# Improving NFA-based Signature Matching using Ordered Binary Decision Diagrams[⋆]

Liu Yang[*], Rezwana Karim[*], Vinod Ganapathy[*], Randy Smith[†]

[*]Rutgers University [†]Sandia National Laboratories

**Abstract.** Network intrusion detection systems (NIDS) make extensive use of regular expressions as attack signatures. Internally, NIDS represent and operate these signatures using finite automata. Existing representations of finite automata present a well-known time-space tradeoff: Deterministic automata (DFAs) provide fast matching but are memory intensive, while non-deterministic automata (NFAs) are space-efficient but are several orders of magnitude slower than DFAs. This time/space tradeoff has motivated much recent research, primarily with a focus on improving the space-efficiency of DFAs, often at the cost of reducing their performance.

This paper presents NFA-OBDDs, a symbolic representation of NFAs that retains their space-efficiency while improving their time-efficiency. Experiments using Snort HTTP and FTP signature sets show that an NFA-OBDD-based representation of regular expressions can outperform traditional NFAs by up to three orders of magnitude and is competitive with a variant of DFAs, while still remaining as compact as NFAs.

**Key words:** NIDS, signature matching, ordered binary decision diagrams.

## 1 Introduction

Deep packet inspection allows network intrusion detection systems (NIDS) to accurately identify malicious traffic by matching the contents of network packets against attack signatures. In the past, attack signatures were keywords that could efficiently be matched using string matching algorithms. However, the increasing complexity of network attacks has lead the research community to investigate richer signature representations, which require the full power of regular expressions. Because NIDS are often deployed over high-speed network links, algorithms to match such rich signatures must also be efficient enough to provide high-throughput intrusion detection on large volumes of network traffic. This problem has spurred much recent research, and in particular has lead to the investigation of new representations of regular expressions that allow for efficient inspection of network traffic (*e.g.,* [1,2,3,4]).

To be useful for deep packet inspection in a NIDS, any representation of regular expressions must satisfy two key requirements: *time-efficiency* and *space-efficiency*. Time-efficiency requires the amount of time spent by the NIDS to process each byte of network traffic to be small, thereby allowing large volumes of traffic to be matched

[⋆] Supported in part by NSF grants 0831268, 0915394, 0931992 and 0952128. L. Yang and R. Karim contributed equally, while R. Smith contributed while at the University of Wisconsin.

quickly. Space-efficiency requires the size of the representation to be small, thereby ensuring that it will fit within the main memory of the NIDS. Space-efficiency also mandates that the size of the representation should grow proportionally (*e.g.,* linearly) with the number of attack signatures. This requirement is important because the increasing diversity of network attacks has lead to a quick growth in the number of signatures used by NIDS. For example, the number of signatures in Snort [5] has grown from $3{,}166$ in $2003$ to $15{,}047$ in $2009$.

Finite automata are a natural representation for regular expressions, but offer a tradeoff between time- and space-efficiency. Using deterministic finite automata (DFAs) to represent regular expressions allows efficient matching ($O(1)$ lookups to its transition table to process each input symbol), while non-deterministic finite automata (NFAs) can take up to $O(n)$ transition table lookups to process each input symbol, where $n$ is the number of states in the NFA. However, NFAs are space-efficient, while DFAs for certain regular expressions can be exponentially larger than the corresponding NFAs [6]. More significantly, combining NFAs only leads to an additive increase in the number of states, while combining DFAs can result in a multiplicative increase, *i.e.,* an NFA that combines two NFAs with $m$ and $n$ states has up to $O(m + n)$ states, while a DFA that combines two DFAs with $m$ and $n$ states can have up to $O(m \times n)$ states. DFA representations for large sets of regular expressions often consume several gigabytes of memory, and do not fit within the main memory of most NIDS.

This time/space tradeoff has motivated much recent research, primarily with a focus on improving the space-efficiency of DFAs. These include heuristics to compress DFA transition tables (*e.g.,* [2,7]), techniques to combine regular expressions into multiple DFAs [4], and variable extended finite automata (XFAs) [3,8], which offer compact DFA representations and guarantee an additive increase in states when signatures are combined, provided that the regular expressions satisfy certain conditions. These techniques trade time for space, and though the resulting representations fit in main memory, their matching algorithms are slower than those for traditional DFAs.

In this paper, we take an alternative approach and instead focus on *improving the time-efficiency of NFAs.* NFAs are not currently in common use for deep packet inspection, and understandably so—their performance can be several orders of magnitude slower than DFAs. Nevertheless, NFAs offer a number of advantages over DFAs, and we believe that further research on improving their time-efficiency can make them a viable alternative to DFAs. Our position is supported in part by these observations:

- **NFAs are more compact than DFAs.** Determinizing an NFA involves a subset construction algorithm, which can result in a DFA with exponentially more states than an equivalent NFA [6].
- **NFA combination is space-efficient.** Combining two NFAs simply involves linking their start states together by adding new $\epsilon$ transitions; the combined NFA is therefore only as large as the two NFAs put together. This feature of NFAs is particularly important, given that the diversity of network attacks has pushed NIDS vendors to deploy an ever increasing number of signatures. In contrast, combining two DFAs can result in a multiplicative increase in the number of states, and the combined DFA may be much larger than its constituent DFAs.

- **NFAs can readily be parallelized.** An NFA can be in a set of states (called the *frontier*) at any instant during its operation, each of which may contain multiple outgoing transitions for an input symbol. States in the frontier can be processed in parallel as new input symbols are encountered [9,10].

Motivated by these advantages, we develop a new approach to improve the time-efficiency of NFAs. The frontier of an NFA can contain $O(n)$ states, each of which must be processed using the NFA's transition relation for each input symbol to compute a new frontier, thereby resulting in slow operation. Although this frontier can be processed in parallel to improve performance, NFAs for large signature sets may contain several thousand states in their frontier at any instant. Commodity hardware is not yet well-equipped to process such large frontiers in parallel.

Our core insight is that a technique to efficiently apply an NFA's transition relation to a *set of states* can greatly improve the time-efficiency of NFAs. Such a technique would apply the transition relation to all states in the frontier in a single operation to produce a new frontier. We develop an approach that uses *Ordered Binary Decision Diagrams* [11] (OBDDs) to implement such a technique. Our use of OBDDs to process NFA frontiers is inspired by symbolic model checking, where the use of OBDDs allows the verification of systems that contain an astronomical number of states [12]. NFAs that use OBDDs (**NFA-OBDDs**) can be constructed from regular expressions in a fully automated way, and are robust in the face of structural complexities in these regular expressions (*e.g.,* counters [8, Section 6.2]).

To evaluate the feasibility of our approach, we constructed NFAs in software using HTTP and FTP signatures from Snort. We operated these NFAs using OBDDs and evaluated their time-efficiency and space-efficiency using HTTP and FTP traffic obtained from our department's network. Our experiments showed that NFA-OBDDs outperform traditional NFAs by approximately *three orders of magnitude*—about $1645\times$ in the best case. Our experiments also showed that NFA-OBDDs retain the space-efficiency of NFAs. In contrast, our machine ran out of memory when trying to construct DFAs (or their variants) from our signature sets.

Our main contributions are as follows:

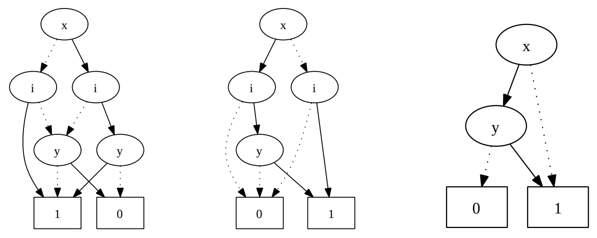
- **Design of NFA-OBDDs.** We develop a novel technique that uses OBDDs to improve the time-efficiency of NFAs (Section 3). We also describe how NFA-OBDDs can be used to improve the time and space-efficiency of NFA-based multi-byte matching (Section 5).
- **Comprehensive evaluation using Snort signatures.** We evaluated NFA-OBDDs using Snort's HTTP and FTP signature sets and observed a speedup of about three orders of magnitude over traditional NFAs. We also compared the performance of NFA-OBDDs against a variety of automata implementations, including the PCRE package and a variant of DFAs (Section 4).

