Computer Networks 55 (2011) 3376–3393

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Fast, memory-efficient regular expression matching with NFA-OBDDs ☆

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

[a] Department of Computer Science, Rutgers University, 110 Frelinghuysen Road, Piscataway, NJ 08854-8019, USA
[b] Sandia National Laboratories, Albuquerque, NM 87185-1027, USA

ARTICLE INFO

Article history:
Received 9 December 2010
Received in revised form 26 April 2011
Accepted 2 July 2011
Available online 18 July 2011

Keywords:
Network intrusion detection
Finite automata
Regular expressions
Ordered binary decision diagrams

ABSTRACT

Modern network intrusion detection systems (NIDS) employ regular expressions as attack signatures. Internally, NIDS represent and operate these regular expressions as finite automata. However, finite automata present a well-known time/space tradeoff. Deterministic automata (DFAs) provide fast matching, but DFAs for large signature sets often consume gigabytes of memory because DFA combination results in a multiplicative increase in the number of states. Non-deterministic automata (NFAs) address this problem and 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.

We consider an alternative approach that focuses instead on improving the time-efficiency of NFA-based signature matching. NFAs are inefficient because they maintain a frontier of multiple states at any instant during their operation, each of which must be processed for every input symbol. We introduce NFA-OBDDs, which use ordered binary decision diagrams (OBDDs) to efficiently process *sets* of NFA frontier states. Experiments using HTTP and FTP signature sets from Snort show that NFA-OBDDs can outperform traditional NFAs by up to three orders of magnitude, thereby making them competitive with a variant of DFAs, while still retaining the space-efficiency of NFAs.

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 led the research community to investigate richer signature representations (*e.g.,* [47,54,57]), many of 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 led to the investigation of new representations of regular expressions that allow for efficient inspection of network traffic (*e.g.,* [3,5,15,28,44,58]).

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 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

☆ This article is a revised and expanded version of our paper that appears in the 13th International Symposium on Recent Advances in Intrusion Detection (RAID'10) [55].

* Corresponding author. Tel.: +1 732 445 2001x3003.
** Principle corresponding author. Fax: +1 732 445 0537.
E-mail addresses: lyangru@cs.rutgers.edu (L. Yang), rkarim@cs.rutgers.edu (R. Karim), vinodg@cs.rutgers.edu (V. Ganapathy), ransmit@sandia.gov (R. Smith).

1389-1286/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2011.07.002

signatures. This requirement is important because the increasing diversity of network attacks has led to a quick growth in the number of signatures used by NIDS. For example, the number of signatures in Snort [36] has grown from 3166 in 2003 to 15,047 in 2009. This upward trend will likely accelerate in the future as NIDS vendors begin to employ automated and semi-automated methods for signature generation [9,26,34,41].

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 [22]. 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. As a result, 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.,* [5,28]), techniques to combine regular expressions into multiple DFAs [58], and variable extended finite automata (XFAs) [44], 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 [22].
- *NFA combination is space-efficient.* Combining two NFAs simply involves linking their start states together by adding new ϵ 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. Smith et al. [45] formally characterize this blowup using the notion of *ambiguity*. However, identifying and eliminating ambiguity currently involves some manual effort.
- *NFAs can readily be parallelized.* An NFA may contain multiple outgoing transitions for a single input symbol from each state, all of which must be followed when that input symbol is encountered. An NFA simulator can easily parallelize these operations as shown in prior work [13,40].

Motivated by these advantages, we develop a new approach to improve the time-efficiency of NFAs. We begin by noting that an NFA can be in a set of states (called the *frontier*) at any instant during its operation. 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* [10] (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 [11].

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 traces of real HTTP and FTP traffic. Our experiments showed that NFAs that use OBDDs (*NFA-OBDDs*) outperform traditional NFAs by approximately *three orders of magnitude*. 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.

In addition to improving the time-efficiency of NFAs, our approach has a number of advantages. First, construction of NFA-OBDDs from regular expressions is fully automated and does not change signature semantics. In contrast, prior work on improved signature representations has required manual analysis of regular expressions (*e.g.,* to identify and eliminate ambiguity [45]) or requires the semantics of signatures to be modified (*e.g.,* [58]). Second, it uniformly handles all regular expressions. Prior techniques, especially those that convert regular expressions into DFAs (or variants), often require manual intervention when regular expressions have certain kinds of constructs (*e.g.,* counters; see [44,58]). Last, NFA-OBDDs may be amenable to a hardware implementation. Both

NFAs (*e.g.,* [13,17,40]) and OBDDs [42,60] have individually been implemented in hardware. It may be possible to combine ideas from prior work to construct NFA-OBDDs in hardware.

Our main contributions in this work 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 6).
- *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 5).

The main benefit of NFA-OBDDs is in improving the *performance* (*i.e.,* time and space-efficiency) of deep packet inspection by NIDS, independent of its *effectiveness* at detecting attacks. We acknowledge that matching network traffic against regular expressions is no longer sufficient to detect a large fraction of attacks, and that additional security mechanisms and advanced forms of signatures (*e.g.,* vulnerability signatures [9,54]) are necessary. Nevertheless, real deployments use layered defenses, and NIDS will remain a cornerstone of network security for the foreseeable future. Advanced signature matching techniques also employ regular expression matching (*e.g.,* see [38]) and we expect that NFA-OBDDs will benefit them as well.

The rest of this article is organized as follows: Section 2 presents background material on OBDDs; Section 3 describes the construction and operation of NFA-OBDDs; Section 4 describes the experimental setup and data sets used in our evaluation, while Section 5 compares the performance of NFA-OBDDs against other techniques to match regular expressions. Section 6 extends NFA-OBDDs to multi-stride automata and presents experimental evaluation of multi-stride NFA-OBDDs. We discuss related work in Section 7 and conclude the article in Section 8.

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 extensively by model checkers to improve the efficiency of state-space exploration algorithms [11]. OBDDs and their variants have also been used in the analysis and design of intrusion detection systems and firewalls [12,18,19,21,59,60].

Formally, an OBDD represents a Boolean function $f(x_1,x_2,\ldots,x_n)$ as a rooted, directed acyclic graph (DAG) that has two kinds of nodes: *non-terminals* and up to two *terminals*, which are labeled **0** and **1**. Terminal nodes do not have outgoing edges. Each non-terminal node v is associated with a label $\text{VAR}(v) \in \{x_1,x_2,\ldots,x_n\}$, and has two successors $\text{LOW}(v)$ and $\text{HIGH}(v)$. The edges to these successors are labeled **0** and **1**, respectively. 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. An OBDD must also satisfy two additional properties:

- there are no two non-terminal nodes u and v such that $\text{VAR}(u) = \text{VAR}(v)$, $\text{LOW}(u) = \text{LOW}(v)$, and $\text{HIGH}(u) = \text{HIGH}(v)$; and
- there is no non-terminal u with $\text{LOW}(u) = \text{HIGH}(u)$.

In his seminal article, Bryant [10] introduced algorithms to construct OBDDs for Boolean formulae and showed that for a given total order of the variables of a Boolean formula, the OBDD representation of that formula is canonical, *i.e.,* for a given ordering, two OBDDs for a Boolean formula are isomorphic. Fig. 1(b) depicts an example of an OBDD for the Boolean formula $f(x,i,y)$ shown in Fig. 1(a). In this figure, the variable ordering is $x < i < y$. To evaluate the Boolean formula for a given variable assignment, say $\{x \leftarrow 1, i \leftarrow 0, y \leftarrow 1\}$, it suffices to traverse the appropriately labeled edges from the root to the terminal nodes; in this case $f(1,0,1)$ evaluates to 0. Fig. 1(c) depicts the OBDD for f with the variable ordering $i < x < y$. Although not evident from this example, the size of OBDDs is sensitive to the total order imposed on the Boolean variables; it is NP-hard to choose a total order that yields the most compact OBDD for a Boolean function [10].

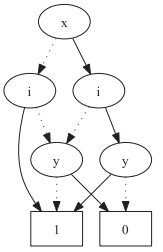
An OBDD representation of a Boolean formula offers several advantages. First, OBDDs are often more compact than other representations of Boolean formulae, such as decision trees, conjunctive normal form (CNF) and disjunctive normal form (DNF). Intuitively, this is because an OBDD captures and eliminates redundant nodes in the decision tree representation of a Boolean formula. Second, OBDDs allow properties of Boolean functions to be checked efficiently. For example, to determine whether a Boolean function is satisfiable (or unsatisfiable), it suffices to check whether the terminal node labeled **1** (respectively, **0**) is reachable from the root node. Because OBDD construction and manipulation algorithms eliminate nodes that are unreachable from the root, checking (un)satisfiability is a constant-time operation.

OBDDs allow Boolean functions to be manipulated efficiently. Bryant [10] describes two operations, APPLY and RESTRICT, which 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. We describe APPLY and RESTRICT informally below, and refer the reader to Bryant's article for details of these algorithms.

