \boldsymbol{s}) \wedge \mathcal{I}_{\sigma}(\boldsymbol{i}) \wedge \Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t}) \quad (1)$$

$\mathcal{F}(\boldsymbol{x}, \boldsymbol{s})$ denotes the current frontier set ($\boldsymbol{s}$ denotes captured substrings), $\mathcal{I}_{\sigma}(\boldsymbol{i})$ denotes an input symbol, and $\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ denotes the transition relations of the tagged-NFA. The conjunctions in Equation (1) basically selects rows in the transition table $\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ that corresponding to outgoing transitions from the states in the current frontier set $\mathcal{F}(\boldsymbol{x}, \boldsymbol{s})$ labeled with symbol σ. The $\boldsymbol{t} = \phi \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ in $\mathcal{G}(\boldsymbol{x})$ selects transitions that are not in any capturing group, $\boldsymbol{t} = t_1 \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ selects transitions that are labeled by t_1 (first capturing group), and $\boldsymbol{t} = t_2 \wedge \Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{s}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ selects transitions labeled by t_2 (second capturing group). Function F_0 denotes a *carry over* action; function F_1 denotes applying a *new* or *update* action to substrings captured by the first capturing group; and function F_2 denotes applying a *new* or *update* action to substrings captured by the second capturing group. Renaming the $\boldsymbol{y}$ to $\boldsymbol{x}$ in $\mathcal{G}(\boldsymbol{y}, \boldsymbol{s})$ gives us the new frontier set $\mathcal{G}(\boldsymbol{x}, \boldsymbol{s})$. The frontier derivation formulae for patterns with multiple back references are similar, except that more tags $t_i (i = 1, 2, \dots)$ are involved.

Current state(x)	Input symbol(i)	Next state(y)	Tag(t)	Action
1	a	1	t_1	$new(t_1)$ or $update(t_1)$
1	a	2	ϕ	$carry\_over(t_1)$
2	a	3	ϕ	$carry\_over(t_1)$
3	a	3	t_2	$new(t_2)$ or $update(t_2)$

TABLE I: Transition table of the tagged-NFA in Figure 2.

Example Consider the example tagged-NFA in Figure 2 with input string "aaaa". Initially, the frontier set is a singleton set {(1, "", "")}. For the first input symbol 'a', we can get that the next state can be state 1 or 2 according to the transition table in Table I. The fourth column of the transition table indicates that the transition from state 1 to 1 is associated with a $new(t_1)$ function, which means we need to create a new substring for the first capturing group using the current input symbol 'a'. The transition from state 1 to 2 is associated with a $carry\_over(t_1)$ action. Since no substring has been captured in (1, "", ""), nothing needs to be copied from state 1 to state 2. As a result, the new frontier set has two elements, i.e., {(1, "a", ""), (2, "", "")}.

Renaming {(1, "a", ""), (2, "", "")} as the current frontier set, with the second input symbol 'a', we can obtain the next frontier set as {(1, "aa", ""), (2, "a", ""), (3, "", "")}. Using the same method to process the third and fourth input symbols. After processing the fourth input symbol 'a', the frontier set is {(1, "aaaa", ""), (2, "aaa", ""), (3, "aa", ""), (3, "a", "a"), (3, "", "aa")}.

2) Acceptance Checking: The accept condition of a tagged-NFA is: at the end of an input string, there exist a tuple $(x, substr_1, substr_2, \dots)$ in the frontier set such that $x \in Fin$ is a final state, and $substr_1$ equals $substr_2$ (for patterns with one instance of back reference). If there are k different back references, we need to have k pairs of captured substrings, where the two substrings in each pair are equal (shown as $C(substr1, substr2, ...) = true$ in line 6 of Algorithm 2). For the example tagged-NFA with input string "aaaa", it can be observed that there is one element, i.e., (3, "a", "a"), in the frontier set satisfying the acceptance condition after processing the fourth input symbol 'a'. Therefore, the input string "aaaa" is accepted by the tagged-NFA, which means the input string matches pattern `(a*)aa\1`.

Remarks: We note that our approach for back reference can be employed to do submatch extraction as well. In that case, nothing needs to be added as constraint to a tagged-NFA. The main benefit is that this approach is capable of performing pattern matching and submatch extraction by just scanning the input string in a single pass.