## 2 Ordered Binary Decision Diagrams

An OBDD is a data structure that can represent arbitrary Boolean formulae. OBDDs transform Boolean function manipulation into efficient graph transformations, and have found wide use in a number of application domains. For example, OBDDs are used

| *x* | *i* | *y* | *f(x, i, y)* |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

**(a) A Boolean function** $f(x, i, y)$.



**(b) OBDD($f$) with** $x < i < y$. **(c) APPLY($\wedge$, OBDD($f$), OBDD($I(i)$)).** **(d) RESTRICT(OBDD($f$),** $i \leftarrow 1$**).**

**Fig. 1. An example of a Boolean formula, its OBDD, and various operations on the OBDD. Solid edges are labeled 1, dotted edges are labeled 0.**

extensively by model checkers to improve the efficiency of state-space exploration algorithms [12]. In this section, we present an informal overview of OBDDs, and refer interested readers to Bryant's seminal article [11] for details.

An OBDD represents a Boolean function $f(x_1, x_2, \ldots, x_n)$ as a rooted, directed acyclic graph (DAG). The DAG has two *terminal* nodes, which are labeled **0** and **1**, and have no outgoing edges. Each remaining *non-terminal* node is associated with a label $\in$ $\{x_1, x_2, \ldots, x_n\}$, and has two outgoing edges labeled **0** and **1**. An OBDD is ordered in the sense that node labels are associated with a total order $<$. Node labels along all paths in the OBDD from the root to the terminal nodes follow this total order.[1] To evaluate the Boolean formula denoted by an OBDD, it suffices to traverse appropriately labeled edges from the root to the terminal nodes of the DAG. Figure 1(b) depicts an example of an OBDD for the Boolean formula $f(x, i, y)$ shown in Figure 1(a) with the variable ordering $x < i < y$.

OBDDs allow Boolean functions to be manipulated efficiently. With OBDDs, checking the satisfiability (or unsatisfiability) of a Boolean formula is a constant time operation, because it suffices to check whether the terminal node labeled **1** (respectively, **0**) is present in the OBDD. The APPLY and RESTRICT operations [11], described below, allow OBDDs to be combined and modified with a number of Boolean operators. These two operations are implemented as a series of graph transformations and reductions to the input OBDDs, and have efficient implementations (their time complexity is polynomial in the size of the input OBDDs).

APPLY allows binary Boolean operators, such as $\wedge$ and $\vee$, to be applied to a pair of OBDDs. The two input OBDDs, OBDD($f$) and OBDD($g$), must have the same variable ordering. APPLY(OP, OBDD($f$), OBDD($g$)) computes OBDD($f$ OP $g$), which has the same variable ordering as the input OBDDs. Figure 1(c) presents the OBDD obtained by combining the OBDD in Figure 1(b) with OBDD($I(i)$), where $I$ is the identity function. The RESTRICT operation is unary, and produces as output an OBDD in which the values of some of the variables of the input OBDD have been fixed to a certain value. That is, RESTRICT(OBDD($f$), $x \leftarrow k$) = OBDD($f|_{(x \leftarrow k)}$), where $f|_{(x \leftarrow k)}$ denotes that $x$ is assigned the value $k$ in $f$. In this case, the output OBDD does not have

[1] DAGs denoting OBDDs satisfy additional properties, as described in Bryant's article. However, they are not directly relevant for this discussion, and we elide them for brevity.

any nodes with the label $x$. Figure 1(d) shows the OBDD obtained as the output of RESTRICT($\mathsf{OBDD}(f)$, $i \leftarrow 1$), where $\mathsf{OBDD}(f)$ is the OBDD of Figure 1(b).

APPLY and RESTRICT can be used to implement existential quantification, which is used in a key way in the operation of NFA-OBDDs, as described in Section 3. In particular, $\exists x_i.f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)|_{(x_i \leftarrow 0)} \vee f(x_1,\ldots,x_n)|_{(x_i \leftarrow 1)}$. Therefore, we have: $\mathsf{OBDD}(\exists\, x_i.f(x_1,\ldots,x_n))$ = APPLY($\vee$, RESTRICT($\mathsf{OBDD}(f)$, $x_i \leftarrow 1$), RESTRICT($\mathsf{OBDD}(f)$, $x_i \leftarrow 0$)). Note that $\mathsf{OBDD}(\exists\, x_i.f(x_1,\ldots,x_n))$ will not have a node labeled $x_i$.

**Representing Relations and Sets.** OBDDs can be used to represent relations of arbitrary arity. If $R$ is an $n$-ary relation over the domain $\{0,\, 1\}$, then we define its *characteristic function* $f_R$ as follows: $f_R(x_1,\ldots,x_n) = 1$ if and only if $R(x_1,\ldots,x_n)$. For example, the characteristic function of the 3-ary relation $R = \{(1,0,1), (1,1,0)\}$ is $f_R(x_1,x_2,x_3) = (x_1 \wedge \bar{x_2} \wedge x_3) \vee (x_1 \wedge x_2 \wedge \bar{x_3})$. $f_R$ is a Boolean function and can therefore be expressed using an OBDD.

An $n$-ary relation $Q$ over an arbitrary domain $D$ can be similarly expressed using OBDDs by *bit-blasting* each of its elements. That is, if the domain $D$ has $m$ elements, we map each of its elements uniquely to bit-strings containing $\lceil \lg m \rceil$ bits (call this mapping $\phi$). We then define a new relation $R(\phi(x_1),\ldots,\phi(x_n)) = Q(x_1,\ldots,x_n)$. $R$ is a $n \times \lceil \lg m \rceil$-ary relation over $\{0,1\}$, and can be converted into an OBDD using its characteristic function.

A set of elements over an arbitrary domain $D$ can also be expressed as an OBDD because sets are unary relations, *i.e.,* if $S$ is a set of elements over a domain $D$, then we can define a relation $R_S$ such that $R_S(s) = 1$ if and only if $s \in S$. Operations on sets can then be expressed as Boolean operations and performed on the OBDDs representing these sets. For example, $S \subseteq T$ can be implemented as $\mathsf{OBDD}(S) \longrightarrow \mathsf{OBDD}(T)$ (logical implication), while ISEMPTY($S \cap T$) is equivalent to checking whether $\mathsf{OBDD}(S) \wedge \mathsf{OBDD}(T)$ is satisfiable. The conversion of relations and sets into OBDDs is used in a key way in the construction and operation of NFA-OBDDs, which we describe next.

## 3 Representing and Operating NFAs

We represent an NFA using a 5-tuple: ($Q$, $\Sigma$, $\Delta$, $q_0$, *Fin*), where $Q$ is a finite set of states, $\Sigma$ is a finite set of input symbols (the alphabet), $\Delta$: $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is a transition function, $q_0 \in Q$ is a start state, and *Fin* $\subseteq Q$ is a set of accepting (or final) states. The transition function $\Delta(s,\, i) = T$ describes the set of all states $t \in T$ such that there is a transition labeled $i$ from $s$ to $t$. Note that $\Delta$ can also be expressed as a relation $\delta$: $Q \times \Sigma \times Q$, so that $(s,i,t) \in \delta$ for all $t \in T$ such that $\Delta(s,i) = T$. We will henceforth use $\delta$ to denote the set of transitions in the NFA.

An NFA may have multiple outgoing transitions with the same input symbol from each state. Hence, it maintains a *frontier* $F$ of states that it can currently be in. The frontier is initially the singleton set $\{q_0\}$ but may include any subset of $Q$ during the operation of the NFA. For each symbol in the input string, the NFA must process all of the states in $F$ and find a new set of states by applying the transition relation.

While non-determinism leads to frontiers of size $O(|Q|)$ in NFAs, it also makes them space-efficient in two ways. First, NFAs for certain regular expressions are exponentially smaller than the corresponding DFAs, *e.g.,* an NFA for `(0|1)*1(0|1)`$^n$ has $O(n)$ states, while the corresponding DFA has $O(2^n)$ states [6]. Second, and perhaps more significant from the perspective of NIDS, NFAs can be combined space-efficiently while DFAs cannot. To combine a pair of NFAs, NFA$_1$ and NFA$_2$, it suffices to create a new state $q_{new}$, add $\epsilon$ transitions from $q_{new}$ to the start states of NFA$_1$ and NFA$_2$, and designate $q_{new}$ to be the start state of the combined NFA. This leads to an NFA with $O(|Q_1| + |Q_2|)$ states. In contrast, combining two DFAs, DFA$_1$ and DFA$_2$, results in a multiplicative increase in the number of states because the combined DFA must have a state corresponding to $s \times t$ for each pair of states $s$ and $t$ in DFA$_1$ and DFA$_2$, respectively. The number of states in the DFA can possibly be reduced using minimization, but this does not always help. For example, the DFAs for the regular expressions `ab`$^*$`cd`$^*$ and `ef`$^*$`gh`$^*$ have 5 states and 6 transitions each, and the combined DFA (minimized) has 16 states and 22 transitions.