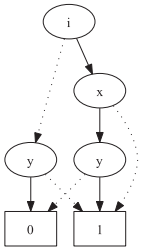
APPLY allows binary Boolean operators, such as $\wedge$ and $\vee$, to be applied to a pair of OBDDs. The two input OBDDs, $\text{OBDD}(f)$ and $\text{OBDD}(g)$, must have the same variable ordering. $\text{APPLY}(\langle\text{OP}\rangle, \text{OBDD}(f), \text{OBDD}(g))$ computes $\text{OBDD}(f \ \langle\text{OP}\rangle \ g)$, which has the same variable ordering as the input OBDDs. Fig. 2(a) presents the OBDD obtained by combining the OBDD in Fig. 1(b) with $\text{OBDD}(I(i))$ (Fig. 1(d)), where I is

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

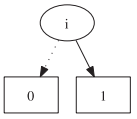
(a) A Boolean function $f(x, i, y)$.



(b) OBDD(f) with $x < i < y$.



(c) OBDD(f) with $i < x < y$.



(d) OBDD($I(i)$), the identity function.

Fig. 1. An example of a Boolean formula and OBDDs with different variable orderings. Solid edges are labeled 1, dotted edges are labeled 0.

the identity function. Intuitively, APPLY is implemented as a simple recursive algorithm that processes the DAG representing the OBDD in layers, with each recursive step processing a subgraph of the previous step, and finally reducing the resulting DAG so that it satisfies the properties of an OBDD (*e.g.,* deleting unreachable nodes, and suitably merging nodes).

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 any nodes with the label x. Fig. 2(b) shows the OBDD obtained as the output of RESTRICT(OBDD(f), $i \leftarrow 1$), where OBDD(f) is the OBDD of Fig. 1(b). Intuitively, the RESTRICT operation is implemented by eliminating the nodes labeled i, suitably redirecting edges from i's predecessors to point to i's successors and removing unreachable nodes.

Finally, 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:

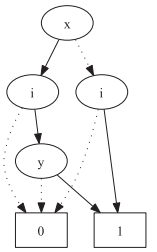
$$\text{OBDD}(\exists x_i . f(x_1, \ldots, x_n)) = \text{Apply}(\vee, \text{Restrict}(\text{OBDD}(f), x_i \leftarrow 1), \text{Restrict}(\text{OBDD}(f), x_i \leftarrow 0)).$$

Note that OBDD($\exists x_i \cdot f(x_1, \ldots, x_n)$) will not have a node labeled x_i.

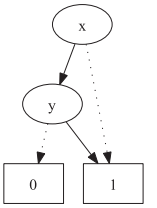
2.1. Representing relations and sets

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 ϕ). 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) APPLY($\wedge$, OBDD(f), OBDD($I(i)$)).



(b) RESTRICT(OBDD(f), $i \leftarrow 1$).

Fig. 2. Result of the APPLY and RESTRICT operations on the OBDD in Fig. 1(b).

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 OBDD(S) $\rightarrow$ OBDD(T) (logical implication), while ISEMPTY($S \cap T$) is equivalent to checking whether OBDD(S) $\wedge$ 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 and NFA-OBDDs

We represent an NFA using a 5-tuple: $(Q, \Sigma, \Delta, q_0, Fin)$, where Q is a finite set of states, Σ is a finite set of input symbols (the alphabet), Δ: $Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ 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 Δ can also be expressed as a relation δ: $Q \times \Sigma \times Q$, so that $(s,i,t) \in \delta$ for all $t \in T$ such that $\Delta(s,i) = T$. We will henceforth use δ 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 be in at each step during execution. 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. For example, an NFA for `(0|1)*1(0|1)`^{n} has $O(n)$ states, while the corresponding DFA has $O(2^n)$ states [22]. Second, and perhaps more significantly from the perspective of NIDS, NFAs can be combined space-efficiently while DFAs often cannot. To combine a pair of NFAs, NFA_1 and NFA_2, it suffices to create a new state q_{new}, add ϵ 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, can sometime result 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 $16 \times |\Sigma|$ transitions.

3.1. NFA operation using boolean function manipulation

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 ϵ transitions have been eliminated (using standard techniques [22]). This is mainly for ease of exposition; NFAs with ϵ transitions can also be expressed using NFA-OBDDs. Note that ϵ 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: $\vec{x}$, $\vec{y}$, and $\vec{i}$. The vectors $\vec{x}$ and $\vec{y}$ are used to denote states in Q, and therefore contain $\lceil \lg|Q| \rceil$ variables each. The vector $\vec{i}$ denotes symbols in Σ, and contains $\lceil \lg|\Sigma| \rceil$ variables. As an example, for the NFA in Fig. 3, these vectors contain one Boolean variable each; we denote them as x, y, and i.

- $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$ denotes the NFA's transition relation δ. Recall that δ 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 2.1. For example, consider the NFA in Fig. 3. Using 0 to denote state A and 1 to denote state B, $\mathcal{T}(x, i, y)$ is the function shown in Fig. 1(a).
- $\mathcal{I}_\sigma(\vec{i})$ is defined for each $\sigma \in \Sigma$, and denotes a Boolean representation of that symbol. For the NFA in Fig. 3, $\mathcal{I}_0(i) = \bar{i}$ (*i.e.*, $i = 0$) and $\mathcal{I}_1(i) = i$.
- $\mathcal{F}(\vec{x})$ 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 our running example, if $F = \{A\}$, $\mathcal{F}(x) = \bar{x}$, while if $F = \{A, B\}$, then $\mathcal{F}(x) = x \vee \bar{x}$.
- $\mathcal{A}(\vec{x})$ is a Boolean representation of Fin, and denotes the accepting states. In Fig. 3, $\mathcal{A}(x) = x$.

Note that $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$, $\mathcal{I}_\sigma(\vec{i})$ and $\mathcal{A}(\vec{x})$ 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 $\mathcal{F}(\vec{x})$, and that the next symbol in the input is σ. The following Boolean formula, $\mathcal{G}(y)$, symbolically denotes the new frontier of states in the NFA after σ has been processed.

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

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

We illustrate this idea using the example in Fig. 3. 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 Fig. 1(a), $\mathcal{I}_0(i) = \bar{i}$ and $\mathcal{F}(x) = \vee\bar{x}$. 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 $F \cap Fin \neq \emptyset$. Using the Boolean notation, this translates to check whether $\mathcal{F}(\vec{x}) \wedge \mathcal{A}(\vec{x})$ 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

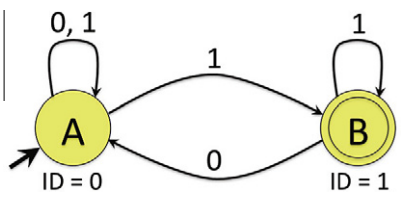


Fig. 3. NFA for `(0|1)*1`.

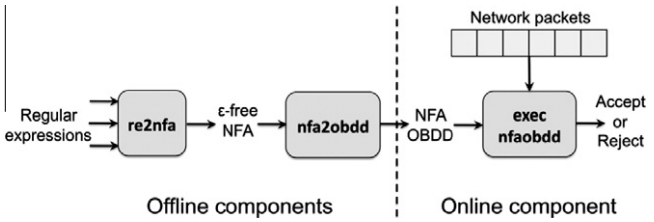


Fig. 4. Components of our software-based implementation of NFA-OBDDs.

of a Boolean function is an $O(1)$ operation if the function is represented as an OBDD.

3.2. 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 $(\vec{x}, \vec{i}, \vec{y}, \text{OBDD}(\mathcal{T}), \{\text{OBDD}(\mathcal{I}_\sigma) | \forall \sigma \in \Sigma)\}, \text{OBDD}(\mathcal{F}_{q_0}), \text{OBDD}(\mathcal{A}))$, where $\vec{x}$, $\vec{i}$, $\vec{y}$ are vectors of Boolean variables, and $\mathcal{T}$, $\mathcal{I}_\sigma$, and $\mathcal{A}$ are the Boolean formulae discussed in Section 3.1. $\mathcal{F}_{q_0}$ denotes the Boolean function that denotes the frontier $\{q_0\}$. For each input symbol σ, 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 approximately $O(\text{sizeof}(\text{OBDD}(\mathcal{T})) \times \text{sizeof}(\text{OBDD}(\mathcal{F})))$.[1] Because OBDDs are a compact representation of the frontier F and the transition relation δ, 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 representations of δ 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 $\text{OBDD}(\mathcal{T})$, 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. Experimental apparatus and data sets

We evaluated the feasibility of our approach using a software-based implementation of NFA-OBDDs. As depicted in Fig. 4, the experimental apparatus 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 ϵ-free NFA as output. To do so, it first constructs NFAs for each of the regular expressions using Thompson's construction [22,51], combines these NFAs into a single NFA, and eliminates ϵ transitions. The nfa2obdd component analyzes this NFA to determine the number of Boolean variables needed (*i.e.,* the sizes of the $\vec{x}, \vec{i}$ and $\vec{y}$ vectors), and constructs $\text{OBDD}(\mathcal{T})$, $\text{OBDD}(\mathcal{A})$, $\text{OBDD}(\mathcal{I}_\sigma)$ for each $\sigma \in \Sigma$, and $\text{OBDD}(\mathcal{F}_{q_0})$.

