Fast Submatch Extraction using OBDDs[*]

Liu Yang
Rutgers University
lyangru@cs.rutgers.edu

Pratyusa Manadhata
HP Laboratories
manadhata@hp.com

William Horne
HP Laboratories
william.horne@hp.com

Prasad Rao
HP Laboratories
prasad.rao@hp.com

Vinod Ganapathy
Rutgers University
vinodg@cs.rutgers.edu

ABSTRACT

Network-based intrusion detection systems (NIDS) commonly use *pattern languages* to identify packets of interest. Similarly, security information and event management (SIEM) systems rely on pattern languages for real-time analysis of security alerts and event logs. Both NIDS and SIEM systems use pattern languages extended from regular expressions. One such extension, the *submatch* construct, allows the extraction of substrings from a string matching a pattern. Existing solutions for submatch extraction are based on non-deterministic finite automata (NFAs) or recursive backtracking. NFA-based algorithms are time-inefficient. Recursive backtracking algorithms perform poorly on pathological inputs generated by algorithmic complexity attacks. We propose a new approach for submatch extraction that uses *ordered binary decision diagrams* (OBDDs) to represent and operate pattern matching. Our evaluation using patterns from the Snort HTTP rule set and a commercial SIEM system shows that our approach achieves its ideal performance when patterns are combined. In the best case, our approach is faster than RE2 and PCRE by one to two orders of magnitude.

Categories and Subject Descriptors

C.2 [**Computer-Communication Networks**]: Network Operations

General Terms

Algorithms

Keywords

Regular expression, pattern matching, submatch, tagged-NFA, Ordered Binary Decision Diagram (OBDD)

[*]The first author completed parts of this work during an internship at HP Labs, Princeton, NJ.

1. INTRODUCTION

Regular expression-like pattern languages are widely used as building blocks of network security products. For example, network intrusion detection systems (NIDS) use thousands of patterns to describe malicious traffic. Security information and event management (SIEM) systems also use patterns to process event logs generated by hardware devices and software systems.

Pattern languages commonly used by NIDS are regular expressions extended with other features. One of the important features is the *capturing group*. A capturing group is a syntax used in modern regular expression implementations to specify a subexpression of a regular expression. Given a string that matches the regular expression, *submatch extraction* is the process of extracting the substrings corresponding to those subexpressions. In Snort 2012 rule set, more than 10% of `pcre` fields of the HTTP rules contain capturing groups. When a pattern containing a capturing group matches an input string, the submatch construct can identify parts of the input that are of interest to security administrators for analysis. For a regular expression like `username=(.*),hostname=(.*)` with an input string `username=Bob,hostname=Foo`, submatch construct can extract the two substrings `Bob` and `Foo` specified by the two capturing groups (the subexpressions wrapped by the two pairs of parentheses).

Likewise, SIEM systems, which perform real-time analysis of event logs and security alerts in enterprise networks, also make extensive use of submatch extraction. SIEM systems often collect data from a variety of hardware and software sensors, and must therefore normalize this data into a common format by extracting common fields from various data sources. SIEM systems use submatch extraction during data normalization and alert reporting. In a typical SIEM system, more than 90% of regular expressions used for data normalization contain capturing groups.

In both SIEM systems and NIDS, scalability of pattern matching and submatch extraction is key. NIDS are often deployed over high-speed network links, which require algorithms for pattern matching and submatch extraction be efficient enough to provide high throughput intrusion detection on large volume of network traffic. Similarly, a typical SIEM system collects logs from hundred of devices and applications, and must process terabytes of logs every day in enterprise networks.

There is plenty of prior work on making pattern matching for regular expressions time-efficient [1, 26, 6, 7, 15, 3, 13, 12, 18] and space-efficient [28, 2, 1, 23, 20, 21]. However, most of these works only considered regular expressions containing no capturing groups, i.e., they did not support submatch extraction. Existing solutions for submatch extraction are based on non-deterministic finite automata (NFAs) [14, 10] or recursive backtracking [16]. While NFAs are space-efficient and can extract submatches with a compact memory footprint, they are not time-efficient because they maintain a *frontier*, i.e., a set of states in which a NFA can be at any instant, that can contain $O(n)$ states where n is the NFA's number of states. This leads to an $O(n)$ operation time for the NFA for each input symbol. Google's RE2 package uses a combination of DFAs and NFAs to improve the time efficiency of submatch extraction [10]. RE2 constructs DFAs on demand (determination on the fly) and uses DFAs to locate a pattern's overall match location in an input string and then uses a NFA-based method to extract submatches. The time-efficiency of DFAs, however, often comes with a cost of state blow-up. RE2 can be very slow when the DFA construction fills up the limited state cache; it has to empty the state cache and restart the DFA construction process. Moreover, the actual submatch extraction of RE2 is performed using a NFA-based method, which is space-efficient, but not time-efficient. Tools such as Perl, PCRE, and Python use recursive backtracking for regular expression matching. The execution time of backtracking, however, can be exponential for certain types of regular expressions [9]; NIDS which employ backtracking suffer from algorithmic complexity attacks [19].

We present a novel approach to perform submatch extraction for regular expression-like pattern languages. Our approach is an extension of the NFA-OBDD work by Yang et al [26]. While both works employ the ordered binary decision diagram (OBDD) data structure, the NFA-OBDD approach in [26] did not consider the submatch construct, making it inapplicable to the 90% of regular expressions in a typical SIEM system. We extend the NFA-OBDD approach [26] in two ways: (1) we propose an approach to annotate capturing groups in regular expressions, and (2) present a new approach to perform submatch extraction. To demonstrate the feasibility of our approach, we evaluated our approach using patterns extracted from the Snort NIDS and a commercial SIEM product. Our experiments show that our approach achieves its ideal performance when patterns are combined. In the best case, our approach is faster than RE2 and PCRE by one to two orders of magnitude. In particular, we make the following contributions:

- We propose a new approach to tag capturing groups in a regular expression, and extend Thompson's NFA construction approach to convert a regular expression with capturing groups to a tagged-NFA.
- We present a novel and time-efficient technique (henceforth called *Submatch-OBDD*) to perform submatch extraction for regular expression-like pattern languages.
- We evaluated our approach's time efficiency and space efficiency by matching the patterns from the Snort system and a commercial SIEM system with network traces, synthetic traces, and enterprise event logs, and then compared our performance with two popular regular expression engines: RE2 and PCRE.

The remainder of the paper is organized as follows. We briefly describe ordered binary decision diagrams (OBDDs) as background knowledge in Section 2. After that, we present our design and implementation of Submatch-OBDD in Section 3, followed by our evaluation in Section 4. We discuss related work in Section 5 and conclude in Section 6.

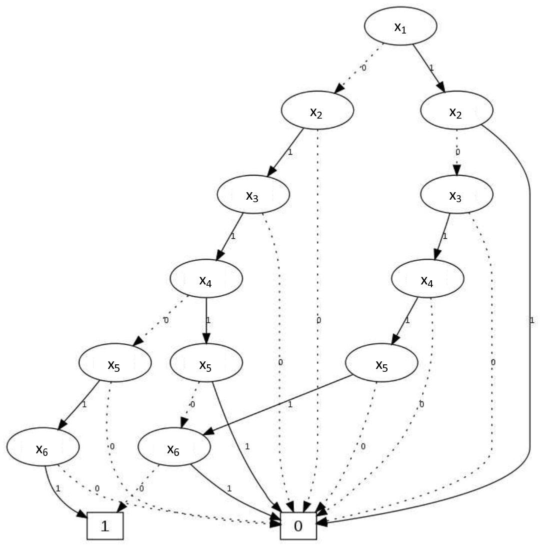

Figure 1: The ordered binary decision diagram of a Boolean function $f(x_1,x_2,x_3,x_4,x_5,x_6) = (\bar{x_1} \wedge x_2 \wedge x_3 \wedge \bar{x_4} \wedge x_5 \wedge x_6) \vee (\bar{x_1} \wedge x_2 \wedge x_3 \wedge x_4 \wedge \bar{x_5} \wedge \bar{x_6}) \vee (x_1 \wedge \bar{x_2} \wedge x_3 \wedge x_4 \wedge x_5 \wedge \bar{x_6})$ **with ordering** $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6$**.**

2. ORDERED BINARY DECISION DIAGRAMS

Bryant proposed ordered binary decision diagrams as a data structure for symbolically representing arbitrary Boolean functions [4]. OBDDs are widely used in digital logic design and testing, artificial intelligence, and model checking [5] to construct efficient algorithms for Boolean functions.