**NFA Operation using Boolean Functions.** We now describe how the process of applying an NFA's transition relation to a frontier of states can be expressed as a sequence of Boolean function manipulations. NFA-OBDDs implement Boolean functions and operations on them using BDDs. For the discussion below and in the rest of this paper, we assume NFAs in which $\epsilon$ transitions have been eliminated (using standard techniques [6]). This is mainly for ease of exposition; NFAs with $\epsilon$ transitions can also be expressed using NFA-OBDDs. Note that $\epsilon$ elimination may increase the total number of transitions in the NFA, but does not increase the number of states.

We now define four Boolean functions for an NFA ($Q$, $\Sigma$, $\delta$, $q_0$, *Fin*). These functions use three vectors of Boolean variables: $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{i}$. The vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ are used to denote states in $Q$, and therefore contain $\lceil \lg |Q| \rceil$ variables each. The vector $\boldsymbol{i}$ denotes symbols in $\Sigma$, and contains $\lceil \lg |\Sigma| \rceil$ variables. As an example, for the NFA in Figure 2, these vectors contain one Boolean variable each; we denote them as $x$, $y$, and $i$.

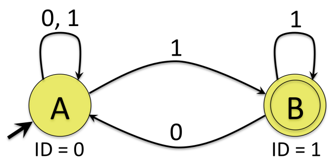


**Fig. 2. NFA for `(0|1)*1`.**

- $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ denotes the NFA's transition relation $\delta$. Recall that $\delta$ is a set of triples $(s, i, t)$, such that there is a transition labeled $i$ from state $s$ to state $t$. It can therefore be represented as a Boolean function as described in Section 2. For example, consider the NFA in Figure 2. Using 0 to denote state A and 1 to denote state B, $\mathcal{T}(x, i, y)$ is the function shown in shown in Figure 1(a).
- $\mathcal{I}_\sigma(\boldsymbol{i})$ is defined for each $\sigma \in \Sigma$, and denotes a Boolean representation of that symbol. For the NFA in Figure 2, $\mathcal{I}_0(i) = \bar{i}$ (*i.e.,* $i$ = 0) and $\mathcal{I}_1(i) = i$.
- $\mathcal{F}(\boldsymbol{x})$ denotes the current set of frontier states of the NFA. It is thus a Boolean representation of the set $F$ at any instant during the operation of the NFA. For the example in Figure 2, if $F = \{$A$\}$, $\mathcal{F}(x) = \bar{x}$, while if $F = \{$A, B$\}$, then $\mathcal{F}(x) = x \vee \bar{x}$.
- $\mathcal{A}(\boldsymbol{x})$ is a Boolean representation of *Fin*, and denotes the accepting states. In Figure 2, $\mathcal{A}(x) = x$.

Note that $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$, $\mathcal{I}_\sigma(\boldsymbol{i})$ and $\mathcal{A}(\boldsymbol{x})$ can be computed automatically from any representation of NFAs. The initial frontier $F = \{q_0\}$ can also be represented as a Boolean formula. Suppose that the frontier at some instant during the operation of the NFA is $\mathcal{F}(\boldsymbol{x})$, and that the next symbol in the input is $\sigma$. The following Boolean formula, $\mathcal{G}(y)$, symbolically denotes the new frontier of states in the NFA after $\sigma$ has been processed.

$$\mathcal{G}(\boldsymbol{y}) = \exists\, \boldsymbol{x}.\exists\, \boldsymbol{i}.[\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y}) \wedge \mathcal{I}_\sigma(\boldsymbol{i}) \wedge \mathcal{F}(\boldsymbol{x})]$$

To see why $\mathcal{G}(\boldsymbol{y})$ is the new frontier, consider the truth table of the Boolean function $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$. By construction, this function evaluates to 1 only for those values of $\boldsymbol{x}$, $\boldsymbol{i}$, and $\boldsymbol{y}$ for which $(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ is a transition in the automaton. Similarly, the function $\mathcal{F}(\boldsymbol{x})$ evaluates to 1 only for the values of $\boldsymbol{x}$ that denote states in the current frontier of the NFA. Thus, the conjunction of $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ with $\mathcal{F}(\boldsymbol{x})$ and $\mathcal{I}_\sigma(\boldsymbol{i})$ only "selects" those rows in the truth table of $\mathcal{T}(\boldsymbol{x}, \boldsymbol{i}, \boldsymbol{y})$ that correspond to the outgoing transitions from states in the frontier labeled with the symbol $\sigma$. However, the resulting conjunction is a Boolean formula in $\boldsymbol{x}$, $\boldsymbol{i}$ and $\boldsymbol{y}$. To find the new frontier of states, we are only interested in the values of $\boldsymbol{y}$ (*i.e.,* the target states of the transitions) for which the conjunction has a satisfying assignment. We achieve this by existentially quantifying $\boldsymbol{x}$ and $\boldsymbol{i}$ to obtain $\mathcal{G}(\boldsymbol{y})$. To express the new frontier in terms of the Boolean variables in $\boldsymbol{x}$, we rename the variables in $\boldsymbol{y}$ with the corresponding ones in $\boldsymbol{x}$.

We illustrate this idea using the example in Figure 2. Suppose that the current frontier of the NFA is $F = \{$A, B$\}$, and that the next input symbol is a 0, which causes the new frontier to become $\{$A$\}$. In this case, $\mathcal{T}(x, i, y)$ is the function shown in Figure 1(a), $\mathcal{I}_0(i) = \bar{i}$ and $\mathcal{F}(x) = x \vee \bar{x}$. We have $\mathcal{T}(x, i, y) \wedge \mathcal{I}_0(i) \wedge \mathcal{F}(x) = (x \wedge \bar{i} \wedge \bar{y})$. Existentially quantifying $x$ and $i$ from the result of this conjunction, we get $\mathcal{G}(y) = \bar{y}$. Renaming the variable $y$ to $x$, we get $\mathcal{F}(x) = \bar{x}$, which is a Boolean formula that denotes $\{$A$\}$, the new frontier.

To determine whether the NFA accepts an input string, it suffices to check that $F \cap \mathit{Fin} \neq \emptyset$. Using the Boolean notation, this translates to check whether $\mathcal{F}(\boldsymbol{x}) \wedge \mathcal{A}(\boldsymbol{x})$ has a satisfying assignment. In the example above with $F = \{$A$\}$, $\mathcal{F}(x) = \bar{x}$ and $\mathcal{A}(x) = x$, so the NFA is not in an accepting configuration. Recall that checking satisfiability of a Boolean function is an $O(1)$ operation if the function is represented as an OBDD.

**NFA-OBDDs.** The main idea behind NFA-OBDDs is to represent and manipulate the Boolean functions discussed above using OBDDs. Formally, an NFA-OBDD for an NFA ($Q$, $\Sigma$, $\delta$, $q_0$, *Fin*) is a 7-tuple ($\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{y}$, $\mathsf{OBDD}(\mathcal{T})$, $\{\mathsf{OBDD}(\mathcal{I}_\sigma \mid \forall\sigma \in \Sigma)\}$, $\mathsf{OBDD}(\mathcal{F}_{q_0})$, $\mathsf{OBDD}(\mathcal{A})$), where $\boldsymbol{x}$, $\boldsymbol{i}$, $\boldsymbol{y}$ are vectors of Boolean variables, and $\mathcal{T}$, $\mathcal{I}_\sigma$, and $\mathcal{A}$ are the Boolean formulae discussed earlier. $\mathcal{F}_{q_0}$ denotes the Boolean function that denotes the frontier $\{q_0\}$. For each input symbol $\sigma$, the NFA-OBDD obtains a new frontier as discussed earlier. The main difference is that the Boolean operations are performed as operations on OBDDs.

The use of OBDDs allows NFA-OBDDs to be more time-efficient than NFAs. In an NFA, the transition table must be consulted for each state in the frontier, leading to $O(|\delta| \times |F|)$ operations per input symbol. In contrast, the complexity of OBDD operations to obtain a new frontier is $O(\textsc{sizeof}(\mathsf{OBDD}(\mathcal{T})) \times \textsc{sizeof}(\mathsf{OBDD}(\mathcal{F})))$. Because OBDDs are a compact representation of the frontier $F$ and the transition relation $\delta$, NFA-OBDDs are more time-efficient than NFAs. The improved performance of

NFA-OBDDs is particularly pronounced when the transition table of the NFA is sparse or the NFA has large frontiers. This is because OBDDs can effectively remove redundancy in the represention of $\delta$ and $F$.