As discussed in Section 2, the size of an OBDD for a Boolean formula is sensitive to the total order imposed on its variables. Variable ordering also impacts the structure of OBDDs, and therefore the performance of NFA-OBDDs. We empirically determined that an ordering of variables of the form $\vec{i} < \vec{x} < \vec{y}$ yields high-performance NFA-OBDDs. Our implementation of nfa2obdd therefore uses this ordering for $\vec{i}$, $\vec{x}$ and $\vec{y}$. Within each vector, nfa2obdd orders variables in increasing order from most significant bit to least significant bit. Section 5.6 presents a detailed evaluation of the impact of variable ordering on the performance of 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 contents 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 [46]. 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.[2] Note that in a NIDS setting, it is important to check whether the frontier $\mathcal{F}$ obtained after processing *each* input symbol contains an accepting state (rather than after processing the entire input string, which is the traditional operating model for finite automata). 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. This model is

[1] This is because the complexity of obtaining a new frontier is dominated by the cost of an Apply operation on $\text{OBDD}(\mathcal{T})$ and $\text{OBDD}(\mathcal{F})$, which costs $O(\text{sizeof}(\text{OBDD}(\mathcal{T})) \times \text{sizeof}(\text{OBDD}(\mathcal{F})))$ [10].

[2] Multiple regular expressions may trigger a match on an input symbol; these regular expressions can be identified using the set of states that appear in the conjunction $\mathcal{F}(\vec{x}) \wedge \mathcal{A}(\vec{x})$.

equivalent to using regular expressions to find all matching substrings within a string, as the characters in the string are presented to the matching algorithm one at a time.

4.1. Signature sets and network traffic

Signature Sets. We evaluated our implementation of NFA-OBDDs with three sets of regular expressions, described below. All three sets of regular expressions include client-side and server-side signatures.

(1) *HTTP/1503*. The first set was obtained from the authors of the XFA paper [44], and contains 1503 regular expressions that were synthesized from the March 2007 snapshot of the Snort HTTP signature set.
(2) *HTTP/2612*. The second set, numbering 2612 regular expressions, was synthesized from the October 2009 snapshot of the Snort's HTTP signature set as follows. The October 2009 snapshot has 4288 rules for HTTP traffic, of which 1658 have `uricontent` fields (1387 after eliminating duplicates) and 3078 have `pcre` fields (some rules have both fields). We excluded `pcre` fields that contain non-regular constructs, such as back-references and subroutines because these constructs are not regular and therefore cannot be implemented in NFA-based models, which eliminated 1853 `pcre` fields (leaving us with 1225 regular expressions extracted from these fields). This resulted in a total of 2612 unique regular expressions synthesized from the Snort's HTTP signature set. Although extracting just `pcre` and `uricontent` fields from individual Snort rules only captures a portion of the corresponding rules, it suffices for our experiments, because our primary goal is to evaluate the performance of NFA-OBDDs against other regular-expression based techniques.
(3) *FTP/98*. The third set of signatures was synthesized from Snort's signatures for FTP traffic. We used 27 regular expressions extracted from the `pcre` fields of FTP rules from the October 2009 rule set. We combined these regular expressions with 71 regular expressions for FTP traffic, synthesized by the authors of the XFA paper [44], leaving us with a total of 98 regular expressions for FTP traffic.

We have made these regular expression sets available for use by other researchers [56].

HTTP traffic. We evaluated the performance of HTTP signatures by feeding three sets of HTTP traffic traces to `exec_nfaobdd`:

(1) *Rutgers traces.* We recorded HTTP traffic at the Web server of the Rutgers Computer Science Department for a one week period in August 2009. This traffic was collected using tcpdump, and includes whole packets of port 80 traffic from the Web server. 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, this week-long trace contained connections from 18,618 distinct source IP addresses. It contained a total of 10,069,369 network packets, amounting to 1.24 GB worth of data, with the payloads in the network packets ranging in size from 1 byte to 1460 bytes, with an average of 126 bytes (standard deviation of 271). Table 1 presents statistics that characterize various other aspects of the trace.[3]
(2) *DARPA traces.* We used publicly available traces from the 1999 DARPA intrusion detection evaluation data sets [29]. Privacy concerns preclude us from releasing the network traces collected in our department. We therefore report experimental results with the DARPA traces to ensure that our experiments can be repeated independently by other researchers.
We acknowledge that the DARPA traces are no longer in popular use for intrusion detection research. Indeed, researchers have even argued that they are inadequate for the purpose that they were originally developed (to test the effectiveness of intrusion detection systems at detecting attacks; *e.g.,* see [31,48]). Nevertheless, they suffice as an independent data point for our experiments because our goal is to measure the *performance* of regular expression matching, and not to test their *effectiveness* at detecting real attacks.
We used traces from weeks two, four and five of the DARPA data set (only the traffic from these weeks contain actual instances of attacks). These traces contain 53,174,585 network packets, amounting to about 11.7 GB worth of data, and contain connections from 8331 distinct source IP addresses. The payloads in the network packets ranged in size from 2 bytes to 1460 bytes, with an average size of 351 bytes (standard deviation of 576).
(3) *Synthetic trace.* Because of the relatively small sizes of payloads in the above traces, we additionally included a synthetic trace. We generated this trace by crawling URLs appearing on Twitter using an in-house crawler built by extending the tweepy python library. As we crawled URLs, we used the tcpdump tool to record the generated HTTP traffic. The resulting trace contained 1.3 GB worth of data in 2,135,578 network packets. The payloads in the network packets were 1202 bytes on average (standard deviation of 472).

FTP traffic. 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 19.4 MB and 24.7 MB worth of data. The traffic consisted of FTP requests to fetch and update technical reports hosted by our department. We observed traffic from 528

[3] The total number of matches triggered shown in Table 1 is 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.

Table 1
Statistics characterizing various aspects of the HTTP traces used in our experiments. The "matches triggered" columns show the total number of signature matches that were triggered (note that a single network packet may trigger multiple signature matches) as well as the number of distinct signatures that matched.

Trace	Number of HTTP commands				Matches triggered: total number (# distinct sigs.)	
	GET	POST	HEAD	PUT	HTTP/1503	HTTP/2612
Rutgers	653,670	137,737	3504	1576	1,816,410 (47)	17,107,588 (120)
DARPA	1,333,469	36,386	450,480	126,824	37,952,078 (121)	190,662,579 (205)
Synthetic	107,436	8220	10,078	15,529	27,948,418 (47)	28,350,698 (135)

Table 2
Statistics showing the number of commands observed in the FTP traces used in our experiments.

Command	CWD	LIST	MDTM	MKD	PASS	PORT	PWD
Number of Instances	62,561	3098	613	89	14,701	232	453
Command	QUIT	RETR	SIZE	STOR	TYPE	USER	–
Number of Instances	12,244	7676	1110	1401	12,201	14,834	–

distinct source IP addresses during this period. Statistics on various FTP commands observed during this period appear in Table 2 (commands that were not observed are not reported). This traffic triggered 9656 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).

Because our primary goal is to study the performance of NFA-OBDDs, we assume that the HTTP and FTP 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.

4.2. Experimental setup

All our experiments were performed on a Intel Core2 Duo E7500 Linux-2.6.27 machine, running at 2.93 GHz with 2 GB 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.

5. Experimental evaluation

This section reports the performance of NFA-OBDDs, and compares them against the performance of NFAs, the PCRE package, which is a popular library for regular expression matching, and variants of DFAs. Our experiments show that NFA-OBDDs:

(1) outperform traditional NFAs by up to three orders of magnitude while retaining their space-efficiency (Section 5.2);
(2) outperform or are competitive in performance with the PCRE package (Section 5.3);
(3) are competitive in performance with variants of DFAs while being drastically less memory-intensive (Section 5.4).

We also present a detailed performance breakdown of NFA-OBDDs in terms of OBDD operations (Section 5.5) and the impact of OBDD variable ordering on NFA-OBDD performance (Section 5.6).

5.1. NFA-OBDDs: construction and performance

We used nfa2obdd to construct NFA-OBDDs from ϵ-free NFAs of the regular expression sets. Table 3 presents statistics on the sizes of the input NFAs, the size of the largest of the four OBDDs in the NFA-OBDD ($\mathrm{OBDD}(\mathcal{T})$), and the time taken and memory consumed by nfa2obdd. For the NFA-OBDDs corresponding to the HTTP signature sets, the vectors $\vec{x}$ and $\vec{y}$ had 18 Boolean variables each, while the vector

Table 3
NFA-OBDD construction results.