An OBDD represents a Boolean function, $f(x_1, x_2, \ldots, x_n)$, as a rooted directed acyclic graph (DAG), which has two types of nodes. A *non-terminal* node, v, is associated with an argument $\in \{x_1, x_2, \ldots, x_n\}$ and has two children, $low(v)$ and $high(v)$. A *terminal* node takes value $\in \{0, 1\}$. An OBDD is organized so that nodes along all paths from the root to terminal nodes follow a total order, $\prec$. An OBDD with root v denotes a function, f_v, which is recursively defined as:

- If v is a terminal node, then $f_v = value(v)$;
- If v is a non-terminal node and v is associated with argument x_i, then $f_v(x_1, x_2, \ldots, x_n) = \bar{x_i} \cdot f_{low(v)}(x_1, x_2, \ldots, x_n) \vee x_i \cdot f_{high(v)}(x_1, x_2, \ldots, x_n)$.

To evaluate a function with a set of argument values $x_1, x_2, \ldots, x_n$, start from the root, where if a node v is associated with x_i, then traverse to $low(v)$ if $x_i = 0$ and to $high(v)$ if $x_i = 1$, until a terminal node is reached. The function's value equals the value of the terminal node at the end of the traversal. Figure 1 shows an example Boolean function $f(x_1, x_2, x_3, x_4, x_5, x_6)$ and its OBDD representation in the order of $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6$.

OBDDs allow Boolean functions to be represented and manipulated efficiently. Testing satisfiability with OBDDs simply involves comparing a graph to that of a constant function **0**. Many operations of Boolean functions can be represented as two types of graph operations in OBDDs: APPLY and RESTRICT. APPLY allows Boolean operators such as $\vee$ and $\wedge$ to be applied to a pair of OBDDs. It takes a pair of OBDDs representing two functions f_1 and f_2, a binary operator OP, and produces a reduced graph representing function APPLY(OP, f_1, f_2) = f_1OPf_2. The resulting OBDD has the same variable ordering as the input OBDDs. RESTRICT is a unary operator. It transforms a function f into one representing the function $f|_{x_i=b}$ for a specified argument value $x_i = b$. The resulting OBDD does not have any node associated with x_i. The complexity of APPLY and RESTRICT is polynomial in the size of input OBDDs.

One important operation used in our Submatch-OBDD design is *existential quantification*. In particular, $\exists x_i \cdot f(x_1, x_2, \ldots, x_n) = f(x_1, x_2, \ldots, x_n)|_{x_i=0} \vee f(x_1, x_2, \ldots, x_n)|_{x_i=1}$. Expressed by OBDD, we have $OBDD(\exists x_i \cdot f(x_1, x_2, \ldots, x_n))$
$=$ APPLY($\vee$, RESTRICT($OBDD(f), 1 \leftarrow x_i$),
RESTRICT($OBDD(f), 0 \leftarrow x_i$)). As a result, $OBDD(\exists x_i \cdot f(x_1, x_2, \ldots, x_n))$ will have no node associated with x_i.

OBDDs are extremely useful in obtaining concise representations of relations over finite domains. If R is a n-arity relation over $\{0, 1\}$, then R can be represented by an OBDD using its characteristic function $f_R(x_1, x_2, \ldots, x_n) = 1$ iff $R(x_1, x_2, \ldots, x_n)$. For example, a 3-ary relation $R = \{(1, 0, 1), (1, 1, 0)\}$ can be expressed by $f_R(x_1, x_2, x_3) = (x_1 \wedge \bar{x_2} \wedge x_3) \vee (x_1 \wedge x_2 \wedge \bar{x_3})$, which is a Boolean function and can therefore be represented using an OBDD.

A set of elements can also be expressed as an OBDD. If S is a set over a domain D, then we can define a relation $R_S(s) = 1$ if $s \in S$. Operations on sets can be expressed as Boolean operations and manipulated by OBDDs. For example, ISEMPTY($S \cap T$) is equivalent to checking the satisfiability of $OBDD$(APPLY($\wedge, S, T$)). Our Submatch-OBDD design in Section 3 converts relations and sets to OBDDs to achieve time efficient operations.

3. DESIGN AND IMPLEMENTATION

We first give an overview of our approach before describing the technical details.

3.1 Solution Overview

A key observation underlying our approach is that adding a capturing group to a regular expression does not change the language defined by the regular expression. It is known that every language defined by a regular expression is also defined by a finite automaton [11]. However, traditional automata do not support capturing groups. We present an approach to annotate capturing groups in regular expressions and extend Thompson's approach to convert a regular expression with capturing groups to a NFA-like machine where transitions within capturing groups are tagged. We then present a novel approach to do submatch extraction using the tagged-NFAs. To improve the time efficiency of submatch extraction, we represent tagged-NFAs with symbolic Boolean functions, and manipulate the Boolean functions using ordered binary decision diagrams (OBDDs).

3.2 Tagging NFAs for Submatch

The syntax of regular expressions with capturing groups on an alphabet Σ is

$$E ::= \epsilon \cup a \cup EE \cup E|E \cup E* \cup (E) \cup [E]$$

where a stands for an element of Σ, and ϵ denotes for zero occurrence of a symbol. We use square brackets [,] to group terms in a regular expression that are not capturing groups, because the usual parentheses (,) are reserved for marking capturing groups. If X and Y are sets of strings we use XY to denote $\{xy : x \in X, y \in Y\}$, and $X|Y$ to denote $X \cup Y$. We use $E*$ to denote the closure of E under concatenation.

We use tags to distinguish the capturing groups within a regular expression. Given a regular expression containing c capturing groups, we assign tags $t_1, t_2, ..., t_c$ to each capturing group in the order of their left parentheses as E is read from left to right. We denote the set of tags by $T = \{t_1, t_2, ..., t_c\}$. We use $tag(E)$ to refer to the resulting tagged regular expression. For example, if $E = ((a*)|b)(ab|b)$ then $tag(E) = ((a*)_{t_2}|b)_{t_1}(ab|b)_{t_3}$.

The language $L(F)$ for a tagged regular expression $F = tag(E)$ is a set of tagged strings, defined by $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$, $L(F_1F_2) = L(F_1) \cdot L(F_2)$, $L(F_1|F_2) = L(F_1) \cup L(F_2)$, $L(F*) = L(F)*$, $L([F]) = L(F)$, and $L((F)_t) = \{\alpha_t : \alpha \in L(F)\}$, where $()_t$ denotes a capturing group with tag t and α_t denotes the string α tagged with t. A string α is tagged by t, if and only if each character in α is tagged by t. Substrings of α may be tagged by other tags. Since capturing groups can be nested, a character can be tagged by multiple tags. An example of tagged string for a tagged regular expression is: $ab_{t_1}b_{t_1}b_{t_1} \in L(a(b*)_{t_1})$.