NFA-OBDDs retain the space-efficiency of NFAs because NFA-OBDDs can be combined using the same algorithms that are used to combine NFAs. Although the use of OBDDs may lead NFA-OBDDs to consume more memory than NFAs, our experiments show that the increase is marginal. In particular, the cost is dominated by $\mathsf{OBDD}(\mathcal{T})$, which has a total of $2 \times \lceil \lg |Q| \rceil + \lceil \lg |\Sigma| \rceil$ Boolean variables. Even in the worst case, this OBDD consumes only $O(|Q|^2 \times |\Sigma|)$ space, which is comparable to the worst-case memory consumption of the transition table of a traditional NFA. However, in practice, the memory consumption of NFA-OBDDs is much smaller than this asymptotic limit.

## 4 Implementation and Evaluation

We evaluated the feasibility of our approach using a software-based implementation of NFA-OBDDs. As depicted in Figure 3, the implementation consists of two offline components and an online component.

The offline components are executed once for each set of regular expressions, and consist of re2nfa and nfa2obdd. The re2nfa component accepts a set of regular expressions as input, and produces an $\epsilon$-free NFA as output. To do so, it first constructs NFAs for each of the regular expressions using Thompson's construction [13,6], combines these NFAs into a single NFA, and eliminates $\epsilon$ transitions. The nfa2obdd component analyzes this NFA to determine the number of Boolean variables needed (*i.e.,* the sizes of the $\boldsymbol{x}$, $\boldsymbol{i}$ and $\boldsymbol{y}$ vectors), and constructs $\mathsf{OBDD}(\mathcal{T})$, $\mathsf{OBDD}(\mathcal{A})$, $\mathsf{OBDD}(\mathcal{I}_\sigma)$ for each $\sigma \in \Sigma$, and $\mathsf{OBDD}(\mathcal{F}_{q_0})$.

It is well-known that the size of an OBDD for a Boolean formula is sensitive to the total order imposed on its variables. For the NFA-OBDDs used in our experiments, we empirically determined that an ordering of variables of the form $\boldsymbol{i} < \boldsymbol{x} < \boldsymbol{y}$ yields the best performance for NFA-OBDDs. For example, we found that an implementation of NFA-OBDDs that uses the variable ordering $\boldsymbol{x} < \boldsymbol{i} < \boldsymbol{y}$ operates more than an order of magnitude slower than one that uses the ordering $\boldsymbol{i} < \boldsymbol{x} < \boldsymbol{y}$; we therefore used the latter ordering in our implementation. Within each vector, nfa2obdd uses a simple sorting scheme to order variables. Although it is NP-hard to choose a total order that yields the most compact OBDD for a Boolean function [11], future work could develop heuristics that leverage the structure of the input regular expressions to determine orderings that yield high-performance NFA-OBDDs.

The online component, exec_nfaobdd, begins execution by reading these OBDDs into memory and processes a stream of network packets. It matches the payloads of these network packets against the regular expressions using the NFA-OBDD. To manipulate OBDDs and produce a new frontier for each input symbol processed, this component interfaces with Cudd, a popular C++-based OBDD library [14]. It checks whether each frontier $\mathcal{F}$ produced during the operation of the NFA-OBDD contains an accepting state. If so, it emits a warning with the offset of the character in the input stream that triggered a match, as well as the regular expression(s) that matched the input. Note that in a NIDS setting, it is important to check whether the frontier $\mathcal{F}$ obtained after

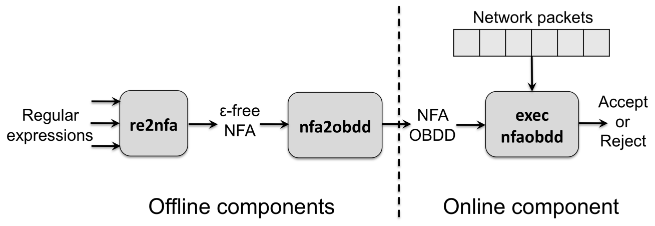


**Fig. 3. Components of our software-based implementation of NFA-OBDDs.**

processing *each* input symbol contains an accepting state. This is because *any* byte in the network input may cause a transition in the NFA that triggers a match with a regular expression. We call this the *streaming* model because the NFA continuously processes input symbols from a network stream.

**Data Sets.** We evaluated our implementation of NFA-OBDDs with three sets of regular expressions [15]. The first set was obtained from the authors of the XFA paper [8], and contains 1503 regular expressions that were synthesized from the March 2007 snapshot of the Snort HTTP signature set. The second and third sets, numbering 2612 and 98 regular expressions, were obtained from the October 2009 snapshot of the Snort HTTP and FTP signature sets, respectively. About 50% of these regular expressions were taken from the `uricontent` fields of the signatures, while the rest were extracted from the `pcre` fields. Although extracting just `pcre` fields from individual Snort rules only captures a portion of the corresponding signatures, it suffices for our experiments, because our primary goal is to evaluate the performance of NFA-OBDDs against other regular-expression based techniques. All three sets of regular expressions include client-side and server-side signatures. For all sets, we excluded Snort signatures that contained non-regular constructs, such as back-references and subroutines (which are allowed by PCRE [16]), because these constructs cannot be implemented in NFA-based models. In all, we excluded 1837 HTTP and 41 FTP signatures due to non-regular constructs.

To evaluate the performance of HTTP signatures, we fed traces of live HTTP traffic obtained from our department's network to exec_nfaobdd. We collected this traffic over a one week period in August 2009. This traffic was collected using tcpdump, and includes whole packets of port 80 traffic from our departmental Web server and our lab's machines.

The traffic observed during this period consisted largely of Web traffic typically observed at an academic department's main Web server; most of the traffic was to view and query Web pages hosted by the department. Overall, we observed connections from 18,618 distinct source IP addresses during this period, with 653,670 GET, 137,737 POST, 3,504 HEAD, and 1,576 PUT commands. This traffic triggered 1,816,410 and 17,107,588 matches in the HTTP/1503 and HTTP/2612 signature sets, corresponding to 47 and 120 distinct signatures, respectively.[2] The payloads in these packets ranged in size from 1 byte to 1460 bytes, with an average of 126 bytes (standard deviation

[2] These numbers are not indicative of the number of alerts produced by Snort because our signature sets only contain patterns from the `pcre` and `uricontent` fields of the Snort rules. The large number of matches is because signatures contained patterns common in HTTP packets.

of 271). However, we partitioned this traffic into 33 traces of various sizes, containing between 5.1MB–1.24GB worth of data. We did so because the the NFA and PCRE-based implementations discussed in this section were too slow to process the weeklong traffic trace. The size distribution of these traces was as follows: 21 traces of 5.1-7.2MB, 9 traces of 10.3-20.1MB, and one trace each of 227.2MB, 575.8MB, and 1.24GB.

We evaluated the FTP signatures using two traces of live FTP traffic (from the command channel), obtained over a two week period in March 2010 from our department's FTP server; these FTP traces contained 19.4MB and 24.7MB worth of data. The traffic consisted of FTP requests to fetch and update technical reports hosted by our department. We observed traffic from 528 distinct source IP addresses during this period. Statistics on various FTP commands observed during this period appear in the table below (commands that were not observed are not reported). This traffic triggered 9,656 and 15,976 matches in the FTP/98 signature set, corresponding to matches on 6 and 5 distinct signatures, respectively. The payload sizes of packets ranged from 2 to 402 bytes with an average of 40 bytes (standard deviation of 44).

| **Command** | CWD | LIST | MDTM | MKD | PASS | PORT | PWD | QUIT | RETR | SIZE | STOR | TYPE | USER |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Number** | 62,561 | 3,098 | 613 | 89 | 14,701 | 232 | 453 | 12,244 | 7,676 | 1,110 | 1,401 | 12,201 | 14,834 |

We also used synthetic traces during our experiments, but do not report these results in the paper because they are substantially similar to those obtained using real traffic. Because our primary goal is to study the performance of NFA-OBDDs, we assume that the network traces have been processed using standard NIDS operations, such as defragmentation and normalization. We fed these traces, which were in tcpdump format, to exec_nfaobdd.