Signature set	#Reg. exps.	Size of the input NFA		$\lvert\mathrm{OBDD}(\mathcal{T})\rvert$	Construction
		#States	#Transitions	#Nodes	Time/memory
HTTP (March 2007)	1503	159,734	3,986,769	659,981	305 s/176 MB
HTTP (October 2009)	2612	239,890	5,833,911	989,236	453 s/176 MB
FTP (October 2009)	98	26,536	5,927,465	69,619	246 s/34 MB

$\vec{i}$ had 8 Boolean variables to denote the 256 possible ASCII characters. For the NFA-OBDD corresponding to the FTP signature set, the vectors $\vec{x}$ and $\vec{y}$ 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.

Fig. 5 depicts the performance of NFA-OBDDs. Fig. 5(a) and (b) show the performance for the HTTP traces, while Fig. 5(c) shows the performance for the FTP traces. Table 4 presents the raw throughput and memory consumption of NFA-OBDDs observed for each signature set. The throughput and memory consumption of NFA-OBDDs varies slightly across different traces for each signature set. This difference was most pronounced for the HTTP/2612 signature set, where the Rutgers trace was processed almost 1.8× faster than the DARPA trace. The variance in performance can be attributed to the size and shapes of OBDD($\mathcal{F}$) (the OBDD of the NFA's frontier) observed during execution.

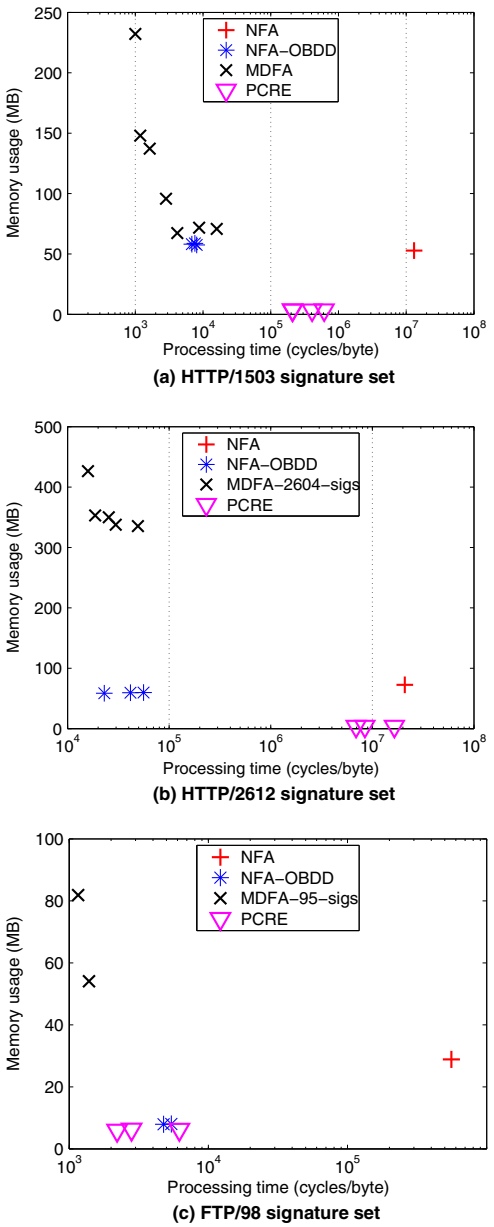


Fig. 5. Comparing memory versus processing time of NFA-OBDDs, traditional NFAs, the PCRE package, and different MDFAs for the Snort HTTP and FTP signature sets. The *x*-axis is in log-scale. Note that Fig. 5(b) and (c) only report the performance of MDFAs with 2604 and 95 regular expressions, respectively.

5.2. 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 σ (the input symbol), and compute the new frontier as $F' = \bigcup_{s \in F} T_s$. The performance and memory consumption of our NFA implementation (as also the PCRE package and DFA variants in Sections 5.3 and 5.4) was relatively stable across all the traces for each signature set. Fig. 5 therefore reports only the averages across these traces.

As Fig. 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 Fig. 5(a), for example, NFA-OBDDs are approximately 1600×–1800× faster than NFAs while consuming almost 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 3.2, the benefits of NFA-OBDDs are more pronounced if larger frontiers are to be processed. Since there is 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 FTP signatures. Nevertheless, *these results clearly demonstrate that OBDDs can improve the time-efficiency of NFAs without compromising their space-efficiency.*

5.3. Comparison with the PCRE package

We compared the performance of NFA-OBDDs with that of the PCRE package, which is a popular library for regular expression matching that is used by a number of tools, including Snort and Perl. PCRE represents regular expressions using a tree-like structure. For a given input string, this algorithm iteratively explores paths in this structure until it finds an accepting state; if so, it declares a match. If it fails to find an accepting state in one path, it backtracks and tries another path until all paths have been exhausted.

Fig. 5 reports three numbers for the performance of the PCRE package, corresponding to different values of configuration parameters of the package.[4] In both Fig. 5(a) and

[4] These parameters determine whether PCRE must process input in the ASCII or Unicode formats, and whether the matching algorithm must terminate after finding the first matching substring or all matching substrings. Of the four possible configurations based upon these two parameters, we were able to experiment with only three because the PCRE package crashed under one of the configurations.

Table 4
Raw performance numbers for the charts shown in Fig. 5.

Signature set	Processing time	Memory
NFA-OBDDs		
HTTP/1503	6844–7582 cycles/byte	58 MB
HTTP/2612	22,968–41,588 cycles/byte	61 MB
FTP/98	5095 cycles/byte	8 MB
NFAs		
HTTP/1503	1.3×10^7 cycles/byte	53 MB
HTTP/2612	2.1×10^7 cycles/byte	73 MB
FTP/98	5.6×10^5 cycles/byte	29 MB
PCRE		
HTTP/1503	2.1×10^5–6.2×10^5 cycles/byte	3.6 MB
HTTP/2612	1.3×10^7–2.8×10^7 cycles/byte	3.9 MB
FTP/98	2210–6185 cycles/byte	5.9–6.2 MB
MDFA (partial signature sets in Fig. 5(b) and (c))		
HTTP/1503	1000–15,951 cycles/byte	71–232 MB
HTTP/2604	15,891–49,296 cycles/byte	335–426 MB
FTP/95	1160–1386 cycles/byte	54–82 MB

(b), NFA-OBDDs outperform the PCRE package. The throughput of NFA-OBDDs is about an order of magnitude 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 approximately 300×–500×. 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 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 about 4 MB memory.

For the FTP signatures (Fig. 5(c)), NFA-OBDDs are about 2.5× 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 [51] to parse input strings. Despite this gap in performance, Cox [14] 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 [43].

5.4. Comparison with DFA variants

5.4.1. Multiple DFAs

We compared the performance of NFA-OBDDs with a variant of DFAs, called multiple DFAs (MDFAs), produced by set-splitting [58].[5] 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 executed 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 [58], we produced several MDFAs by combining the Snort signatures in several ways, each with different space/time utilization. Each point in Fig. 5 denotes the performance of *one* MDFA (again, averaged over all the input traces), which in turn consists of a collection of 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 choice (*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 techniques, such as on-the-fly determinization [47] 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.

Fig. 5 show 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 Fig. 5(b) (constructed for a subset of 2604 signatures) offered about 50% more throughput than NFA-OBDDs, but consumed 7× 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× 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 also 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*

[5] We were unable to compare the performance of NFA-OBDDs against DFAs because DFA construction ran out of memory. However, prior work [45] estimates that DFAs may offer throughputs of about 50 cycles/byte.

of NFAs and even make them competitive with MDFAs, which are a deterministic 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, NFA-OBDDs were produced automatically from regular expressions. In contrast, 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 [58]. 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 in attempting to construct 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 in a straightforward manner from regular expressions, including those with counters and other complex structural patterns, and are yet competitive in performance and more compact than MDFAs.

5.4.2. Hybrid finite automata

Finally, we also attempted to compare the performance of NFA-OBDDs with a variant of DFAs, called hybrid finite automata (HFA) [4]. HFAs are a hybrid of NFAs and DFAs, and 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 [4] 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. It may be possible to construct a *collection* of HFAs in a manner akin to MDFAs, but we did not consider this design in our experiments.

5.5. Deconstructing NFA-OBDD performance

We further analyzed the performance of NFA-OBDDs to understand the time consumption of each OBDD operation. The results of this analysis can motivate techniques to optimize OBDD packages to further improve the efficiency of NFA-OBDDs. The results reported in this section are based upon the HTTP/1503 signature set; the results with the other signature set were similar.

Table 5 shows the fraction of time that exec_nfaobdd spends performing various OBDD operations as it processes a single input symbol. These include the operations needed to compute a new frontier and those needed to check if the frontier contains an accepting configuration.