Definition A *valid assignment of submatches* for a string α that matches regular expression E is a map $sub : \{t_1, t_2, \ldots t_c\} \rightarrow \Sigma^*$ such that there exists $\beta \in L(tag(E))$ satisfying the following:

(i) $\beta|_\Sigma = \alpha$, where $\beta|_\Sigma$ represents the projection of characters in β onto their corresponding values of Σ;

(ii) if t_i occurs in β then $sub(t_i)$ is the last consecutive sequence of characters that are assigned with tag t_i;

(iii) if t_i does not occur in β, then $sub(t_i) =$ NULL;

For example, consider the regular expression $[(a|c)(b|d)]*$ with input string *abcd*. A valid submatch assignment satisfying the above conditions is $sub(t_1) = c$, $sub(t_2) = d$.

It is well known that a regular expression can be converted to an ϵ-NFA which defines the same language using Thompson's approach [11]. An ϵ-NFA can be reduced to an ϵ-free NFA through an ϵ-closure mechanism [11]. In this paper, we extend Thompson's algorithm in a way such that it can convert a regular expression containing capturing groups to a tagged ϵ-NFA defining the same language. A tagged ϵ-NFA can be described by a 7-tuple $A = (Q, \Sigma, T, \delta, \gamma, S, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, T is a finite set of tags that each represents a capturing group, S is a set of start states, F is a set of accept states, δ is the transition function, and γ is a tag output function $\gamma : Q \times \Sigma \times Q \rightarrow 2^T$, which associates each transition with a tag set (which can be empty).

A tagged NFA can be constructed as follows, starting from the three base cases shown in Figure 2. Figure 2(a) is the NFA of expression ϵ, Figure 2(b) handles the empty regular expression, and Figure 2(c) gives the NFA of a single symbol a with a set of tags $\tau \in 2^T$ corresponding to capturing groups associated with the illustrated transition. More complex tagged NFAs can be constructed using the union, concatenation, and closure constructs, by combining smaller tagged NFAs as shown in Figure 3. A tagged NFA constructed using the above approach contains ϵ transitions. Such a tagged NFA can be converted to an ϵ-free NFA in

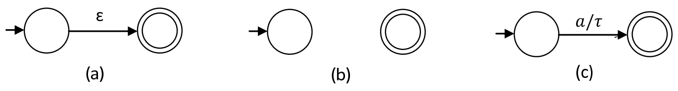

Figure 2: Constructing tagged NFAs for (a) NFA of ϵ; (b) NFA of an empty regular expression; (c) NFA of a symbol a wrapped by capturing groups denoted by τ.

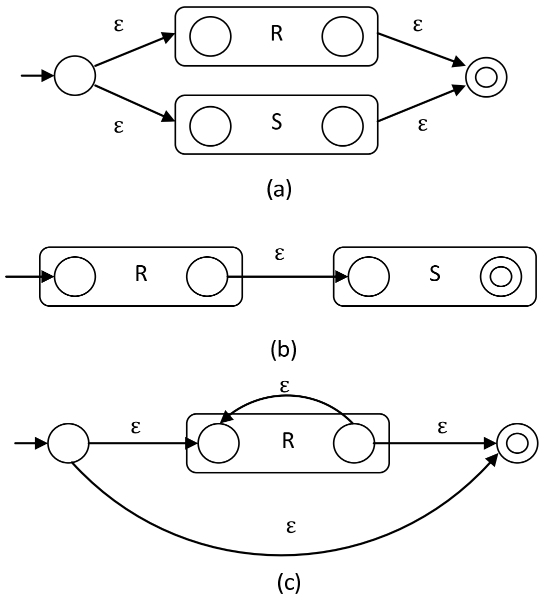

Figure 3: The (a) union $R|S$, (b) concatenation RS, and (c) closure constructs $R*$ of tagged NFA construction from a regular expression.

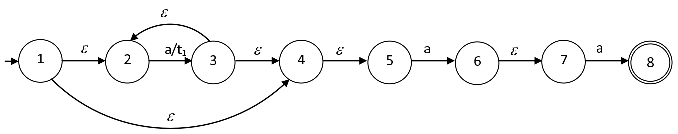

Figure 4: The tagged ϵ-NFA of $(a*)aa$, where the transition associated with the leftmost character a is tagged by t_1 because $a*$ is within a capturing group.

a manner akin to the standard ϵ-closure algorithm for standard NFAs. We denote the corresponding ϵ-free tagged NFA as $A_1 = (Q_1, \Sigma, T, \delta_1, \gamma_1, S_1, F_1)$, where the components of A_1 are defined in a manner akin to A (the tagged ϵ-NFA).

Example Consider an example regular expression $(a*)aa$. Figure 4 shows an the tagged ϵ-NFA, where the capturing group is tagged by t_1. Figure 5 shows the corresponding ϵ-free tagged-NFA.

3.3 Operations on Tagged NFAs

The transition function δ_1 and tag output function γ_1 of a tagged ϵ-free NFA can be represented by a four-column table denoted by $\Delta(x, i, y, \tau)$, which is a set of quadruples (x, i, y, τ) such that there is a transition labeled by input symbol i from state x to state y with a set of output tags τ. Table 1 shows the tagged transition table of the example NFA in Figure 5, where each tagged transition is represented

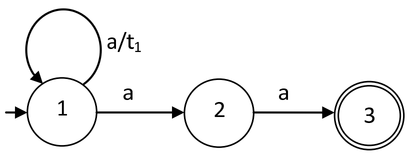

Figure 5: The tagged ϵ-free NFA of $(a*)aa$ after ϵ-elimination, where state numbers 1, 2, and 3 are obtained by renaming and merging states 2, 5, 7, and 8 in the ϵ-NFA during ϵ-closure calculation.

x	i	y	τ
1	a	1	$\{t_1\}$
1	a	2	ϕ
2	a	3	ϕ

Table 1: Transition table of the tagged NFA in Figure 5.

by a row in the table. $\Delta(x, i, y, \tau)$ allows us to perform two key operations on tagged NFAs — *match test* and *submatch extraction*, where the match test checks whether an input string is accepted by a tagged NFA; if so, the submatch extraction procedure returns a valid assignment of submatches of the input string.

3.3.1 Match Test

Testing whether an input string matches a regular expression with capturing groups can be done by operating its tagged NFA. The process is similar to operating a traditional NFA, except that we need to do bookkeeping to be used for submatch extraction. The match test of a tagged NFA for a given input string $a_1a_2 \ldots a_l \in \Sigma^*$ is performed by consuming one input symbol at a time, and modifying the frontier of active states appropriately using the transition function δ_1. As we modify the frontier, we also record the transitions that the tagged-NFA makes by recording quadruples that store the states traversed by each transition, as well as the tags corresponding to those transitions. We denote these sets of transitions using Δ_1, Δ_2, and Δ_l, where each Δ_i is a set of quadruples of the form (x, i, y, τ) corresponding to a source state, an input symbol, a target state, and the corresponding tag.

After the last input symbol a_l is consumed, we check whether any state in the frontier set belongs to accept states F_1. If so, the input string $a_1a_2 \ldots a_l$ is accepted by the tagged NFA A_1, i.e., the input string matches the regular expression defined by A_1.