**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 memory (however, our programs are single-threaded, and only used one of the available cores). We used the Linux `/proc` file system to measure the memory consumption of nfa2obdd and the Cudd ReadMemoryInUse utility to obtain the memory consumption of exec_nfaobdd. We instrumented both these programs to report their execution time using processor performance counters. We report the performance of exec_nfaobdd as the number of CPU cycles to process each byte of network traffic (cycles/byte), *i.e.,* fewer processing cycles/byte imply greater time-efficiency. All our implementations were in C++; we used the GNU `g++` compiler suite (v4.3.2) with the O6 optimization level to produce the executables used for experimentation.

Our experiments show that NFA-OBDDs: (**1**) outperform traditional NFAs by up to three orders of magnitude while retaining their space-efficiency; (**2**) outperform or are competitive in performance with the PCRE package, a popular library for regular expression matching; (**3**) are competitive in performance with variants of DFAs while being drastically less memory-intensive.

**Constructing NFA-OBDDs.** We used nfa2obdd to construct NFA-OBDDs from $\epsilon$-free NFAs of the regular expression sets. Figure 4 presents statistics on the sizes of the input NFAs, the size of the largest of the four OBDDs in the NFA-OBDD (OBDD($\mathcal{T}$)), and the time taken and memory consumed by nfa2obdd. For the NFA-OBDDs corresponding to the HTTP signature sets, the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ had 18 Boolean variables

| | | Size of the input NFA | | $\lvert \mathsf{OBDD}(\mathcal{T}) \rvert$ | Construction |
|---|---|---|---|---|---|
| **Signature Set** | **#Reg. Exps.** | **#States** | **#Transitions** | **#Nodes** | **Time/Memory** |
| HTTP (March 2007) | 1503 | 159,734 | 3,986,769 | 659,981 | 305sec/176MB |
| HTTP (October 2009) | 2612 | 239,890 | 5,833,911 | 989,236 | 453sec/176MB |
| FTP (October 2009) | 98 | 26,536 | 5,927,465 | 69,619 | 246sec/134MB |

**Fig. 4. NFA-OBDD construction results.**

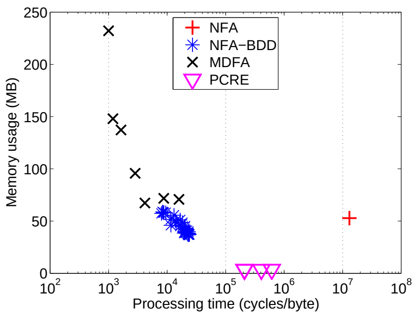
each, while the vector $\boldsymbol{i}$ had 8 Boolean variables to denote the 256 possible ASCII characters. For the NFA-OBDD corresponding to the FTP signature set, the vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ had 15 Boolean variables each. We also tried to determinize these NFAs to produce DFAs, but the determinizer ran out of memory in all three cases.

**Performance of NFA-OBDDs.** Figure 5 depicts the performance of NFA-OBDDs. Figures 5(a) and 5(b) show the performance for each of the 33 HTTP traces, while Figure 5(c) shows the performance for both FTP traces. Figure 5(d) also presents the raw throughput and memory consumption of NFA-OBDDs observed for each signature set. The throughput and memory consumption of NFA-OBDDs varies across different traces for each signature set; this variance can be attributed to the size and shapes of $\mathsf{OBDD}(\mathcal{F})$ (the OBDD of the NFA's frontier) observed during execution. We also observed that larger traces are processed *more* efficiently on average than smaller traces. For example, in Figure 5(a), the 1.24GB trace was processed at 7,935 cycles/byte, whereas a 20MB trace was processed at 19,289 cycles/byte. We hypothesize that the improved throughput observed for larger traces is because of cache effects. As exec_nfaobdd executes, it is likely that NFA-OBDDs that are frequently observed will be cached, therefore producing improved throughput for larger traces.

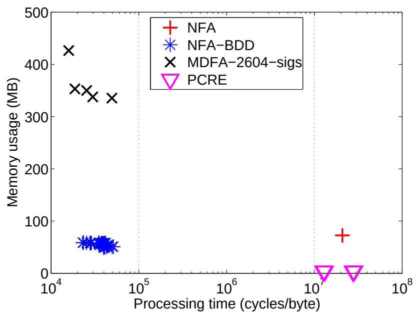
**Comparison with NFAs.** We compared the performance of NFA-OBDDs with an implementation of NFAs that uses Thompson's algorithm. This algorithm maintains a frontier $F$, and operates as follows: for each state $s$ in the frontier $F$, fetch the set of targets $T_s$ of the transitions labeled $\sigma$ and compute the new frontier $F'$=$\bigcup_{s \in F} T_s$.

Our implementation of NFAs makes heavy use of the C++ standard template library. It stores the transition table as an array of $|Q|$ multimaps. The entry for state $s$ denotes the set of outgoing transitions from $s$, where each transition is of the form $(\sigma, t)$. There may be multiple entries with the same input symbol $\sigma$ in each multimap, corresponding to all the states reachable from $s$ via transitions labeled $\sigma$. The performance and memory consumption of our NFA implementation was relatively stable across the traces used for each signature set. Figure 5 therefore reports only the averages across these traces.

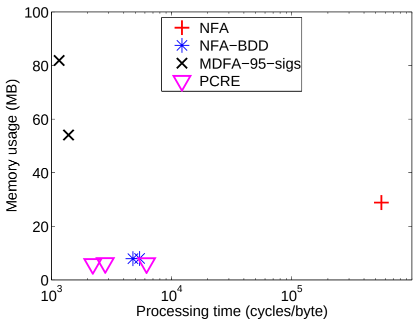
As Figure 5 shows, NFA-OBDDs outperform NFAs for all three sets of signatures by approximately three orders of magnitude for the HTTP signatures, and two orders of magnitude for the FTP signatures. In Figure 5(a), for example, NFA-OBDDs are between $570\times$–$1645\times$ faster than NFAs, while consuming approximately the same amount of memory. The difference in the performance gap between NFA-OBDDs and NFAs for the HTTP and FTP signatures can be attributed to the number and structure of these signatures. As discussed in Section 3, the benefits of NFA-OBDDs are more pronounced if larger frontiers are to be processed. Since there are a larger number of HTTP signatures, the frontier for the corresponding NFAs are larger. As a result, NFA-OBDDs are much faster than the corresponding NFAs for HTTP signatures than for



**(a) HTTP/1503 regular expressions**



**(b) HTTP/2612 regular expressions**



**(c) FTP/98 regular expressions**

| Sig. Set | Processing time | Memory |
|---|---|---|
| **NFA-OBDDs** | | |
| HTTP/1503 | 7,935–22,895 cycles/byte | 39–59MB |
| HTTP/2612 | 22,968–51,215 cycles/byte | 54–61MB |
| FTP/98 | 5,095 cycles/byte | 8MB |
| **NFAs** | | |
| HTTP/1503 | $1.3 \times 10^7$ cycles/byte | 53MB |
| HTTP/2612 | $2.1 \times 10^7$ cycles/byte | 73MB |
| FTP/98 | $5.6 \times 10^5$ cycles/byte | 29MB |
| **PCRE** | | |
| HTTP/1503 | $2.1–6.2 \times 10^5$ cycles/byte | 3.6MB |
| HTTP/2612 | $1.3–2.8 \times 10^7$ cycles/byte | 3.9MB |
| FTP/98 | 2,210–6,185 cycles/byte | 5.9–6.2MB |
| **MDFA (partial signature sets in Figure 5(b) and (c))** | | |
| HTTP/1503 | 1,000–15,951 cycles/byte | 71–232MB |
| HTTP/2604 | 15,891–49,296 cycles/byte | 335–426MB |
| FTP/95 | 1,160–1,386 cycles/byte | 54–82MB |

**(d) Raw performance numbers**

**Fig. 5. Comparing memory versus processing time of (1) NFA-OBDDs, (2) traditional NFAs, (3) the PCRE package, and (4) different MDFAs for the Snort HTTP and FTP signature sets. The x-axis is in log-scale. Note that Figure 5(b) and Figure 5(c) only report the performance of MDFAs with 2604 and 95 regular expressions, respectively.**

FTP signatures. Nevertheless, *these results clearly demonstrate that OBDDs can improve the time-efficiency of NFAs without compromising their space-efficiency.*

**Comparison with the PCRE package.** We compared the performance of NFA-OBDDs with the PCRE package used by a number of tools, including Snort and Perl. The PCRE package represents regular expressions using a tree structure, and matches input strings against this structure using a backtracking algorithm. For a given input string, this algorithm iteratively explores paths in the tree until it finds an accepting state. If it fails to find an accepting state in one path, it backtracks and tries another path until all paths have been exhausted.