Table 5
Fraction of time spent performing OBDD operations.

Operation	Fraction (%)
ANDABSTRACT	50
AND	39
MAP	4
Acceptance check	7

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 $\vec{y}$ variables in $\mathcal{G}(\vec{y})$ to $\vec{x}$ variables instead.

We implemented the Boolean operations required to obtain a new frontier (described in Section 3.1) using one set of AND, ANDABSTRACT and MAP operations. Each ANDABSTRACT step existentially quantifies 26 Boolean variables (the $\vec{x}$ and $\vec{i}$ 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 Table 5.

Table 5 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 for more efficient ANDABSTRACT and AND operations) can further improve the performance of NFA-OBDDs.

5.6. Impact of variable ordering on NFA-OBDD performance

As mentioned in Section 2, the size of an OBDD is sensitive to the total order imposed on its variables. Bryant [10] showed that it is NP-hard to determine whether a particular variable ordering minimizes the size of an OBDD for a Boolean function. Variable order also impacts the structure of OBDDs, and in our experience the order of the variables in the vectors $\vec{i}$, $\vec{x}$ and $\vec{y}$ influences the performance of NFA-OBDDs.

We experimented with various total orders to empirically determine their impact on the size and throughput of NFA-OBDDs before settling on one of the total orders that yielded the best performance ($\vec{i} < \vec{x} < \vec{y}$) for the numbers reported earlier in this section.

Fig. 6 compares the performance of NFA-OBDDs constructed using seven total orders (all four constituent OBDDs of each NFA-OBDD use the same total order):

(1) $\vec{i} < \vec{x} < \vec{y}$: the variables within each vector are arranged in increasing order from most significant bit (MSB) to least significant bit (LSB);

(2) $\vec{x} < \vec{i} < \vec{y}$: in-vector order is similar to the case above;
(3) $\vec{y} < \vec{x} < \vec{i}$: in-vector order is similar to the case above;
(4) $\vec{i} <$ Interleave$[\vec{x} < \vec{y}]$: variables in the vector $\vec{i}$ appear before the variables in $\vec{x}$ and $\vec{y}$. The variables in $\vec{x}$ and $\vec{y}$ appear interleaved, with a variable in $\vec{x}$ appearing before the corresponding variable in $\vec{y}$. The variables in each vector increase from MSB to LSB;
(5) $\vec{i} <$ Interleave$[\vec{y} < \vec{x}]$: similar to the case above, except that variables in $\vec{y}$ appear before their counterparts in $\vec{x}$;
(6) Interleave$[\vec{x} < \vec{y}] < \vec{i}$: as above, except that the interleaved variables of $\vec{x}$ and $\vec{y}$ appear in the total order before the variables of $\vec{i}$;
(7) Interleave$[\vec{y} < \vec{x}] < \vec{i}$: as above, with the variables in $\vec{y}$ preceding their counterparts in $\vec{x}$.

The above variable orders are only a tiny fraction of the set of possible total orders, which is exponential in the number of variables. However, they provide insight into which orders empirically provide high-performance NFA-OBDDs. We considered NFA-OBDDs for HTTP/1503, HTTP/2612 and FTP/98 to determine whether the performance NFA-OBDDs for different signature sets is sensitive to the order imposed on the variables. As before, we used exec_nfaobdd to feed network traces to these NFA-OBDDs. For these experiments, we only used the network traces collected at Rutgers.

Fig. 6 presents the results of these experiments, showing both the throughput and overall memory consumption of exec_nfaodd. It shows that the total order $\vec{i} < \vec{x} < \vec{y}$ performs consistently well across all three signature sets, but consumes more memory than the most compact implementation. The total order $\vec{i} <$ Interleave$[\vec{x} < \vec{y}]$ also provides competitive performance for the NFA-OBDDs of all three signature sets. However, the performance gap between the best and the worst total orders ($\vec{y} < \vec{x} < \vec{i}$) is almost four orders of magnitude. We used the order $\vec{i} < \vec{x} < \vec{y}$ for the experiments reported earlier in this section, though we could have used $\vec{i} <$ Interleave$[\vec{x} < \vec{y}]$ as well, with a slightly smaller memory consumption for the FTP/98 NFA-OBDD.

This figure also shows that there is no direct correlation between the size and performance of exec_nfaobdd for different variable orders. Although the time complexity of algorithms such as Apply and Restrict asymptotically depends on the size of their input OBDDs, factors such as the structure of the OBDDs also affect the number of graph operations, and therefore the performance of the corresponding NFA-OBDDs.

These experiments lead us to conclude that the performance of NFA-OBDDs is indeed sensitive to the total order imposed on its variables. The vast search space of total orders diminishes hopes of a tractable algorithm to identify the total order that would yield the best-performing NFA-OBDD for a given signature set. Nevertheless, in practice, experiments with a few total orders (such as the ones in Fig. 6) can help empirically determine high-performance NFA-OBDDs. Future work could develop heuristics that leverage the structure of the regular expressions in the input signature set to determine "good" total orders.

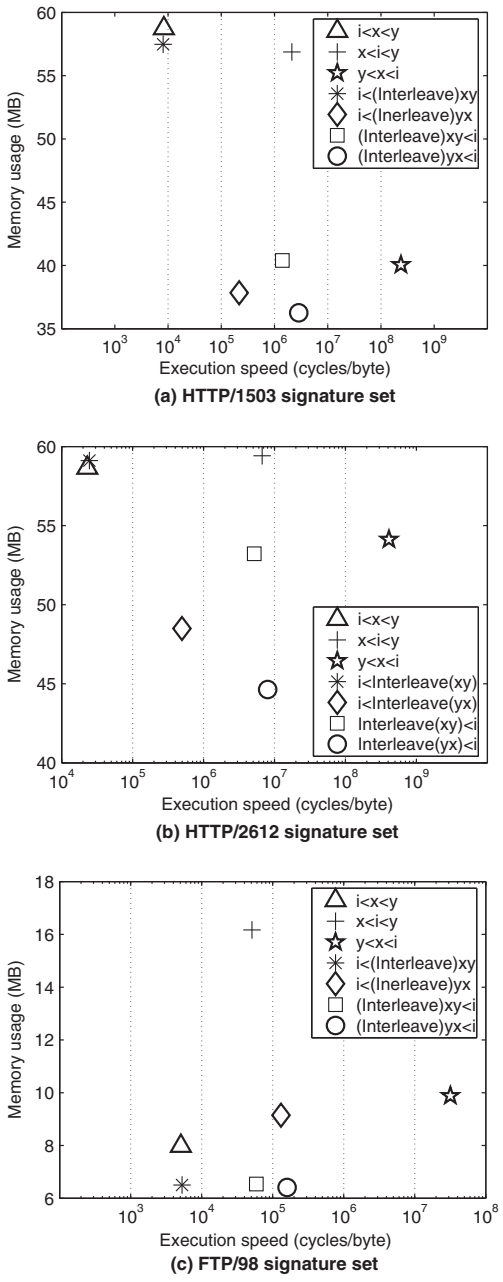


Fig. 6. Impact of OBDD variable ordering on the performance of NFA-OBDDs.

6. 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 [6,8]. In this section, we present one such technique, *k-stride NFAs* [8], 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) ϵ-free NFA (Q,Σ,Δ,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 Σ^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 function Γ of the *k*-stride NFA is computed as a *k*-step closure of Δ, *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 Γ from Δ 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 [6,8] for a detailed description of the construction.

Fig. 7 presents an example of a 2-stride NFA corresponding to the NFA in Fig. 3. The do-not-care symbol is denoted by a "•". Thus, for instance, an input string 101 would be padded with • to become 101•. 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•, 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 Σ^k (plus a new symbol "•"); the vector $\vec{i}$ would therefore contain *k* times as many Boolean variables. However, two additional details must be addressed when applying *k*-stride NFAs (and the corresponding NFA-OBDDs) to the problem of matching traffic patterns in a NIDS, namely, (i) adapting *k*-stride NFAs to work in the streaming model; and (ii) reducing the space consumption of *k*-stride NFAs. These are discussed next.

6.1. Adapting to the streaming model

When operating a 1-stride NFA to process a stream of inputs, the frontier of states must be checked after processing each input symbol to determine whether the input triggered a match. However, this technique does not suffice for *k*-stride NFAs in the streaming model. To see why, consider how the NFAs in Figs. 3 and 7 would process the input 10. The 1-stride NFA would trigger an alert after the first symbol has been processed (because the frontier $F=\{A,B\}$ contains an accepting state). In contrast, the 2-stride NFA would process the entire input 10 in one step, resulting in the frontier $F=\{A\}$, which does not contain the accepting state. Therefore, for *k*-stride NFAs, it does not suffice to simply check *F* to determine acceptance.