Example Consider the example regular expression $(a*)aa$ in Figure 5, where its tagged transitions $\Delta(x, i, y, \tau)$ are shown in Table 1. For convenience, we denote the three quadruples in Table 1 by row_1, row_2, and row_3. Let's use $aaaa$ as an input string. For the i^{th} input symbol, we use X_i to denote the current frontier set and Y_i to denote the next frontier set after the symbol is consumed. Start from the first input symbol a and start states $S_1 = \{1\}$, we have $\Delta_1 = \{row_1, row_2\}$, $Y_1 = \{1, 2\}$. Rename Y_1 to X_2 and follow the process described in the frontier derivation, we can obtain $\Delta_2 = \{row_1, row_2, row_3\}$, $X_3 = \{1, 2, 3\}$, $\Delta_3 = \{row_1, row_2, row_3\}$, $X_4 = \{1, 2, 3\}$, and $\Delta_4 = \{row_1, row_2, row_3\}$, $X_5 = \{1, 2, 3\}$. Figure 6 visualizes how the frontier set evolves after consuming each input symbol during the match test. An arrow between two

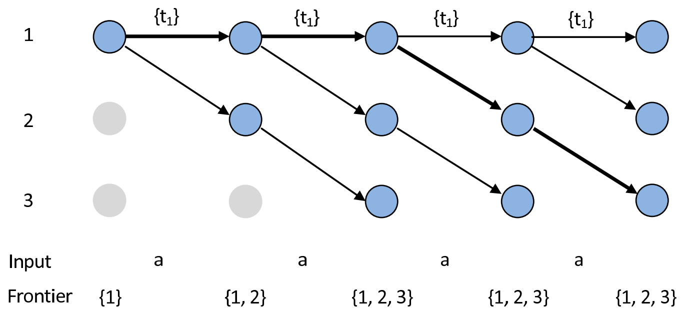

Figure 6: Example of frontier derivation for the tagged NFA in Figure 5 with input string $aaaa$. The dark circles of each column stands for the frontier states after consuming an input symbol. A light gray circle means that a state is not in a frontier state set. An arrow between two circles represent a transition. An arrow is labeled by a submatch tag if the denoted transition is within a capturing group.

nodes denotes a transition. If an arrow is tagged, it means that a transition is associated with one or more submatch tags, e.g., the $\{t_1\}$ above the arrow between states 1 and 1 indicate this transition is within a capturing group.

3.3.2 Submatch Extraction

If an input string is accepted by a regular expression which has capturing groups, the submatches of the input string need to be extracted. Recall that the NFA match test process described above actually considers all possible branches (transitions) when consuming each input symbol. If the input string is accepted by the tagged NFA, then there exists at least one path from a start state to an accept state, where edges of the path denote transitions between states and are sequentially associated with the individual symbols of the input string. An edge may be associated with one or more submatch tags, or no tag at all. For example, the bold arrows in Figure 6 shows a path from start state 1 to the accept state 3. Such a path allows us to perform submatch extraction.

In fact, *any path from a start state to an accept state during a match test on a tagged NFA generates a valid assignment of submatches*. A review of the match test process can help us to understand why: Since a path from a start state to an accept state is a number of sequential and valid operations of a tagged NFA on an input string, the assignment of submatch tags on each input symbol is also valid. The collections of the last consecutive sequences of symbols associated with the same tags that satisfy the conditions of the definition in Section 3.2 generate a valid assignment of submatches.

Path Finding Assume an input string $a_1a_2\dots a_l$ is accepted by a tagged NFA A_1, and q_f is an accept state after consuming the last symbol a_l. We present a backward traversal approach to find a path which allows for submatch extraction. Starting from one of the accept states q_f, with the last input symbol a_l, perform a lookup on Δ_l for quadruples (x, i, y, τ) such that $y = q_f$ and $i = a_l$. Pick any quadruple (q_l, a_l, q_f, τ) which satisfies this condition, then q_l is a previous state which leads the automaton to q_f with the last input symbol a_l, and τ is the corresponding submatch tags associated with a_l. We note that τ can be empty. Using q_l, with input symbol a_{l-1}, perform a lookup on Δ_{l-1} for quadruples (x, i, y, τ) such that $y = q_l$ and $i = a_{l-1}$. Such quadruples will allow us to find a previous state of q_l with input symbol a_{l-1}, along with submatch tags associated with a_{l-1} if there are any. Continue this process for $a_{l-2}, \dots,$ and a_1. Finally, we will reach a start state q_1. Then $q_1, q_2, \dots, q_l, q_f$ is a valid traversal path for input string $a_1a_2\dots a_l$. During the backward path finding, each symbol in $a_1a_2\dots a_l$ is assigned with zero or a set of submatch tags. Submatches of an accepted input string can be extracted by scanning the input strings and collecting the last consecutive sequence of symbols associated with the same submatch tags. Given a regular expression and a matching string, there might exist multiple paths from a start state to an accept state. Thus, there might exist multiple ways to assign valid submatches.

Example Figure 6 shows a traversal path of input string $aaaa$ on the tagged NFA shown in Figure 5. The path is marked by bold arrows. Along this path, we can see that the first two symbols of $aaaa$ are associated with tag t_1, and the last two symbols have no submatch tag. Thus, the submatch of $aaaa$ for regular expression $(a*)aa$ is the substring of the first two symbols, i.e., aa.

The match test and submatch extraction algorithms described in Section 3.3.1 and 3.3.2 are space efficient since the construction is based on NFA. However, they are not time efficient. During the match test, the number of states in a frontier is $O(|Q_1|)$ (the size of NFA). To derive the next frontier, states in the current frontier need to be processed one by one. Thus, the number of lookups at the transition table during the match test for an input string of length l is $O(|Q_1| \times l)$. Similarly, the number of table lookups performed during submatch extraction can be estimated as $O(|Q_1| \times l)$. If we can find an approach which allows us to derive frontiers (in match test) and previous states (in submatch extraction) more efficiently, then the time efficiency of the algorithms can be improved. Fortunately, we already have the data structures to do that. Our approach is to represent tagged NFAs, perform match testing and submatch extraction using Boolean functions (Section 3.4), and manipulate the Boolean functions using ordered binary decision diagrams (OBDDs) (Section 3.5).

3.4 Boolean Function Representation

For convenience, we discuss tagged NFAs in which ϵ transitions have been eliminated. The Boolean function of a tagged NFA $A_1 = (Q_1, \Sigma, T, \delta_1, \gamma_1, S_1, F_1)$ uses four vectors of Boolean variables, $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{i}$, and $\boldsymbol{t}$. Vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are used to denote states in Q_1, and they contain $\lceil \lg |Q_1| \rceil$ Boolean variables each. Vector $\boldsymbol{i}$ is used to denote symbols in Σ and it contains $\lceil \lg |\Sigma| \rceil$ Boolean variables. Vector $\boldsymbol{t}$ is used to denote submatch tags and it contains $\lceil |T| \rceil$ Boolean variables. We construct the following Boolean functions for the tagged NFA A_1.

• $\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ denotes the tagged transition table of A_1. It is a disjunction of all tagged transition relations (x, i, y, t). As an example, the Boolean encoding of transition relations in Table 1 is shown in Table 2, where states are encoded by two bits, input symbol is encoded by one bit (since there is only one symbol a), and submatch tags are encoded by one bit. Specifically, states 1, 2, and 3 are encoded as 01, 10, and 11; symbol a is encoded as 1; and submatch tag t_1 is encoded as 1. The fifth column of Table 2 lists the function values for each set of Boolean encodings. The function value of

$\boldsymbol{x}$	$\boldsymbol{i}$	$\boldsymbol{y}$	$\boldsymbol{t}$	$\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$
0 1	1	0 1	1	1
0 1	1	1 0	0	1
1 0	1	1 1	0	1

Table 2: Boolean encoding of transitions in Table 1.

Boolean encodings for tagged transitions is 1. The Boolean encoding in Table 2 can be symbolically translated to

$$\begin{aligned}\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) &= (\bar{x_1} \wedge x_2 \wedge i \wedge \bar{y_1} \wedge y_2 \wedge t_1) \\ &\vee (\bar{x_1} \wedge x_2 \wedge i \wedge y_1 \wedge \bar{y_2} \wedge \bar{t_1}) \\ &\vee (x_1 \wedge \bar{x_2} \wedge i \wedge y_1 \wedge y_2 \wedge \bar{t_1})\end{aligned}$$

$\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ is equivalent to the function f in Figure 1 if we rename variables i, y_1, y_2, and t_1 to x_3, x_4, x_5, and x_6 respectively.