IV. Implementation

We evaluated our approach using a software-based implementation, dubbed as NFA-backref. The implementation is in C++ and has two components: a *compilation* component and an *execution* component, as shown in Figure 3. The compilation component reads patterns with back references and compiles them into tagged-NFAs with constraints as described in Section III-A. The execution component loads compiled patterns (tagged-NFAs) and matches them with a stream of input strings. In our implementation, captured substrings are represented by their starting and ending offsets in the input strings. In this way, substrings do not have to be copied around from states to states. Each substring is represented using only two integers, which saves space and reduces overhead of string copy operation.

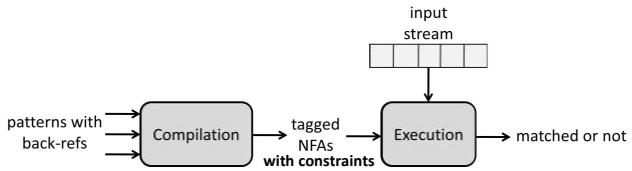


Fig. 3: The overview of our software-based back reference implementation.

V. Evaluation

We evaluated the performance of our approach using both synthetic patterns and patterns from real-world NIDS. Our experimental results show that our approach is immune to Algorithmic Complexity Attacks.

A. Experimental Setup

All our experiments were performed on a Intel Core2 Duo E7500 Linux-2.6.27 machine, running at 2.93GHz with 2GB of RAM (however, our programs are single-threaded, and only used one of the available cores). We instrumented the matching tool to report its execution time using processor performance counters. We report the performance of execution as the number of CPU cycles to process each symbol (cycles/byte), i.e., fewer processing cycles/byte implies greater time-efficiency.

B. Data Sets

We evaluated the performance of different implementations using the following two data sets:

a) Patho-01: Patterns in this data set are in the form of `(a?{n})a{n}\1`, where the `?` character is a 0 or 1 quantifier. This pattern will match a string starting with zero or one 'a' repeated by n times, followed by n characters of 'a', followed by the substring captured by the first capturing group. We evaluated the pattern for $n = 5, 10, 15, 20, 25$, and 30. For each pattern, we use input string in the form of a^n, i.e., 'a' repeated by n times, which will be matched by a pattern with the same value of n.

b) Snort-46: The second data set includes 46 patterns containing back references from the Snort HTTP signature set. We use two input traces to evaluate this pattern set. The first trace, which we call *benign trace*, was generated using a string generator created by ourselves. Given a set of patterns and a user expected match percentage p, the string generator generates a trace file where p percent of strings are matched by at least one pattern in the pattern set. The size of the benign trace we generated in our evaluation is 5MB. The second trace was manually crafted after carefully reviewing the 46 patterns. We found that at least one of these patterns will suffer from the Algorithmic Complexity Attacks if a pattern matching engine is implemented using the recursive backtracking approach. We thus manually created a 100KB *pathological* trace using the approach described in [18] to evaluate how different implementations perform under an Algorithmic Complexity Attack.

C. Performance

We measure the time efficiency of different implementations using the number of CPU cycles required for processing each byte of input trace (cycle/byte). We evaluate the performance of two implementations: Our approach (NFA-backref), and PCRE using the data sets described in Section V-B. We did not measure the space efficiency of different implementations since both NFA-based approach and recursive backtracking are space efficient, as presented in [25] and [27].

Figure 4 shows the execution time of different implementations for the Patho-01 data set. The x-axis denotes the value of n in pattern `(a?{n})a{n}\1`, and the y-axis denotes the execution time in unit of cycle/byte. It can be observed that PCRE is a slower implementation as n increases from 5 to 30. NFA-backref is faster than PCRE by at least three orders of magnitude (i.e., 1000+ times) after $n \geq 25$.

As shown in Figure 4, PCRE performs extremely slow for this pattern set. This is mainly because that PCRE performs exhaustive recursive backtracking when matching an input string a^n (i.e., 'a' repeated n times) against pattern `(a?{n})a{n}\1`. During a recursive backtracking, the first matching path that is tried by PCRE is to match the n characters of 'a' with the `(a?{n})` part of the pattern. This path will fail because there is no characters to match the remaining part `a{n}\1`. Then PCRE will backtrack one step and use $n-1$ characters to match the `(a?{n})` part and will fail again. Continue this way, it needs to traverse $O(2^n)$ paths before it finally succeeds by using zero 'a' to match the `(a?{n})` part, n characters of 'a' to match `a{n}` , and zero 'a' to match the back reference part `\1`. As the value of n gets large, the number of traversal paths increases exponentially, which will cause PCRE to abort the backtracking process when the size of the stack is too large. In our experimentation, we observed that PCRE failed to give correct matching results when $n \geq 25$, while our implementation always returns correct results for all input traces. The failure of PCRE for patterns when $n \geq 25$ is mainly due to that PCRE aborts the recursive backtracking when the size of recursive stack is over a threshold.