Figure 5 reports three numbers for the performance of the PCRE package, corresponding to different values of configuration parameters of the package. In both Figure 5(a) and (b), NFA-OBDDs outperform the PCRE package. The throughput of NFA-OBDDs is about an order of magnitude (9×–26×) better than the fastest configuration of the PCRE package for the set HTTP/1503. The difference in performance is more pronounced for the set HTTP/2612, where NFA-OBDDs outperform the most time-efficient PCRE configuration by $248\times$–$554\times$. The poorer throughput of the PCRE package for the second set of signatures is likely because the backtracking algorithm that it employs degrades in performance as the number of paths to be explored in the NFA increases. However, in both cases, the PCRE package is more space-efficient than NFA-OBDDs, and consumes between $3.7$MB–4MB memory.

For the FTP signatures (Figure 5(c)), NFA-OBDDs are about $2\times$ slower than the fastest PCRE configuration. However, unlike NFA-OBDDs which report all substrings of an input packet that match signatures, this PCRE configuration only reports the first matching substring. The performance of the PCRE configurations that report all matching substrings is comparable to that of NFA-OBDDs.

Note that in all cases, the PCRE package outperforms our NFA implementation, which use Thompson's algorithm [13] to parse input strings. Despite this gap in performance, Cox [17] shows that Thompson's algorithm performs more consistently than the backtracking approach employed by PCRE. For example, the backtracking approach is vulnerable to algorithmic complexity attacks, where a maliciously-crafted input can trigger the worst-case performance of the algorithm [18].

**Comparison with DFA variants.** We compared the performance of NFA-OBDDs with a variant of DFAs, called multiple DFAs (MDFAs), produced by set-splitting [4].[3] An MDFA is a collection of DFAs representing a set of regular expressions. Each DFA represents a disjoint subset of the regular expressions. To match an input string against an MDFA, each constituent DFA is simulated against the input string to determine whether there is a match. MDFAs are more compact than DFAs because they result in a less than multiplicative increase in the number of states. However, MDFAs are also slower than DFAs because all the constituent DFAs must be matched against the input string. An MDFA that has a larger number of constituent DFAs will be more compact, but will also have lower time-efficiency than an MDFA with fewer DFAs.

Using Yu *et al.*'s algorithms [4], we produced several MDFAs by combining the Snort signatures in several ways, each with different space/time utilization. Each point in Figure 5 denotes the performance of *one* MDFA (again, averaged over all the input traces), which in turn consists of a collection of combined DFAs as described above.

Producing MDFAs for the HTTP/2612 and FTP/98 signature sets was more challenging, primarily because these sets contained several structurally-complex regular expressions that were difficult to determinize efficiently. For example, they contained several signatures with large counters (*i.e.,* sequences of repeating patterns) often used in combination with the alternation (*i.e.,* $re_1|re_2$) operator. Our determinizer frequently ran out of memory when attempting to construct MDFAs for such regular expressions. As an example, consider the following regular expression in HTTP/2612:

```
/.*\x2FCSuserCGI\x2Eexe\x3FLogout\x2B[^\s]{96}/i
```

Our determinizer consumed $1.6$GB of memory *for this regular expression alone*, before aborting. Producing a DFA for such regular expressions may require more sophisticated

[3] We were unable to compare the performance of NFA-OBDDs against DFAs because DFA construction ran out of memory. However, prior work [3] estimates that DFAs may offer throughputs of about 50 cycles/byte.

techniques, such as on-the-fly determinization [19] that are not currently implemented in our prototype. We therefore decided to exclude problematic regular expressions, and constructed MDFAs with the remaining ones (2604 for HTTP/2612 and 95 for FTP/98). Note that the MDFAs for these smaller sets of regular expressions may be *more* time-efficient and much more space-efficient than corresponding MDFAs for the entire set of regular expressions.

Figure 5 shows that in many cases NFA-OBDDs can provide throughputs comparable to those offered by MDFAs while utilizing much less memory. For example, the fastest MDFA in Figure 5(b) (constructed for a subset of $2604$ signatures) offered about $50\%$ more throughput than NFA-OBDDs, but consumed $7\times$ more memory. The remaining MDFAs for this signature set had throughputs comparable to those of NFA-OBDDs, but consumed $270$MB more memory than NFA-OBDDs. The performance gap between NFA-OBDDs and MDFAs was largest for FTP signature set, where the MDFAs (for a subset of $95$ signatures) were about $4\times$ faster than the NFA-OBDD; however, the MDFAs consumed $46$MB-$74$MB more memory.

These results are significant for two reasons. First, conventional wisdom has long held that traditional NFAs operate *much* slower than their deterministic counterparts. This is supported by our experiments, which show that the time-efficiency of NFAs is three to four orders of magnitude slower than that of MDFAs. However, our results show that *OBDDs can drastically improve the performance of NFAs and even make them competitive with MDFAs*, which are a determinstic variant of finite automata. We believe that further enhancements to improve the time-efficiency of NFA-OBDDs can make them operate even faster than MDFAs (*e.g.,* by relaxing the OBDD data structure, and thereby eliminating several graph operations in the APPLY and RESTRICT operations).

Second, processing the set of regular expressions to produce compact yet performant MDFAs is a non-trivial exercise, often requiring time-consuming partitioning heuristics to be applied [4]. Some of the partitioning heuristics described by Yu *et al.* also require modifications to the set of regular expressions, thereby changing their semantics. Our own experience constructing MDFAs for HTTP/2612 and FTP/98 shows that this process is often challenging, especially if the regular expressions contain complex structural patterns. In contrast, NFA-OBDDs can be constructed automatically in a straightforward manner from regular expressions, including those with counters and other complex structural patterns, yet are competitive in performance and more compact than MDFAs.

Finally, we also attempted to compare the performance of NFA-OBDDs with a variant of DFAs, called hybrid finite automata (HFA) [20]. HFAs are constructed by interrupting the determinization algorithm when it encounters structurally-complex patterns (*e.g.,* large counters and `.*` patterns) that are known to cause memory blowups when determinized. We used Becchi and Crowley's implementation [20] in our experiments, but found that it ran out of memory when trying to construct HFAs from our signature sets. For example, the HFA construction process exhausted the available memory on our machine after processing just 106 regular expressions in the HTTP/1503 set.

**Deconstructing NFA-OBDD Performance.** We further analyzed the performance of NFA-OBDDs to understand the time consumption of each OBDD operation. The

| Operation | Fraction |
|---|---|
| ANDABSTRACT | 50% |
| AND | 39% |
| MAP | 4% |
| Acceptance check | 7% |

**Fig. 6. Fraction of time spent performing OBDD operations.**

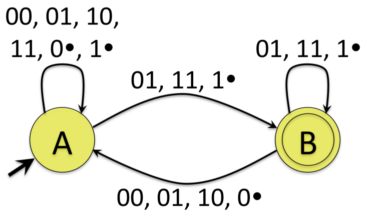


**Fig. 7. 2-stride NFA for Figure 2.**

results reported in this section are based upon the first set of 1503 signatures; the results with the other signature sets were similar.

Figure 6 shows the fraction of time that exec_nfaobdd spends performing various OBDD operations as it processes a single input symbol. As discussed earlier, exec_nfaobdd uses the Cudd package to manipulate OBDDs. Although Cudd implements the OBDD operations described in Section 2, it also implements composite operations that combine multiple Boolean operations; the composite operations are often more efficient than performing the individual operations separately. ANDABSTRACT is one such operation, which allows two OBDDs to be combined using an AND operation followed by an existential quantification. ANDABSTRACT takes a list of Boolean variables to be quantified, and performs the OBDD transformations needed to eliminate all these variables. The MAP operation allows variables in an OBDD to be renamed, *e.g.,* it can be used to rename the $\boldsymbol{y}$ variables in $\mathcal{G}(\boldsymbol{y})$ to $\boldsymbol{x}$ variables instead.

We implemented the Boolean operations required to obtain a new frontier (described in Section 3) using one set of AND, ANDABSTRACT and MAP operations. Each ANDABSTRACT step existentially quantifies 26 Boolean variables (the $\boldsymbol{x}$ and $\boldsymbol{i}$ variables). To check whether a frontier should be accepted, we used another AND operation to combine OBDD($\mathcal{F}$) and OBDD($\mathcal{A}$); the cost of an acceptance check appears in the last row of Figure 6.