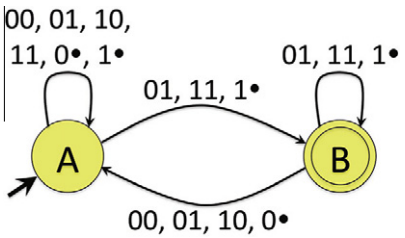


Fig. 7. 2-stride NFA for Fig. 3.

To address this problem, the algorithm to convert a 1-stride NFA into a *k*-stride NFA must "remember" that an accepting state of the 1-stride NFA was encountered when computing the *k*-step closure of the transition relation Δ of the 1-stride NFA. One way to compute *k*-stride NFAs that achieve this goal is by *adding a new accepting state* to the *k*-stride NFA. This algorithm adds incoming transitions to the new accepting state suitably from other states in the *k*-stride NFA to "remember" that a substring of the input *k*-gram would have triggered a match in the 1-stride NFA. The resulting *k*-stride NFA $(Q^*,\Sigma^k,\Gamma^*,q_0,F^*)$ has the same semantics as the 1-stride NFA in the streaming model (*i.e.,* they both accept the same set of strings), but adds a state to the 1-stride NFA. We refer the reader to prior work [6,8] for details on this algorithm.[6] This *k*-stride NFA can be converted into an NFA-OBDD using the same technique presented in Section 3, and operated in the same way.

Fig. 8 presents an example of this approach applied to the 2-stride NFA in Fig. 7. The new transition from state A to accepting state C on input 10 uses the state C to remember that the substring 1 of 10 triggered a match in the corresponding 1-gram NFA. The state C will therefore be in the frontier when an input 10 is processed at state *A*, thereby triggering a match when the OBDD operation to check acceptance is performed. However, there are no outgoing transitions from *C*. This state is therefore removed from the frontier when the next digram is processed by the 2-stride NFA (unless that digram also triggers acceptance).

6.2. Reducing space consumption using alphabet compression

The transition table of a *k*-stride NFA can have $O(|Q|\times|\Sigma|^k)$ entries, each of which can be of size $O(|Q|)$ (to store the set of "next" states, which can be $O(|Q|)$), which can result in a memory utilization of $O(|Q|^2\times|\Sigma|^k)$. However, this asymptotic limit is rarely reached in practice, and transition tables encountered in practice are generally sparse. In particular, there may be several transitions labeled with the same set of symbols from the alphabet Σ^k. That is, if for any state $s\in Q$, and input symbols σ_1 and σ_2, if $\Gamma(s,\sigma_1)=\Gamma(s,\sigma_2)$, then the symbols σ_1 and σ_2 can potentially be merged into an equivalence class. This idea is called *alphabet compression* [2,5,8,24,27].

The output of an alphabet compression algorithm is a partition of Σ^k into equivalence classes. Each equivalence

[6] This algorithm can also be adapted easily to identify the regular expression that matched the input.

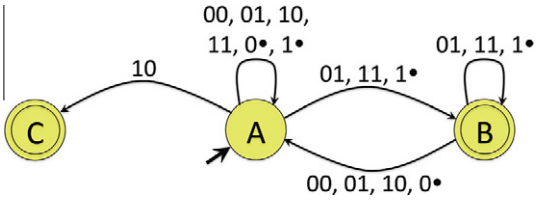


Fig. 8. The NFA in Fig. 7 adapted for streaming.

class is assigned a symbol, thereby yielding a new alphabet $\mathcal{E}$ with fewer elements than Σ^k. An alphabet compression algorithms also outputs an *encoding function* $m : \Sigma^k \to \mathcal{E}$ that translates elements in Σ^k to elements in $\mathcal{E}$. In the above example, $m(\sigma_1) = m(\sigma_2)$. The transitions of an alphabet-compressed NFA would also be appropriately relabeled to use symbols from $\mathcal{E}$ instead. Similarly, symbols in the input would also have to be appropriately translated using m before they are passed to the NFA for matching. 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.

We implemented the alphabet compression algorithm described by Brodie et al. [8] for 2-stride NFAs and empirically found that an alphabet compression reduces the memory consumption of 2-stride NFAs. However, this alphabet compression algorithm itself is quite resource-intensive because it operates on the transition relation of the *entire* (2-stride) NFA, thereby causing the algorithm to exhaust the available memory on our machine. For example, we found that Brodie et al.'s algorithm frequently ran out of memory when processing 2-stride NFAs obtained by combining more than 200 regular expressions from the HTTP/1503 and HTTP/2612 signature sets.

```
Algorithm: Combine_Compressed_Alphabet (X, Y)
Input: X = {X_1,...,X_p} and Y = {Y_1,...,Y_q}, the
   compressed alphabet of NFA_X and NFA_Y
Output: Z, the compressed alphabet of
   NFA_Z = NFA_X ∪ NFA_Y
1          Z = X ∪ Y
2          Z' = ∅
3          foreach (A ∈ Z) do
4             split = false
5             foreach (B ∈ Z such that B ≠ A) do
6                if(A ∩ B ≠ ∅) then
7                   Z' = Z' ∪ (A ∩ B) ∪ (A − B) ∪ (B − A);
8                   split = true
9             if (split == false) then Z' = Z' ∪ A;
10         if (Z ≠ Z') then
11            Z = Z'
11            goto line 2;
13         return Z;
```

We therefore developed a scheme (Algorithm 1) that applies Brodie et al.'s algorithm to smaller NFAs (thereby limiting the algorithm's memory consumption), and merges the results to obtain a compressed alphabet for the combination of the smaller NFAs. Our scheme is based upon the following fact: if two symbols σ_1 and σ_2 appear in the same equivalence class of the compressed alphabet of each of two NFAs (say, NFA_X and NFA_Y), then they will appear in the same equivalence class of the NFA (say NFA_Z) obtained by merging the set of states and transitions of NFA_X and NFA_Y. Algorithm 1 uses this observation to combine the compressed alphabet X and Y of NFA_X and NFA_Y and produce the compressed alphabet Z of the NFA_Z. It proceeds by combining X and Y into a set Z, and iteratively refining Z (in line 7) so that if any two symbols σ_1 and σ_2 appear in the same equivalence class in the output set Z, then they also appear in the same equivalence classes in *both* X and Y.

Our experiments confirm the scalability of Algorithm 1. For example, we were able to use this algorithm to compress the alphabet of the 2-stride NFA representing the 2604 signatures from the HTTP/2612 set.[7] We did so by first splitting these signatures into 61 smaller subsets, applying Brodie et al.'s alphabet compression to the 2-stride NFAs representing these subsets, and combining the compressed alphabet pairwise using Algorithm 1. The size of the compressed alphabet of the 2-stride NFA was 11,119. In contrast, Brodie et al.'s algorithm ran out of memory when processing the 2-stride NFA representing set of 2604 signatures in its entirety.

6.3. 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. Although our implementation accepts k as an input parameter, we have only conducted experiments for $k = 2$ because our alphabet compression algorithm 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/2604, a subset of 2604 HTTP signatures from HTTP/2612 and (2) FTP/95, a subset of 95 FTP signatures from FTP/98 (we omitted three signatures for the reason discussed in Footnote 7).

Table 6 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 about 1.6 GB memory. Fig. 9(a) and (b) compare the performance of 1-stride NFA-OBDDs with the performance of 2-stride NFA-OBDDs (using the traces described in Section 4). As expected, these figures show that matching multiple bytes in the input stream improves the performance of NFA-OBDDs, roughly doubling the throughput in each case. In fact, the 2-stride NFA-OBDD of the HTTP/2612 signature set more than dou-

[7] We excluded 8 signatures from the HTTP/2612 signature set because alphabet compression ran out of memory for these 8 signatures. These 8 signatures contained complex structural patterns that caused Brodie et al.'s compression algorithm to run out of memory for 2-stride NFAs representing *each* of these signatures (so Algorithm 1 was never invoked). Further research is needed to develop alphabet compression algorithms that can handle such complex signatures.

Table 6
2-stride NFA-OBDD construction results.