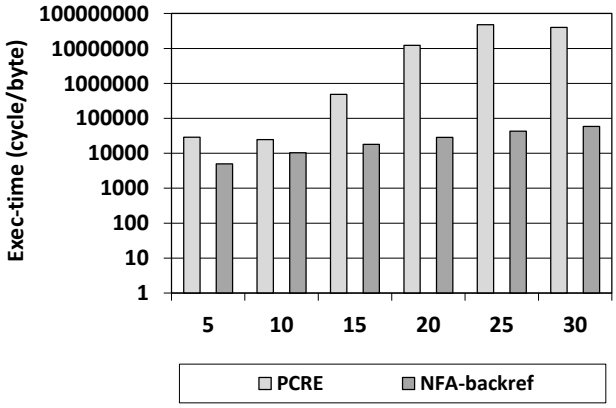
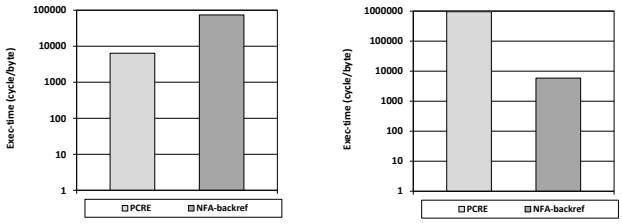


Fig. 4: Execution time (cycles/byte) of different implementations for the Patho-01 data set, the smaller the better. It can be observed that NFA-backref resists the Algorithmic Complexity Attack and it is at least three orders of magnitude (1000+ times) faster than PCRE for $n \geq 25$.



(a) Execution time with benign trace

(b) Execution time with pathological trace

Fig. 5: Execution time (cycles/byte) of different implementations for the Snort-46 pattern set, the smaller the better. NFA-backref outperforms PCRE by two to three orders of magnitude when the pathological trace is used as input.

Figure 5 shows the execution time of different implementations for the Snort-46 data set. Figure 5a shows the performance with the benign trace, and Figure 5b shows the performance with the pathological trace. It can be observed that PCRE is about one order of magnitude faster than NFA-backref when the benign trace is used as input strings. However, NFA-backref is two to three orders of magnitude faster than PCRE for the pathological trace. The low performance of PCRE in Figure 5b is due to that the pathological trace triggers PCRE to do exhaustive recursive backtracking during pattern matching.

Both Figure 4 and Figure 5 show that our approach (NFA-backref) is immune to the Algorithmic Complexity Attack. Under pathological traces, our NFA-backref implementation outperforms PCRE by orders of magnitude. Although NFA-

backref is slower than PCRE for benign traces, we argue that NFA-backref is a better implementation because network security tools, e.g., NIDS, are often exposed to attacking network traffic, in which an attacker may deliberately craft pathological network contents to perform a DoS attack to a recursive backtracking-based pattern matching engine. Thus, we believe that our approach is better suited to be deployed to process hostile network traffic.

VI. Related Work

Pattern matching in practice demonstrates a time/space tradeoff. DFA-based approaches are time efficient, but suffer from state blow-up. NFA-based approaches are space efficient, but are slow in operation. Recursive backtracking-based approach is fast in general, but can be orders of magnitude slower under an Algorithmic Complexity Attack. The time/space tradeoff has spurred a lot of recent research, primarily focused on patterns that can be described by regular languages (regular expressions). Many researchers aimed at reducing the memory foot prints of DFA-based approaches [22], [28], [12], [19], [20], some researchers worked on improving the time efficiency of NFA-based approaches with hardware [10], [14], [5], [17] or software solutions [25], [26].

Patterns used in real-world security tools are often regular expressions with extended features. One of the important features, submatch extraction, is discussed in [27]. Another important one, back reference is discussed in this paper. Up to now, not much work has been done on submatch extraction and back reference. Existing approach on submatch extraction include Google's RE2 [7], Haber et al.'s DFA-based algorithm [8], Laurikari's tagged-NFA approach [13], and Yang et al.'s Submatch-OBDD model [27]. However, RE2 does not support back references. Recursive backtracking is the de facto approach to implement back references and has been adopted by tools such as PCRE and regular expression libraries in many high level languages such as Java, Python, and Perl [6]. As we have shown, a recursive backtracking based implementation suffers from the Algorithmic Complexity Attacks. Becchi and Crowley proposed to model a back reference problem with an automaton-like machine [3]. Their approach constructs a special state for each back reference instance. Substrings are recorded in a back reference state and are matched in a *consuming* way. Becchi's approach works in the situation when there is only one back reference instance for a capturing group but fails when there are multiple back reference instances for a same capturing group. Also, it is not clear how Becchi's approach performs because no execution time was reported in their paper. Namjoshi and Narlikar presented an automaton-based back reference approach [15] similar to [3]. Our approach differentiates from [3] and [15] in that we do not construct special states or input symbols for back references. Instead, we treat all the states in an NFA-like machine in the same manner, and add constraints to the acceptance condition of the constructed tagged-NFA. In addition, we have shown that our approach is immune to known Algorithmic Complexity Attacks.