- $\mathcal{I}_\sigma(\boldsymbol{i})$ stands for the Boolean representation of symbols in Σ. As an example, symbol a in Table 1 can be symbolically represented by $\mathcal{I}_a(\boldsymbol{i}) = i$.
- $\mathcal{F}(\boldsymbol{x})$ is a Boolean function representing frontier states. In the tagged NFA shown in Figure 5, consider state $\{1\}$ with input symbol a, the new frontier has two states $\{1, 2\}$, which can be symbolically represented by $\mathcal{F}(\boldsymbol{x}) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge \bar{x}_2)$.
- $\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ is used to represent the intermediate transitions for frontier $\mathcal{F}(\boldsymbol{x})$ during a match test process.
- $\mathcal{A}(\boldsymbol{x})$ is used to define the Boolean representation of accept states of a tagged NFA. For the tagged NFA shown in Figure 5, the accept states is $\{3\}$, thus, $\mathcal{A}(\boldsymbol{x}) = x_1 \wedge x_2$.

The Boolean functions described above can be automatically computed for any tagged NFA. We next describe how to perform the match test and submatch extraction described in Section 3.3 using these Boolean functions.

3.4.1 Match Test

The match test process is similar to that described in [26], except that we do book-keeping here to be used for submatch extraction. Suppose the frontier of a tagged NFA is $\mathcal{F}(\boldsymbol{x})$ at some instant of frontier derivation, and the next input symbol is σ, then the next frontier states can be computed using the following Boolean operations:

$$\mathcal{G}(\boldsymbol{y}) = \exists\, \boldsymbol{x} \cdot\ \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{t} \cdot [\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})] \qquad (1)$$

where

$$\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) = \mathcal{F}(\boldsymbol{x}) \wedge \mathcal{I}_\sigma(\boldsymbol{i}) \wedge \Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) \qquad (2)$$

We now explain why Equation (1) produces the new frontier states. Recall that $\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ is the disjunction of the tagged transitions of a NFA. The conjunctions of $\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ with $\mathcal{F}(\boldsymbol{x})$ and $\mathcal{I}_\sigma(\boldsymbol{i})$ on the right side of Equation (2) actually *selects* rows in the truth table of $\Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ that correspond to outgoing transitions from the states in the current frontier $\mathcal{F}(\boldsymbol{x})$ labeled with symbol σ. These transitions are denoted by $\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$, which is a function of $\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}$, and $\boldsymbol{t}$. The new frontier states are the target states of the selected transitions and are only associated with $\boldsymbol{y}$. To extract the new frontier states, we existentially quantify $\boldsymbol{x}, \boldsymbol{i}$, and $\boldsymbol{t}$ using the existential quantification operator introduced in Section 2. We rename $\boldsymbol{y}$ to $\boldsymbol{x}$ to express the new frontier states in terms of $\boldsymbol{x}$.

Consider the tagged NFA in Figure 5. Suppose the current frontier is $\{1\}$ and the next input symbol is a. Then

$$\begin{aligned}\mathcal{F}(\boldsymbol{x}) \wedge \mathcal{I}_a(\boldsymbol{i}) \wedge \Delta(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) &= (\bar{x_1} \wedge x_2 \wedge i \wedge \bar{y_1} \wedge y_2 \wedge t_1) \\ &\vee (\bar{x_1} \wedge x_2 \wedge i \wedge y_1 \wedge \bar{y_2} \wedge \bar{t_1})\end{aligned}$$

Apply existential quantification of $\boldsymbol{x}, \boldsymbol{i}$, and $\boldsymbol{t}$ on the above conjunctions we obtain $(\bar{y_1} \wedge y_2) \vee (\wedge y_1 \wedge \bar{y_2})$, which is the symbolic Boolean representation of the new frontier states $\{1, 2\}$.

To check whether the automaton is in an accept state, simply check the satisfiability of the conjunction between $\mathcal{F}(\boldsymbol{x})$ and $\mathcal{A}(\boldsymbol{x})$. Rename the above example frontier $(\bar{y_1} \wedge y_2) \vee (\wedge y_1 \wedge \bar{y_2})$ to $(\bar{x_1} \wedge x_2) \vee (\wedge x_1 \wedge \bar{x_2})$ and do a conjunction with $\mathcal{A}(\boldsymbol{x}) = x_1 \wedge x_2$. The result is not satisfiable, thus, the automaton is not in an accept state.

3.4.2 Submatch Extraction.

Now we discuss how to extract submatches using Boolean function operations. The process starts from the last symbol and one of the states where the input string is accepted. For convenience, we call the current state of a backward path finding a reverse frontier, which contains only one state because we are only interested in finding one path. Suppose at an instant of the path finding the reverse frontier representation is $\mathcal{F}_r(\boldsymbol{y})$, and the previous input symbol is σ. A previous state which leads the automaton to $\mathcal{F}_r(\boldsymbol{y})$ can be derived from the following Boolean function:

$$\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) = \mathcal{F}_r(\boldsymbol{y}) \wedge \mathcal{I}_\sigma(\boldsymbol{i}) \wedge \Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) \qquad (3)$$

where $\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ denotes the intermediate tagged transitions corresponding to symbol σ during the match test process. The conjunctions on the right side of Equation (3) *selects* tagged transitions (labeled by σ) from $\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ where the target state is $\mathcal{F}_r(\boldsymbol{y})$. The previous states are associated with $\boldsymbol{x}$ in $\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$. Since we are only interested in one path, we simply pick one row in the truth table of $\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$ to find one previous state of $\mathcal{F}_r(\boldsymbol{y})$. If we denote the picked row as PickOne($\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$), a previous state $\mathcal{G}(\boldsymbol{x})$ of $\mathcal{F}_r(\boldsymbol{y})$ can be derived by

$$\mathcal{G}(\boldsymbol{x}) = \exists\, \boldsymbol{y} \cdot \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{t} \cdot \mathcal{H}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) \qquad (4)$$

$$\mathcal{H}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) = \textsc{PickOne}(\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})) \qquad (5)$$

To obtain submatch tags $\tau(\boldsymbol{t})$ associated with σ, we existentially quantify $\boldsymbol{x}$, $\boldsymbol{i}$, and $\boldsymbol{y}$ on $\mathcal{H}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t})$.

$$\tau(\boldsymbol{t}) = \exists\, \boldsymbol{x} \cdot \exists\, \boldsymbol{i} \cdot \exists\, \boldsymbol{y} \cdot \mathcal{H}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) \qquad (6)$$

Consider the example in Figure 6. After consuming the fourth input symbol of $aaaa$, the automaton accepts and

$$\begin{aligned}\Delta_\mathcal{F}(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) &= (\bar{x_1} \wedge x_2 \wedge i \wedge \bar{y_1} \wedge y_2 \wedge t_1) \\ &\vee (\bar{x_1} \wedge x_2 \wedge i \wedge y_1 \wedge \bar{y_2} \wedge \bar{t_1}) \\ &\vee (x_1 \wedge \bar{x_2} \wedge i \wedge y_1 \wedge y_2 \wedge \bar{t_1})\end{aligned}$$

Starting from the accept state 3 ($\mathcal{F}_r(\boldsymbol{y}) = y_1 \wedge y_2$) and the last symbol a ($\mathcal{I}_a(\boldsymbol{i}) = i$), do a conjunction according to Equation (3) we get $\Delta_r(\boldsymbol{x},\boldsymbol{i},\boldsymbol{y},\boldsymbol{t}) = (x_1 \wedge \bar{x_2} \wedge i \wedge y_1 \wedge y_2 \wedge \bar{t_1})$, which has only one tagged transition. Perform existential quantifications according to Equation (4) and (5) we obtain the Boolean representation of a previous state as $x_1 \wedge \bar{x_2}$, which translates to state 2. Do an existential quantifications according to Equation (6) we get $\tau(\boldsymbol{t}) = \bar{t_1}$, which means that no tag is associated with the fourth symbol a. Applying the same approach on the 3rd, 2nd, and 1st symbols we obtain a path from state 1 to 3, where the 1st and 2nd symbol a are assigned with submatch tag t_1. Thus, the submatch of $aaaa$ to $(a*)aa$ is $sub(t_1) = aa$.