Figure 6 shows that the cost of processing an input symbol is dominated by the cost of the ANDABSTRACT and AND operations to compute a new frontier. This is because the sizes of the OBDDs to be combined for frontier computation are bigger than the OBDDs that must be combined to check acceptance. Moreover, computing new frontiers involves several applications of APPLY and RESTRICT, as opposed to an acceptance check, which requires only one APPLY, thereby causing frontier computation to dominate the cost of processing an input symbol.

These results suggest that an OBDD implementation that optimizes the ANDABSTRACT and AND operations (or a relaxed variant of OBDDs that allows more efficient ANDABSTRACT and AND) can further improve the performance of NFA-OBDDs.

## 5 Matching Multiple Input Symbols

The preceding sections assumed that only one input alphabet is processed in each step. However, there is growing interest to develop techniques for *multi-byte matching*, *i.e.,* matching multiple input symbols in one step. Prior work has shown that multi-byte matching can improve the throughput of NFAs [21,22]. We present one such technique,

*$k$-stride NFAs* [21], and show that OBDDs can further improve the performance of $k$-stride NFAs.

A $k$-stride NFA matches $k$ symbols of the input in a single step. Given a traditional (*i.e.,* 1-stride) $\epsilon$-free NFA ($Q$, $\Sigma$, $\delta$, $q_0$, $F$), a $k$-stride NFA is a 5-tuple ($Q$, $\Sigma^k$, $\Gamma$, $q_0$, $F$), whose input symbols are $k$-grams, *i.e.,* elements of $\Sigma^k$. The set of states and accepting states of the $k$-stride NFA are the same as those for the 1-stride NFA. Intuitively, the transition relation $\Gamma$ of the $k$-stride NFA is computed as a $k$-step closure of $\delta$, *i.e.,* ($s$, $\sigma_1\sigma_2\ldots\sigma_k$, $t$) $\in \Gamma$ if and only if the state $t$ is reachable from state $s$ in the original NFA via transitions labeled $\sigma_1$, $\sigma_2$, $\ldots$, $\sigma_k$. The algorithm to compute $\Gamma$ from $\delta$ must also consider cases where the length of the input string is not a multiple of $k$. Intuitively, this is achieved by padding the input string with a new "do-not-care" symbol, and introducing this symbol in the labels of selected transitions. We refer the interested reader to prior work [21,22] for a detailed description of the construction.

Figure 7 presents an example of a 2-stride NFA corresponding to the NFA in Figure 2. The do-not-care symbol is denoted by a "$\bullet$". Thus, for instance, an input string 101 would be padded with $\bullet$ to become $101\bullet$. The 2-stride NFA processes digrams in each step. Thus, the first step would result in a transition from state $A$ to itself $A$ (because of the transition labeled 10), followed by a transition from $A$ to $B$ when it reads the second digram $1\bullet$, thereby accepting the input string.

A $k$-stride NFA ($Q$, $\Sigma^k$, $\Gamma$, $q_0$, $F$) can readily be converted into a $k$-stride NFA-OBDD using the same approach described in Section 3. The main difference is that the input alphabet is $\Sigma^k$ (plus "$\bullet$"). Transition tables of $k$-stride NFAs encountered in practice are generally sparse. We therefore applied a well-known technique called *alphabet compression* [21], which reduces the size of the input alphabet by combining symbols in the input alphabet into equivalence classes. An alphabet-compressed NFA can also be converted into an NFA-OBDD using the same techniques described in Section 3, and operated in the same way.

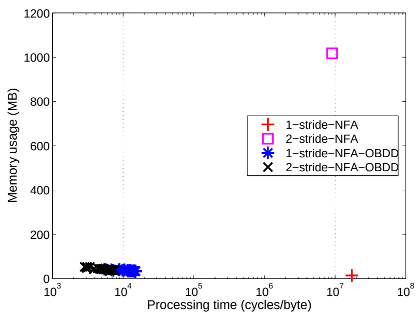
**Performance of $k$-stride NFA-OBDDs.** To evaluate the performance of $k$-stride NFAs and $k$-stride NFA-OBDDs, we used a toolchain similar to the one discussed in Section 4, but additionally applied alphabet compression. Our implementation accepts $k$ as an input parameter. However, we have only conducted experiments for $k = 2$ because alphabet compression ran out of memory for larger values of $k$.

The setup that we used for the experiments reported below is identical to that described in Section 4. However, we only used two sets of Snort signatures in our measurements: (1) HTTP/1400: a subset of 1400 HTTP signatures from HTTP/1503 and (2) FTP/95 a subset of 95 FTP signatures from FTP/98. This was because the 2-stride NFA for a larger number of signatures ran out of memory during execution, thereby precluding a head-to-head comparison between the performance of 2-stride NFAs and NFA-OBDDs. We did not consider HTTP/2612 for the experiments reported in this section, because alphabet compression ran out of memory on these signature sets.

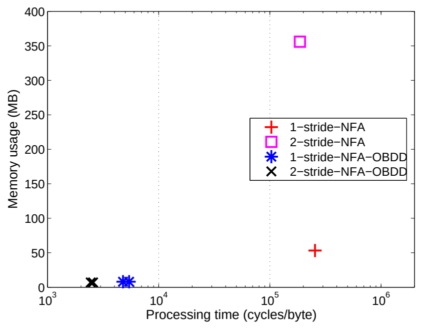
Figure 8(a) presents the size of the 1-stride and 2-stride NFA-OBDDs, and the size of the compressed alphabet. In each case, the alphabet compression algorithm took over a day to complete, and consumed over 1GB of memory. Figure 8(b) and (c) compare the performance of 1-stride NFAs and NFA-OBDDs with the performance of 2-stride NFAs and NFA-OBDDs. As in Section 4, for NFAs we only report the average perfor-

| Signature Set | #States | #Transitions (1-stride) | #Transitions (2-stride) | #Alpbahet Symbols |
|---|---|---|---|---|
| HTTP/1400 | 146,992 | 2,246,701 | 44,815,280 | 6,928 |
| FTP/95 | 15,266 | 3,361,065 | 5,136,420 | 848 |

**(a) 1-stride and 2-stride NFA-OBDD construction results.**



**(b) HTTP/1400 regular expressions.**



**(c) FTP/95 regular expressions.**

**Fig. 8. Memory versus throughput for 1-stride NFAs, 1-stride NFA-OBDDs, 2-stride NFAs, and 2-stride NFA-OBDDs.**

mance across all network traces because their performance was relatively stable across all traces. We first note from Figure 8 that as expected, matching multiple bytes in the input stream improved the performance of NFAs. However, this increase in throughput comes at a drastic increase in the memory consumption of the 2-stride NFA.

In both the 1-stride and 2-stride NFAs, the use of OBDDs improved throughput—by about three orders of magnitude for HTTP/1400 and about two orders of magnitude for FTP/95. In both cases, the memory utilization of the 2-stride NFA-OBDD was smaller than that of the 2-stride NFA by two orders of magnitude. This is because OBDDs compactly encode the NFA's transition relation. These results show that *2-stride NFA-OBDDs are drastically more efficient in time and space than 2-stride NFAs.* Further investigation of the benefits of $k$-stride NFAs is a topic for future work.

## 6 Related Work

Early NIDS exclusively employed strings as attack signatures. String-based signatures are space-efficient, because their size grows linearly with the number of signatures. They are also time-efficient, and have $O(1)$ matching algorithms (*e.g.,* Aho-Corasick [23]). They are ideally suited for wire-speed intrusion detection, and have been implemented both in software and hardware [24,25,26,27,28,29]. However, prior work has shown that string-based signatures can easily be evaded by malware using polymorphism, metamorphism and other mutations [30,31,32,33]. The research community has therefore been investigating sophisticated signature schemes that require the full power of regular expressions. This in turn, has spurred both the research community to develop improved algorithms for regular expression matching, as well as NIDS vendors, who are increasingly beginning to deploy products that use regular expressions [34,35,36].

DFAs provide high-speed matching, but DFAs for large signature sets often consume gigabytes of memory. Researchers have therefore investigated techniques to improve the space-efficiency of DFAs. These include, for example, techniques to determinize on-the-fly [19]; MDFAs, which combine signatures into multiple DFAs (as discussed in Section 4) [4]; $D^2$FAs [2], which reduce the memory footprint of DFAs via edge compression; and XFAs [3,8], which extend DFAs with scratch memory to store auxiliary variables, such as bitmaps and counters, and associate transitions with instructions to manipulate these variables. Some DFA variants (*e.g.,* [2,3,21]) also admit efficient hardware implementations.