Signature set	#States	#Transitions in NFA (1-stride | 2-stride)		#Alphabet symbols
HTTP/2604	237,972	5,567,317	136,212,770	11,119
FTP/95	15,266	3,361,065	5,136,420	848

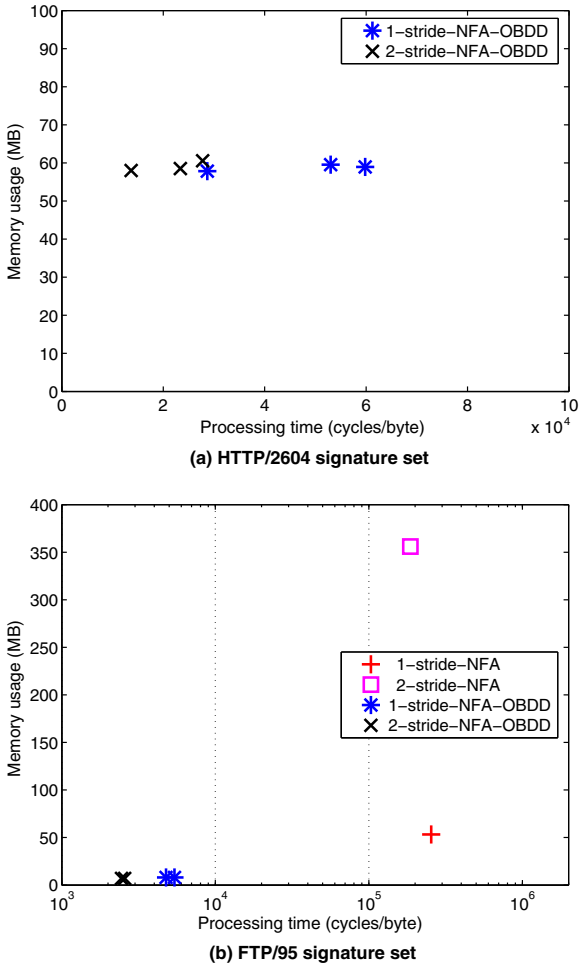


Fig. 9. Memory versus throughput for 1-stride and 2-stride NFA-OBDDs. Fig. 9(b) also shows the performance of the corresponding 1-stride and 2-stride NFAs.

bled (2.26×) the throughput of the 1-stride NFA-OBDD on the DARPA trace.

These experiments also demonstrate that the use of OBDDs allows 2-stride NFA-OBDDs to be *more space-efficient* than NFAs. While we were able to create and operate a 2-stride NFA-OBDD for the HTTP/2604 signature set, the 2-stride NFA for this signature set exhausted the memory available on our machine. We were able to create a 2-stride NFA for the FTP/95 signature set; Fig. 9(b) depicts the performance of both the 1-stride and 2-stride NFAs for this signature set. As this figure shows, the memory utilization of the 2-stride NFA-OBDD is about two orders of magnitude smaller than that of the 2-stride NFA, and is also about two orders of magnitude faster.

These results lead us to conclude 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 (for higher values of k) is a topic for future work.

7. 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 [1]). They are ideally suited for wire-speed intrusion detection, and have been implemented both in software and hardware [16,30,49,50,52,53]. However, prior work has shown that string-based signatures can easily be evaded by malware using polymorphism, metamorphism and other mutations [20,25,35,39]. The research community has therefore been investigating sophisticated signature schemes, such as session signatures [37,47,57] and vulnerability signatures [9,54], 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 (*e.g.,* Tipping Point,[8] LSI Corporation[9] and Cisco[10]).

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 [47]; MDFAs, which combine signatures into multiple DFAs (as discussed in Section 5) [58]; D^{2}FAs [28], which reduce the memory footprint of DFAs via edge compression; and XFAs [44,45], 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.,* [7,28,32,45]) 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

[8] See http://www.tippingpoint.com.
[9] Tarari RegEx content processor: http://www.tarari.com.
[10] See the IOS terminal services configuration guide: http://tinyurl.com/2eouvq.

from regular expressions in a fully automated way. This is in contrast to XFAs [44], 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 [58]. Last, because these techniques operate with DFAs, they may sometimes encounter regular expressions that are hard to determinize. For example, Smith et al. [44, Section6.2] present a regular expression from the Snort data set for which the XFA construction algorithm runs out of memory. In contrast, 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 [13,23,33,40]. 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 [13,23,33,40].

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 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 [60] and FPGAs [42]. Such an implementation of NFA-OBDDs can be used to improve the performance of hardware-based NFAs as well.

8. Summary and future work

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 outperform NFAs by almost three orders of magnitude. We also showed how OBDDs can enhance the performance of NFAs that match multiple input symbols in a single step.

There are several avenues of future work to improve the results reported in this article. We outline three promising directions below.

(1) *Implementing NFA-OBDDs in hardware.* NIDS vendors are increasingly beginning to deploy hardware-based deep packet inspection products. While this article explored the potential of NFA-OBDDs using a software-based implementation, a hardware-based solution would be required to provide raw matching speeds approaching multiple gigabit/s. Although OBDDs [42,60] and NFAs [13,40] have each been individually implemented in hardware (such as FPGAs and CAMs), further research is needed to investigate the possibility of a hardware-based NFA-OBDD implementation. The key challenge in implementing NFA-OBDDs in hardware is to devise techniques that would allow OBDDs to be modified within hardware, *e.g.,* to allow $\mathrm{OBDD}(\mathcal{F})$ to be modified efficiently as each input alphabet is processed.

(2) *Relaxed variants of OBDDs.* NFA-OBDDs improve the performance of NFAs because OBDDs are a compact way to represent the transition relation and frontier sets. OBDDs were primarily developed to provide a canonical representation of Boolean functions, *i.e.,* they satisfy the property that for a fixed total order of variables, two OBDDs representing the same Boolean function are isomorphic. Algorithms such as APPLY and RESTRICT include several steps that ensure this property, which in turn affects the raw performance of NFA-OBDDs. Future work could investigate relaxed variants of OBDDs that do not satisfy this property, yet provide fast and space-efficient regular expression matching.

(3) *Beyond regular expressions.* NFA-OBDDs provide fast, space-efficient matching of regular expressions. However, modern intrusion detection systems also include a significant fraction of patterns that are extensions of regular expressions (*e.g.,* back-references). Packages such as PCRE use backtracking algorithms to match traffic against such patterns, which may result in inefficient and memory-intensive operation. Future work could investigate whether OBDDs can improve the time- and space-efficiency of algorithms to match such regular expression extensions.

In summary, the main contribution of this paper is in showing that the use of OBDDs drastically improves NFA performance and brings them within the realm of feasible use in intrusion detection systems. In the light of this contribution and the space-efficiency of NFAs, we conclude with a call for further research on the use of NFAs to represent signatures.

Acknowledgments

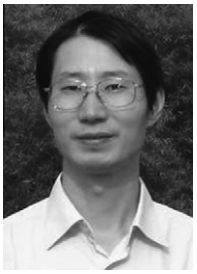
We thank Cristian Estan and Somesh Jha for useful discussions in the early stages of this project. This work was supported in part by NSF Grants 0831268, 0915394, 0931992 and 0952128.

References

[1] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.
[2] A.V. Aho, S.C. Johnson, Optimal code generation for expression trees, in: ACM Symposium on Theory of Computing, ACM, New York, NY, USA, 1975.