A submatch assignment obtained by our approach is not necessarily the left most, longest submatch, which is required by POSIX. However, POSIX doesn't have a notion of "greedy" and "reluctant" closures, which give some control over the length of the submatch. Thus, POSIX is incomplete. Standard libraries like Java and PCRE have behaviors that are not POSIX compliant.

3.5 Submatch-OBDD

To improve the efficiency of the match test and submatch extraction, we represent and manipulate the Boolean functions defined in Section 3.4 using OBDDs. We call our model Submatch-OBDD. A Submatch-OBDD for a tagged NFA $A_1 = (Q_1, \Sigma, T, \delta_1, \gamma_1, S_1, F_1)$ is a 5-tuple $[OBDD(\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})), \{OBDD(\mathcal{I}_\sigma|\forall\sigma \in \Sigma)\}, \{OBDD(\mathcal{T}_t|\forall t \in T)\}, OBDD(\mathcal{F}_{S_1}), OBDD(\mathcal{A})]$, where $\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ is Boolean representation of tagged transitions, $\mathcal{I}_\sigma$ is the Boolean representation of a symbol $\sigma \in \Sigma$, $\mathcal{T}_t$ is Boolean representation of a tag $t \in T$, $\mathcal{F}_{S_1}$ is Boolean representation of start states, and $\mathcal{A}$ is the Boolean representation of accept states F_1.

To understand why OBDDs can improve the time-efficiency of tagged NFA operations, consider frontier derivation on a tagged NFA. To derive a new set of frontier states, the tagged transition table must be retrieved for each state in the current frontier $\mathcal{F}$, leading to $O(|\mathcal{F}|)$ operations for each input symbol. On the other hand, the time-complexity of using OBDDs to derive the next frontier is determined by the two conjunctions and one existential quantification in Equation (1) and (2). When the frontier set $\mathcal{F}$ is large, the cost of doing the two conjunctions and one existential quantification is often smaller than doing $|\mathcal{F}|$ lookups on the transition table. Using the same method, we can calculate that the time complexity of submatch extraction is the same as the match test process. For a tagged-NFA with n states, the size of frontier set $|\mathcal{F}|$ is $O(n)$. Thus, the cost to process an input string of l bytes by our approach is between $O(l)$ and $O(nl)$. In other words, the time complexity of Submatch-OBDD is between a pure DFA and a pure NFA approach.

The space efficiency of Submatch-OBDD is comparable to tagged NFAs. The space cost of a Submatch-OBDD is dominated by $OBDD(\Delta(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t}))$, which needs a total of $2 \times \lceil \lg |Q_1| \rceil + \lceil \lg |\Sigma| \rceil + \lceil |T| \rceil$ Boolean variables. In the worst case, the size of the OBDD is $O(|Q_1|^2 \times |\Sigma| \times 2^{|T|})$, which is comparable to the size of transitions of a tagged NFA. We note that the OBDDs of intermediate transitions $\Delta_{\mathcal{F}}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}, \boldsymbol{t})$ for all input symbols also take some space, mainly depending on the size of input string. We will show that such a cost is not a concern in practice in Section 4.

3.6 Implementation

We implemented Submatch-OBDD as a toolchain in `C++`. The toolchain has two offline components, Re2Tnfa and Tnfa2Obdd, and one online component, PatternMatch. Re2Tnfa accepts patterns as input and outputs tagged-NFAs that defines the same languages as the input patterns. Tnfa2Obdd then generates the tagged-NFAs' OBDD representations. PatternMatch then performs match test and submatch extraction on an input stream using the OBDD representations. Our implementation interfaces with the popular `CUDD` library [22] for OBDD construction and manipulation.

In comparison, both PCRE and RE2 are implemented in `C++`. PCRE uses a recursive backtracking approach: It compiles a pattern into a tree like structure and then uses recursive backtracking to match patterns and extract submatches. RE2 uses a combination of DFAs and NFAs for submatch extraction: Given a pattern and an input string, RE2 constructs and uses backward and forward DFAs to locate the pattern's overall match in the input string. It then uses NFA based approaches to find submatches in the overall match. For memory efficiency, RE2 doesn't construct entire DFAs. It creates DFA states on demand (determination on-the-fly) and stores them in a limited sized cache; when the cache gets full, RE2 empties the cache and restarts the DFA construction process.

4. EVALUATION

We evaluated the performance of our Submatch-OBDD implementation using patterns used in real systems. We measured Submatch-OBDD's time efficiency and space efficiency by matching the patterns with network traces, synthetic traces, and enterprise event logs, and then compared our performance with two popular regular expression engines: RE2 and PCRE. Our findings suggest that Submatch-OBDD achieves its ideal performance when patterns are combined. In the best case, Submatch-OBDD is faster than RE2 and PCRE by one to two orders of magnitude. All the performance numbers of Submatch-OBDD reported in this section were obtained based on the variable ordering of $\boldsymbol{i} \prec \boldsymbol{x} \prec \boldsymbol{y} \prec \boldsymbol{t}$.

4.1 Data Sets

We used three sets of patterns and trace files in our evaluation.

Snort-2009.

We extracted 115 patterns from a Snort 2009 HTTP rule set of 3078 patterns. All patterns were extracted from the `pcre` fields of the rules. Since our focus is submatch extraction, we excluded patterns containing no capturing groups and patterns containing back references as patterns with back references cannot be represented by regular languages. Each extracted pattern contains one to six capturing groups.

We used two network traces and one synthetic trace to evaluate the performance of our approach on the Snort-2009 pattern set.

- The first web trace was a 1.2GB network traffic collected using `tcpdump` from our department's web server. The average packet size of this trace is 126 bytes with a standard deviation of 271. The second web trace was a 1.3GB network traffic collected by crawling URLs that appeared on Twitter using a `python` script and recording the full length packets using `tcpdump`. The average packet size of the second trace is 1202 bytes with a standard deviation of 472.

- We also created a synthetic trace to observe how different implementations perform under the backtracking algorithmic complexity attack [19]. By reviewing the 115 patterns of the Snort-2009 pattern set, we found that several of them are vulnerable to the backtracking algorithmic complexity attack if a regular expression engine is implemented by backtracking, e.g., PCRE. We then crafted a 1MB trace which can exploit the backtracking behavior of a backtracking-based pattern matching engine. The average line length of the trace is 311 bytes with a standard deviation of 5.

Snort-2012.

We also evaluated our approach with the latest rules from the Snort system. We extracted 403 patterns (regular expressions with capturing groups) from a snapshot of the Snort-2012 HTTP rule set containing 3990 rules. All patterns were extracted from the `pcre` fields of the rules. Like the patterns of Snort-2009, we excluded patterns containing back references as they can not be represented by regular languages. Patterns containing no capturing group are also excluded as our focus was on submatch extraction. Each extracted pattern has one to ten capturing groups.