These techniques use the time-efficiency of DFAs as a starting point, and seek to reduce their memory footprint. In contrast, our work uses the space-efficiency of NFAs as a starting point, and seeks to improve their time-efficiency. We believe that both approaches are orthogonal and may be synergistic. For example, it may be possible to use OBDDs to also improve the time-efficiency of MDFAs.

Our approach also provides advantages over several prior DFA-based techniques. First, it produces NFA-OBDDs from regular expressions in a fully automated way. This is in contrast to XFAs [8], which required a manual step of annotating regular expressions. Second, our approach does not modify the semantics of regular expressions, *i.e.,* the NFA-OBDDs produced using the approach described in Section 3 accept the same set of strings as the regular expressions that they were constructed from. MDFAs, in contrast, employ heuristics that relax the semantics of regular expressions to improve the space-efficiency of the resulting automata [4]. Last, because these techniques operate with DFAs, they may sometimes encounter regular expressions that are hard to determinize. For example, Smith *et al.* [8, Section 6.2] present a regular expression from the Snort data set for which the XFA construction algorithm runs out of memory. Our technique operates with NFAs and therefore does not encounter such cases.

Research on NFAs for intrusion detection has typically focused on exploiting parallelism to improve performance [9,10,37,38]. NFA operation can be parallelized in many ways. For example, a separate thread could be used to simulate each state in an NFA's frontier. Else, a set of regular expressions can be represented as a collection of NFAs, which can then be operated in parallel. FPGAs have been used to exploit this parallelism to yield high-performance NFA-based intrusion detection systems [9,10,37,38].

Although not explored in this paper, OBDDs can potentially improve NFA performance in parallel execution environments as well. For example, consider a NIDS that performs signature matching by operating a collection of NFAs in parallel. The performance of this NIDS can potentially be improved by converting it to use a collection of NFA-OBDDs instead; in this case, OBDDs improve the performance of each NFA, thereby increasing the throughput of the NIDS as a whole. Finally, NFA-OBDDs may also admit a hardware implementation. Prior work has developed techniques to implement OBDDs in CAMs [39] and FPGAs [40]. Such an implementation of NFA-OBDDs can potentially be used to improve the performance of hardware-based NFAs as well.

## 7 Summary

Many recent algorithms for regular expression matching have focused on improving the space-efficiency of DFAs. This paper sought to take an alternative viewpoint, and aimed to improve the time-efficiency of NFAs. To that end, we developed NFA-OBDDs, a representation of regular expressions in which OBDDs are used to operate NFAs.

Our prototype software-based implementation with Snort signatures showed that NFA-OBDDs can drastically outperform NFAs—by up to 1645$\times$ in the best case. We also showed how OBDDs can enhance the performance of NFAs that match multiple input symbols.

**Acknowledgements.** We thank Cristian Estan and Somesh Jha for useful discussions in the early stages of this project. We also thank Michael Bailey for shepherding the paper and the anonymous reviewers for their feedback on our work.

## References

1. Becchi, M.: Data Structures, Algorithms and Architectures for Efficient Regular Expression Evaluation. PhD thesis, Washington University in St. Louis (2009)
2. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: ACM SIGCOMM Conference, ACM (2006) 339–350
3. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the Big Bang: Fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM Conference, ACM (2008) 207–218
4. Yu, F., Chen, Z., Diao, Y., Lakshman, T.V., Katz, R.H.: Fast and memory-efficient regular expression matching for deep packet inspection. In: ACM/IEEE Symp. on Arch. for Networking and Comm. Systems. (2006) 93–102
5. Roesch, M.: Snort - lightweight intrusion detection for networks. In: USENIX Conf. on System Administration, USENIX (1999) 229–238
6. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, Third Edition. Addison-Wesley (2007)
7. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: Intl. Conf. on Architectures for Networking and Communication Systems, ACM (2007) 145–154
8. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: Symp. on Security and Privacy, IEEE Computer Society (2008) 187–201
9. Sidhu, R., Prasanna, V.: Fast regular expression matching using FPGAs. In: Symp. on Field-Programmable Custom Computing Machines, IEEE Computer Society (2001) 227–238
10. Clark, C.R., Schimmel, D.E.: Scalable pattern matching for high-speed networks. In: IEEE Symp. on Field-Programmable Custom Computing Machines, IEEE Computer Society (2004) 249–257
11. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers **35**(8) (1986) 677–691
12. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, J.: Symbolic model checking: $10^{20}$ states and beyond. In: Symp. on Logic in Computer Science, IEEE Computer Society (1990) 401–424
13. Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6) (1968) 419–422
14. Somenzi, F.: 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/∼fabio/CUDD`.
15. Signatures referenced in Section 4 and Section 5: Available at `http://www.cs.rutgers.edu/~vinodg/papers/raid2010`.
16. PCRE: The Perl compatible regular expression library `http://www.pcre.org`.

17. Cox, R.: 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`.
18. Smith, R., Estan, C., Jha, S.: Backtracking algorithmic complexity attacks against a NIDS. In: Annual Computer Security Applications Conf., IEEE Computer Society (2006) 89–98
19. Sommer, R., Paxson, V.: Enhancing byte-level network intrusion detection signatures with context. In: Conf. on Computer and Comm. Security, ACM (2003) 262–271
20. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: Intl. Conf. on emerging Networking EXperiments and Technologies. (2007)
21. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. In: Intl. Symp. Computer Architecture, IEEE Computer Society (2006) 191–202
22. Becchi, M., Crowley, P.: Efficient regular expression evaluation: Theory to practice. In: Intl. Conf. on Architectures for Networking and Communication Systems, ACM (2008) 50–59
23. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Comm. ACM **18**(6) (1975) 333–340
24. Dharmapurikar, S., Lockwood, J.W.: Fast and scalable pattern matching for network intrusion detection systems. Jour. on Selected Areas in Comm. **24**(10) (2006) 1781–1792
25. Liu, R., Huang, N., Chen, C., Kao, C.: A fast string-matching algorithm for network processor-based intrusion detection system. Trans. on Embedded Computing Sys. **3**(3) (2004) 614–633
26. Sourdis, I., Pnevmatikatos, D.: Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In: Cheung, P., Constantinides, G., Sousa, J., (eds.) FPL 2003. Volume 2778 of LNCS., Springer (2003) 880–889
27. Tan, L., Sherwood, T.: A high throughput string matching architecture for intrusion detection and prevention. In: Intl. Symp. Computer Architecture, IEEE Computer Society (2005) 112–122
28. Tuck, N., Sherwood, T., Calder, B., Varghese, G.: Deterministic memory-efficient string matching algorithms for intrusion detection. In: IEEE INFOCOM 2004, IEEE Computer Society 333–340
29. Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E.P., Ioannidis, S.: Gnort: High performance network intrusion detection using graphics processors. In: Lippman, R., Kirda, E., Trachtenberg, A., (eds.) RAID 2008. Volume 5230 of LNCS., Springer (2008) 116–134
30. Handley, M., Paxson, V., Kreibich, C.: Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In: Usenix Security, USENIX (2001) 9–9
31. Jordan, M.: Dealing with metamorphism. Virus Bulletin Weekly (2002)
32. Ptacek, T., Newsham, T.: Insertion, evasion and denial of service: Eluding network intrusion detection `http://insecure.org/stf/secnet_ids/secnet_ids.html`.
33. Shankar, U., Paxson, V.: Active mapping: Resisting NIDS evasion without altering traffic. In: Symp. on Security and Privacy, IEEE Computer Society (2003) 44–61
34. TippingPoint: `http://www.tippingpoint.com`.
35. LSI-Corporation: Tarari RegEx content processor `http://www.tarari.com`.
36. Cisco: IOS terminal services configuration guide `http://tinyurl.com/2eouvq`.
37. Hutchings, B.L., Franklin, R., Carver, D.: Assisting network intrusion detection with reconfigurable hardware. In: Annual Symp. on Field-Programmable Custom Computing Machines, IEEE Computer Society (2002) 111–120
38. Mitra, A., Najjar, W., Bhuyan, L.: Compiling PCRE to FPGA for accelerating Snort IDS. In: Symp. on Arch. for Networking and Comm. Systems, ACM (2007) 127–136
39. Yusuf, S., Luk, W.: Bitwise optimized CAM for network intrusion detection systems. In: Intl. Conf. on Field Prog. Logic and Applications, IEEE Press (2005) 444–449
40. Sinnappan, R., Hazelhurst, S.: A reconfigurable approach to packet filtering. In: Brebner, G., and Woods, R., (eds.) FPL 2001. Volume 2147 of LNCS., Springer (2001) 638–642