VII. Conclusion

In this paper, we present a new matching algorithm for patterns with back references. Our approach works by transforming a back reference problem to a conditional submatch extraction problem, which in turn, is compiled to a tagged-NFA subject to some constraints. We build a toolchain and evaluate the performance of our approach using both synthetic data set and data set from real-world NIDS. Our experimental results show that our implementation NFA-backref is immune to known Algorithmic Complexity Attacks. In particular, NFA-backref is about three orders of magnitude faster than PCRE, a recursive backtracking-based pattern matching engine. Under benign traffic, NFA-backref is one order of magnitude slower than PCRE. We believe that our approach is better suited for network security tools because such tools are often exposed to hostile network traffic that can abuse a recursive backtracking based pattern matching engine. We believe that the performance of NFA-backref can be further improved with better code optimization.

References

[1] A. V. Aho. Algorithms for finding patterns in strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 255–300. 1990.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975.

[3] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 25:1–25:12, New York, NY, USA, 2008. ACM.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):62–72, 1977.

[5] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 249–257. IEEE Computer Society, 2004.

[6] R. Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...), 2007. http://swtch.com/~rsc/regexp/regexp1.html.

[7] R. Cox. Implementing regular expressions. http://swtch.com/~rsc/regexp/, Last retrieved in August 2011.

[8] S. Haber, W. Horne, P. Manadhata, M. Mowbray, and P. Rao. Efficient submatch extraction for practical regular expression. In *The 7th International Conference on Language and Automata Theory and Applications*, Bilbao, Spain, April 2013.

[9] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation, Third Edition*. Addison-Wesley, 2007.

[10] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *Annual Symp. on Field-Programmable Custom Computing Machines*, pages 111–120. IEEE Computer Society, 2002.

[11] D. E. Knuth, J. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *ACM SIGCOMM Conference*, pages 339–350. ACM, 2006.

[13] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE'00*, September 2000.

[14] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Symp. on Arch. for Networking and Comm. Systems*, pages 127–136. ACM, 2007.

[15] K. Namjoshi and G. Narlikar. Robust and fast pattern matching for intrusion detection. In *Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 740–748, Piscataway, NJ, USA, 2010. IEEE Press.

[16] PCRE. The Perl compatible regular expression library. http://www.pcre.org.

[17] R. Sidhu and V. Prasanna. Fast regular expression matching using FPGAs. In *Symp. on Field-Programmable Custom Computing Machines*, pages 227–238. IEEE Computer Society, 2001.

[18] R. Smith, C. Estan, and S. Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Annual Computer Security Applications Conf.*, pages 89–98. IEEE Computer Society, 2006.

[19] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Symp. on Security and Privacy*, pages 187–201. IEEE Computer Society, 2008.

[20] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM Conference*, pages 207–218. ACM, 2008.

[21] Snort. Download snort rules. http://www.snort.org/snort-rules/, Last retrieved in March 2013.

[22] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *Conf. on Computer and Comm. Security*, pages 262–271. ACM, 2003.

[23] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11:419–422, June 1968.

[24] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. TR 94-17, Department of Computer Science, University of Arizona, 1994.

[25] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Improving nfa-based signature matching using ordered binary decision diagrams. In *RAID'10*, volume 6307 of *Lecture Notes in Computer Science (LNCS)*, pages 58–78, Ottawa, Canada, September 2010. Springer.

[26] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Fast, memory-efficient regular expression matching with nfa-obdds. *Computer Networks*, 55(15):3376–3393, October 2011.

[27] L. Yang, P. Manadhata, W. Horne, P. Rao, and V. Ganapathy. Fast submatch extraction using obdds. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, ANCS '12, pages 163–174, New York, NY, USA, 2012. ACM.

[28] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ACM/IEEE Symp. on Arch. for Networking and Comm. Systems*, pages 93–102, 2006.