We used two web traces and one synthetic trace to evaluate the performance of different approaches on this pattern set. The two web traces are the same as those used in the Snort-2009 pattern set evaluation. The synthetic trace was created after reviewing the 403 patterns: We found that several of the 403 patterns are vulnerable to backtracking algorithmic attacks. We then crafted a 1MB trace which can exploit the backtracking behavior of a backtracking-based pattern matching engine and evaluated its effects on Submatch-OBDD, RE2, and PCRE. The average line length of this synthetic trace is 689 bytes with a standard deviation of 41.

Firewall-504.

We also obtained a set of 504 patterns used by a commercial SIEM system C to normalize logs generated by a commercial firewall, F. For commercial reasons, we do not disclose the names of the SIEM system and the firewall. Each pattern in the set has 1-22 capturing groups. We collected 87 MBs of firewalls logs generated by F in an enterprise setting and measured our performance on the logs. The logs consist of 1.01 million lines of text and the average line size is 87 bytes with standard deviation of 51. We did not create synthetic trace for this pattern set as firewall logs cannot easily be controlled by an attacker.

4.2 Experimental Setup

We conducted our experiments on an Intel Core2 Duo E7500 Linux-2.6.3 machine running at 2.93 GHz with 2 GB of RAM. We measure the time efficiency of different approaches in the average number of CPU cycles needed to process one byte of a trace file. We only measure pattern matching and submatch extraction time, and exclude pattern compilation time. Similarly, we measure memory efficiency in megabytes (MB) of RAM used during pattern matching and submatch extraction.

We measure the performance of each approach on a pattern set in two *configurations*. In one configuration, Conf.S, we match each pattern with the input stream sequentially. For example, we match each pattern in the Snort-2009 set with each packet in the network traces. Combining all patterns of a pattern set into one single pattern, however, allows us to match each packet with all patterns in one pass. This configuration, Conf.C, is also useful in the log normalization process of a SIEM system. The system can match an event log with all rules in one pass and extract all fields of interest instead of matching the logs with each rule sequentially.

Given a pattern set with n patterns and an input trace of M bytes, we measured performance of an approach in the following two configurations.

- **Conf.S (Sequential):** We compile each pattern individually and then match the compiled patterns with the trace sequentially. If the i^{th} pattern's execution time for the M bytes trace is t_i cycles, then the time efficiency of an approach to the pattern set is $\frac{t_1+\cdots+t_n}{M}$ cycles/byte.

- **Conf.C (Combination):** We combine the n patterns together into one pattern using the UNION operation. We compile the combined pattern and match it with the input trace. If the combined pattern's execution time for the M bytes trace is t cycles, then an approach's time efficiency to the pattern set is $\frac{t}{M}$ cycles/byte. When an input string matches a specific pattern in the combined pattern, Submatch-OBDD emits the submatches, as well as the pattern that matches the input string.

4.3 Performance Results

4.3.1 *Snort-2009*

Table 3 shows the execution times (cycles/byte) and memory consumption of RE2, PCRE, and Submatch-OBDD for the Snort-2009 pattern set on the web traces and synthetic trace. We have the following observations:

- Submatch-OBDD achieves its ideal performance in Conf.C, i.e., when patterns are combined together for pattern matching and submatch extraction.

- Submatch-OBDD is the fastest approach among the three. For the web traces, Submatch-OBDD's best performance (in Conf.C) is an order of magnitude faster than the other approaches' best performance (in Conf.S).

- PCRE suffers from backtracking algorithmic complexity attacks, while Submatch-OBDD and RE2 don't. With the web traces, the best time efficiency of PCRE was 3.67×10^4. However, PCRE was slowed down by two orders of magnitude when the synthetic trace was used, as is shown in Table 3(b). The reason is that the synthetic trace caused PCRE to perform heavily backtracking for some patterns.

- In Conf.C, the memory consumption of Submatch-OBDD and RE2 are comparative, while PCRE consumes the least memory. We do not report the memory requirements in Conf.S as the three approaches use very little memory for simple patterns.

We note that in Conf.S, RE2 is faster than Submatch-OBDD. This is because many patterns did not fill up the DFA state cache and hence did not trigger the DFA reconstruction process. In the case of simple patterns, the cost of OBDD operations, e.g., frontier derivation and existential quantification, is higher than the cost of several lookups on NFA transition table because the frontier size is often very small. Thus, Submatch-OBDD performs slower than RE2 in such situations. The cost of OBDD operations will be paid off when the frontier size of a tagged-NFA is large.

We recommend that Submatch-OBDD to be used in cases where a group of patterns are combined together. The performance boost of Submatch-OBDD is due to the *redundancy elimination*: The OBDD representation eliminates the redundancy in the Boolean representation of tagged NFAs.

Method	Conf.S	Conf.C	
	Exec-time	Exec-time	Memory (MB)
RE2	2.31×10^4	1.21×10^5	7.3
PCRE	3.67×10^4	1.13×10^6	1.2
OBDD	8.76×10^4	3.63×10^3	9.4

(a) Performance numbers with the web traces

Method	Conf.S	Conf.C	
	Exec-time	Exec-time	Memory (MB)
RE2	8.20×10^4	2.22×10^5	7.6
PCRE	1.44×10^6	1.40×10^6	1.0
OBDD	2.12×10^5	2.20×10^4	7.0

(b) Performance numbers with the synthetic trace

Table 3: Execution time (cycles/bytes) and memory consumption for the Snort-2009 data set with (a) the web traces and (b) the synthetic trace. In both traces, Submatch-OBDD's best execution time (Conf.C) is much shorter than RE2's and PCRE's best execution times (Conf.S).

4.3.2 *Snort-2012*

Table 4 shows the performance of RE2, PCRE, and Submatch-OBDD on the 403 patterns from Snort-2012 rule set.We have the following observations:

- Submatch-OBDD achieves its ideal time efficiency in Conf.C, i.e., when patterns are combined together for matching test and submatch extraction.
- For the web traces, Submatch-OBDD is faster than RE2, but slower than PCRE. While for the synthetic trace, Submatch-OBDD is faster than both RE2 and PCRE.
- Like in the Snort-2009 data set, PCRE suffers from the backtracking algorithmic attack performed by the synthetic trace. PCRE's time efficiency under the synthetic trace is two to three orders of magnitude than under the web traces.
- In Conf.C, the memory consumption of Submatch-OBDD and RE2 are comparative.

Although we observed that PCRE performed better for the web traces in Table 4, PCRE is still not recommended to be used as pattern matching engine for a network intrusion detection system (NIDS). The main reason is that it is easy for attackers to craft network traffic performing backtracking algorithmic attacks on PCRE, as was shown by Smith et al. in [19]. Our experimental results in Table 3 and Table 4 also demonstrated that PCRE is easily to be slowed down by hundreds of times with carefully crafted synthetic traces.

4.3.3 *Firewall-504*

Table 5 shows the three approaches' performance on the Firewall-504 data set. Submatch-OBDD is the fastest approach on this data set. In Conf.C, Submatch-OBDD is orders of magnitude faster than RE2 and PCRE. Also, Submatch-OBDD's best performance (in Conf.C) is 62% faster than RE2's best performance (in Conf.S). In memory usage, PCRE is most space compact. Submatch-OBDD consumes slightly more memory than RE2.

4.4 Discussion

During our evaluation, we found a small number regular expressions from the Snort 2009 and 2012 rule sets that can cause either PCRE or RE2 to perform poorly. For example, if we use PCRE to match

```
.*\x2F[^\s]*\.(dat|xml)\?[^\s]*v=[^\s]*t=[^\s]*c=
```

with input string
`/;/;/;.dat?;.dat?;.dat?;v=;v=;v=;t=;t=;t=;c`.
Then PCRE will perform $O(3\times3\times3\times3)$ backtracking evaluations before eventually concluding that the string does not match the pattern. The evaluation time of PCRE will increase exponentially if we increase the number of repetitions of the `/;`, `.dat?`, `v=;`, and `t=;` in the input string. We observed that when these substrings were repeated 20 times, the execution time of PCRE for this regular expression was in the order of 10^6 cycles/byte. Details on how to create pathological traces to exploit the backtracking behavior of PCRE can be found in [19].