[3] M. Becchi, S. Cadambi, Memory-efficient regular expression search using state merging, in: Proceedings of IEEE Infocom, 2007.
[4] M. Becchi, P. Crowley, A hybrid finite automaton for practical deep packet inspection, in: International Conference on emerging Networking EXperiments and Technologies, 2007
[5] M. Becchi, P. Crowley, An improved algorithm to accelerate regular expression evaluation, in: International Conference on Architectures for Networking and Communication Systems, ACM, 2007, pp. 145–154.
[6] M. Becchi, P. Crowley, Efficient regular expression evaluation: theory to practice, in: International Conference on Architectures for Networking and Communication Systems, ACM, 2008, pp. 50–59.
[7] B. Brodie, R. Cytron, D. Taylor, A scalable architecture for high-throughput regular-expression pattern matching, SIGARCH Comput. Archit. News 34 (2) (2006) 191–202.
[8] B.C. Brodie, D.E. Taylor, R.K. Cytron, A scalable architecture for high-throughput regular-expression pattern matching, in: International Symposium on Computer Architecture, IEEE Computer Society, 2006, pp. 191–202.
[9] D. Brumley, J. Newsome, D. Song, H. Wang, S. Jha, Towards automatic generation of vulnerability-based signatures, in: IEEE Symposium on Security and Privacy, 2006.
[10] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. 35 (8) (1986) 677–691.
[11] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, J. Hwang, Symbolic model checking: 10^{20} states and beyond, in: Symposium on Logic in Computer Science, IEEE Computer Society, 1990, pp. 401–424.
[12] M. Christiansen, E. Fleury, An MTIDD based firewall, Telecommunication Systems 27 (2–4) (2004).
[13] C.R. Clark, D.E. Schimmel, Scalable pattern matching for high-speed networks, in: IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2004, pp. 249–257.
[14] Cox, R., 2007. Regular expression matching can be simple and fast (but is slow in Java, Perl, PHP, Python, Ruby, ...). <http://swtch.com/rsc/regexp/regexp1.html>.
[15] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, J.W. Lockwood, Deep packet inspection using parallel bloom filters, IEEE Micro. 24 (1) (2004) 52–61.
[16] S. Dharmapurikar, J.W. Lockwood, Fast and scalable pattern matching for network intrusion detection systems, J. Selected Areas Commun. 24 (10) (2006) 1781–1792.
[17] R.W. Floyd, J.D. Ullman, The compilation of regular expressions into integrated circuits, J. ACM 29 (3) (1982).
[18] M.G. Gouda, X. Liu, Firewall design: consistency, completeness and compactness, in: International Conference on Distributed Computing Systems, 2004.
[19] J.D. Guttman, A.L. Herzog, Rigorous automated network security management, Int. J. Inform. Secur. 4 (1–2) (2004).
[20] M. Handley, V. Paxson, C. Kreibich, Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics, in: Usenix Security. USENIX, 2001, p. 9.
[21] S. Hazelhurst, A. Fatti, A. Henwood, Binary decision diagram representations of firewall and router access lists. Tech. rep., University of Witwatersrand, Johannesburg, South Africa, 1998.
[22] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, third ed., Addison-Wesley, 2007.
[23] B.L. Hutchings, R. Franklin, D. Carver, Assisting network intrusion detection with reconfigurable hardware, in: Annual Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2002, pp. 111–120.
[24] S. Johnson, Yacc – yet another compiler compiler. Computing Science Tech. Rep. 32, AT&T Bell Labs, 1975.
[25] M. Jordan, Dealing with Metamorphism, Virus Bulletin Weekly, 2002.
[26] H. Kim, B. Karp, Autograph: Toward automated, distributed worm signature detection, in: USENIX Security Symposium, 2004, pp. 271–286.
[27] S. Kong, R. Smith, C. Estan, Efficient signature matching with multiple alphabet compression tables, in: Interntional Conference on Security and Privacy in Communication Networks, 2008.
[28] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, J. Turner, Algorithms to accelerate multiple regular expressions matching for deep packet inspection, in: ACM SIGCOMM Conference, ACM, 2006, pp. 339–350.
[29] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba, K. Das, The 1999 DARPA off-line intrusion detection evaluation, Comput. Networks 34 (4) (2000) 579–595.
[30] R. Liu, N. Huang, C. Chen, C. Kao, A fast string-matching algorithm for network processor-based intrusion detection system, Trans. Embedded Comput. Syst. 3 (3) (2004) 614–633.
[31] J. McHugh, Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln laboratories, ACM Trans. Inform. Syst. Secur. 3 (4) (2000) 262–294.
[32] C. Meiners, J. Patel, E. Norige, E. Torng, A.X. Liu, Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems, in: 19th USENIX Security Symposium, 2010.
[33] A. Mitra, W. Najjar, L. Bhuyan, Compiling PCRE to FPGA for accelerating Snort IDS, in: Symposium on Architectures for Networking and Communications Systems, ACM, 2007, pp. 127–136.
[34] J. Newsome, B. Karp, D. Song, Polygraph: automatically generating signatures for polymorphic worms, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2005, pp. 226–241.
[35] T. Ptacek, T. Newsham, Insertion, evasion and denial of service: eluding network intrusion detection, 2001. <http://insecure.org/stf/secnet_ids/secnet_ids.html>.
[36] M. Roesch, Snort – lightweight intrusion detection for networks. In: USENIX Conf. on System Administration. USENIX, 1999, pp. 229–238. <http://www.usenix.org/publications/library/proceedings/lisa99/roesch.html>.
[37] S. Rubin, S. Jha, B. Miller, Language-based generation and evaluation of NIDS signatures, in: Symposium on Security and Privacy, Oakland, California, 2005.
[38] N. Schear, D.R. Albrecht, N. Borisov, High-speed matching of vulnerability signatures, in: R. Lippman, E. Kirda, A. Trachtenberg (Eds.), RAID 2008, Lecture Notes in Computer Science, 5230, Springer, 2008, pp. 155–174.
[39] U. Shankar, V. Paxson, Active mapping: resisting NIDS evasion without altering traffic, in: Symposium on Security and Privacy, IEEE Computer Society, 2003, pp. 44–61.
[40] R. Sidhu, V. Prasanna, Fast regular expression matching using FPGAs, in: Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society, 2001, pp. 227–238.
[41] S. Singh, C. Estan, G. Varghese, S. Savage, Automated worm fingerprinting, in: USENIX/ACM Symposium on Operating System Design and Implementation, 2004, pp. 45–60.
[42] R. Sinnappan, S. Hazelhurst, A reconfigurable approach to packet filtering, in: G. Brebner, R. Woods (Eds.), FPL 2001, Lecture Notes in Computer Science, vol. 2147, Springer, 2001, pp. 638–642.
[43] R. Smith, C. Estan, S. Jha, Backtracking algorithmic complexity attacks against a NIDS, in: Annual Computer Security Applications Conf, IEEE Computer Society, 2006, pp. 89–98.
[44] R. Smith, C. Estan, S. Jha, XFA: faster signature matching with extended automata, in: Symposium on Security and Privacy, IEEE Computer Society, 2008, pp. 187–201.
[45] R. Smith, C. Estan, S. Jha, S. Kong, Deflating the Big Bang: Fast and scalable deep packet inspection with extended finite automata, in: SIGCOMM Conference, ACM, 2008, pp. 207–218.
[46] F. Somenzi, CUDD: CU decision diagram package, release 2.4.2. Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, 2009. <http://vlsi.colorado.edu/fabio/CUDD>.
[47] R. Sommer, V. Paxson, Enhancing byte-level network intrusion detection signatures with context, in: Conference on Computer and Communication Security, ACM, 2003, pp. 262–271.
[48] R. Sommer, V. Paxson, Outside the closed world: on using machine learning for network intrusion detection, in: Symposium on Security and Privacy, IEEE Computer Society, 2010.
[49] I. Sourdis, D. Pnevmatikatos, Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system, in: P. Cheung, G. Constantinides, J. Sousa (Eds.), FPL 2003, Lecture Notes in Computer Science, vol. 2778, Springer, 2003, pp. 880–889.
[50] L. Tan, T. Sherwood, A high throughput string matching architecture for intrusion detection and prevention, in: International Symposium Computer Architecture, IEEE Computer Society, 2005, pp. 112–122.
[51] K. Thompson, Programming techniques: regular expression search algorithm, Communications on ACM 11 (6) (1968) 419–422.
[52] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, in: IEEE INFOCOM, IEEE Computer Society, 2004, pp. 333–340.
[53] G. Vasiliadis, S. Antonatos, M. Polychronakis, E.P. Markatos, S. Ioannidis, Gnort: high performance network intrusion detection using graphics processors, in: R. Lippman, E. Kirda, A. Trachtenberg

(Eds.), RAID 2008, Lecture Notes in Computer Science, 5230, Springer, 2005, pp. 116–134.
[54] H.J. Wang, C. Guo, D.R. Simon, A. Zugenmaier, Shield: vulnerability-driven network filters for preventing known vulnerability exploits, in: ACM SIGCOMM, 2004.
[55] L. Yang, R. Karim, V. Ganapathy, R. Smith, Improving NFA-based signature matching using ordered binary decision diagrams, in: International Symposium on Recent Advances in Intrusion Detection, 2010.
[56] L. Yang, R. Karim, V. Ganapathy, R. Smith, Signatures referenced in Sections 5 and 6, 2010. Available at RAID 2008: <http://www.cs.rutgers.edu/~vinodg/papers/raid2010>.
[57] V. Yegneswaran, J.T. Giffin, P. Barford, S. Jha, An architecture for generating semantics-aware signatures, in: USENIX Security Symposium, Baltimore, Maryland, 2005.
[58] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, R.H. Katz, Fast and memory-efficient regular expression matching for deep packet inspection, in: ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2006, pp. 93–102.
[59] L. Yuan, J. Mai, Z. Su, H. Chen, C. Chuah, P. Mohapatra, FIREMAN: a toolkit for firewall modeling and analysis, in: Symposium on Security and Privacy, 2006.
[60] S. Yusuf, W. Luk, Bitwise optimized CAM for network intrusion detection systems, in: International Conference on Field Prog. Logic and Applications, IEEE Press, 2005, pp. 444–449.



Liu Yang is a Ph.D. student in Computer Science at Rutgers University. He received a M.Sc. degree in Computer Science from Stevens Institute of Technology in 2009. He earned a M.Sc. and a Bachelor's degree in Mathematics from Sichuan University and Chongqing Normal University in China. His research interests include intrusion detection, applied cryptography, and anti-phishing.

Rezwana Karim is a Ph.D. student in Computer Science at Rutgers University. She obtained a B.Sc. degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology (BUET) in 2007. Her research interests are in systems and network security.

Vinod Ganapathy is an assistant professor of Computer Science at Rutgers University. He earned a B.Tech. degree in Computer Science & Engineering from IIT Bombay in 2001 and a Ph.D. degree in Computer Science from the University of Wisconsin-Madison in 2007. He is broadly interested in computer security and privacy.

Randy Smith received his Ph.D. in Computer Sciences in 2009 from the University of Wisconsin-Madison and holds a Master's degree in Computer Science and a Bachelor's Degree in Mathematics from Brigham Young University. Dr. Smith's research interests include algorithms for network analysis and understanding, and intrusion detection and prevention.