RE2 can perform poorly under the case when the DFA states of a regular expression blow up. The blow-up will cause the limited state cache be filled quickly and RE2 has to empty the cache and restart the DFA construction. In our experiments, we have observed an individual regular expression from Snort-2009 where the time efficiency of RE2 is an order of magnitude slower than Submatch-OBDD, which does not suffer from state blow up as it is a NFA-based approach. We also found eight patterns from the SIEM system which cause RE2 to blow up in its DFA construction. For these patterns, the time efficiency of RE2 is an order of magnitude slower than Submatch-OBDD. For commercial reasons, we do not disclose these patterns in the paper.

Please note that RE2 and PCRE are mature and popular engines and their code bases are heavily optimized. We have not devoted significant time to try to optimize Submatch-OBDD. We believe that Submatch-OBDD's performance can be further improved with better optimization.

Method	Conf.S	Conf.C	
	Exec-time	Exec-time	Memory (MB)
RE2	4.79×10^4	2.09×10^6	15.0
PCRE	7.70×10^4	2.69×10^3	1.0
OBDD	3.83×10^5	1.08×10^4	6.3

(a) Performance numbers with the web traces

Method	Conf.S	Conf.C	
	Exec-time	Exec-time	Memory (MB)
RE2	2.92×10^5	8.21×10^6	15.0
PCRE	1.47×10^6	7.64×10^5	1.0
OBDD	4.70×10^5	1.10×10^5	15.3

(b) Performance numbers with the synthetic trace

Table 4: Execution time (cycles/bytes) and memory consumption for the Snort-2012 data set with (a) the web traces and (b) the synthetic trace.

Method	Conf.S	Conf.C	
	Exec-time	Exec-time	Memory (MB)
RE2	2.04×10^5	2.20×10^7	21.0
PCRE	6.88×10^5	1.60×10^6	1.1
OBDD	6.31×10^5	1.25×10^5	30.0

Table 5: Execution time (cycles/bytes) and memory consumption for the Firewall-504 data set.

5. RELATED WORK

Regular expressions are extensively used to construct attack signatures in NIDS and to process event logs in SIEM systems. Finite automata are natural representations for regular expressions. DFAs are fast, but suffer from state blow-up for certain types of regular expressions. NFAs are compact, but slow in operation. Many techniques have been proposed to improve DFAs' space efficiency: compression [2], determinization on-the-fly [23], building multiple DFAs (MDFA) from a group of signatures [28], extending DFAs with scratch memory (XFAs) [20, 21], and constructing DFA variants with hardware implementations [3, 13, 15]. Similarly, many techniques have been proposed to improve NFAs' time efficiency: hardware based parallelism [7, 15, 6, 12, 18] and software based speedup [26, 27]. Hybrid finite automata [1] combines the benefits of NFAs and DFAs.

Submatch extraction, however, has not received much attention from the research community. Pike implemented a submatch extraction approach in the `sam` text editor [17] using a straightforward modification of Thompson's NFA simulation [24]. Google's RE2 tool also uses the modified NFA simulation approach. Laurikari proposed TNFA, an NFA-based approach for submatch extraction, where an NFA is augmented with tags to represent capturing groups [14]. Our approach also uses tags, but we associate tags with non-ϵ transitions whereas TNFA associates tags with ϵ transitions. We use OBDDs to represent and operate on tagged NFAs to achieve time efficiency. We did not include TNFA in our experiments as we already compared with a mature NFA-based approach, RE2.

Java, PCRE, Perl, Python, Ruby, and many other tools implement pattern match and submatch extraction using recursive backtracking, where an input string may be scanned multiple times before a match is found. The backtracking approach's worst case performance is exponential running time [9]. These tools use backtracking to efficiently handle backreference, a non-regular construct that improves the pattern language's expressive power. In contrast, our Submatch-OBDD approach is an NFA-based technique and does not suffer from exponential running time.

Google's RE2 is an open source automata based pattern matching tool that supports submatch extraction [10]. RE2 employs a DFA approach to test whether an input string matches a pattern. If a pattern contains capturing groups, RE2 uses a DFA approach to find the pattern's overall match in an input string and then runs an NFA approach to extract the submatches in the overall match. Similar to RE2, our Submatch-OBDD is NFA-based. We, however, use OBDDs to perform NFA operations and hence improve time efficiency. Submatch-OBDD performs better than RE2 when patterns are combined. Both RE2 and Submatch-OBDD do not support backreferences.

Yang et al.'s NFA-OBDD model [26, 27] is the most relevant work to Submatch-OBDD. A commonality between NFA-OBDD and our Submatch-OBDD is the use of "implicit state enumeration" by means of OBDDs [8, 25]. NFA-OBDD, however, does not support submatch extraction. In our work, we propose a new approach to tag capturing groups in a regular expression, and extend Thompson's NFA construction to support capturing groups. We propose a novel submatch extraction approach using OBDDs.

6. CONCLUSION

We present Submatch-OBDD, which allows fast submatch extraction in regular expression-like pattern matching. We propose a new approach to tag capturing groups in a regular expression, and extend Thompson's NFA construction approach to support regular expressions with capturing groups. We present a novel technique to perform submatch extraction. Our use of OBDDs improves the time efficiency of match test and submatch extraction. We evaluated our Submatch-OBDD implementation using patterns used in the Snort NIDS and a commercial SIEM system. Our experiments on real network traces, synthetic traces, and enterprise event logs show that Submatch-OBDD achieves its ideal performance when patterns are combined. In the best case, our approach is faster than RE2 and PCRE by one to two orders of mangintude.

Acknowledgments

This work was supported in part by NSF grants CNS-0952128 and CNS-1117711. We thank the anonymous ANCS reviewers for helpful comments on an earlier draft of this paper.

7. REFERENCES

[1] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the 2007 ACM CoNEXT conference*, CoNEXT '07, pages 1:1–1:12, New York, NY, USA, 2007. ACM.

[2] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ANCS '07, pages 145–154, New York, NY, USA, 2007. ACM.

[3] B. C. Brodie, D. E. Taylor, and R. K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *Intl. Symp. Computer Architecture*, pages 191–202. IEEE Computer Society, 2006.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35:677–691, August 1986.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Symp. on Logic in Computer Science*, pages 401–424. IEEE Computer Society, 1990.

[6] D. Chasaki and T. Wolf. Fast regular expression matching in hardware using nfa-bdd combination. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 12:1–12:2, New York, NY, USA, 2010. ACM.

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

[8] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 365–373, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[9] R. Cox. Regular expression matching can be simple and fast. `http://swtch.com/~rsc/regexp/regexp1.html`, 2007.

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

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison Wesley, 2001.

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

[13] 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.

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

[15] C. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In *19th USENIX Security Symposium*, August 2010.

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

[17] R. Pike. The text editor sam. *Softw. Pract. Exper.*, 17:813–845, November 1987.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] F. Somenzi. CUDD: CU decision diagram package, release 2.4.2. Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder. `http://vlsi.colorado.edu/$\sim$fabio/CUDD`.

[23] R. Sommer and V. Paxson. Enhancing byte-level network intrusion detection signatures with context. In *CCS'03*, pages 262–271. ACM, 2003.

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

[25] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *IEEE International Conference on Computer-Aided Design*, pages 130–133, Santa Clara, CA, 1990. IEEE.

[26] 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.

[27] 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.

[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.