Efficient Signature Matching with Multiple Alphabet Compression Tables

Shijin Kong* Randy Smith Cristian Estan
Computer Sciences Department
University of Wisconsin-Madison
{krobin,smithr,estan}@cs.wisc.edu

ABSTRACT

Signature matching is a performance critical operation in intrusion prevention systems. Modern systems express signatures as regular expressions and use Deterministic Finite Automata (DFAs) to efficiently match them against the input. In principle, DFAs can be combined so that all signatures can be examined in a single pass over the input. In practice, however, combining DFAs corresponding to intrusion prevention signatures results in memory requirements that far exceed feasible sizes. We observe for such signatures that distinct input symbols often have identical behavior in the DFA. In these cases, an Alphabet Compression Table (ACT) can be used to map such groups of symbols to a single symbol to reduce the memory requirements.

In this paper, we explore the use of multiple alphabet compression tables as a lightweight method for reducing the memory requirements of DFAs. We evaluate this method on signature sets used in Cisco IPS and Snort. Compared to uncompressed DFAs, multiple ACTs achieve memory savings between a factor of 4 and a factor of 70 at the cost of an increase in run time that is typically between 35% and 85%. Compared to another recent compression technique, D^2FAs, ACTs are between 2 and 3.5 times faster in software, and in some cases use less than one tenth of the memory used by D^2FAs. Overall, for all signature sets and compression methods evaluated, multiple ACTs offer the best memory versus run-time trade-offs.

Categories and Subject Descriptors: C.2.0 [Computer Communication Networks]: General - Security and protection (e.g., firewalls)

General Terms: Algorithms, Performance, Security

Keywords: Signature matching, deep packet inspection, regular expressions, alphabet compression

*Employed by Cisco Systems, Inc. at the time of publication; email: shikong@cisco.com

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SecureComm 2008, September 22 - 25, 2008, Istanbul, Turkey.
Copyright 2008 ACM 978-1-60558-241-2 ...$5.00.

1. INTRODUCTION

Network Intrusion Detection Systems (NIDS) and Intrusion Prevention Systems (IPS) have become critical components of modern network infrastructure. Functionally, at the core of any IPS there resides a signature matching engine that potentially compares every byte of incoming and outgoing traffic to a signature database containing known exploits or misuses. However, steady increases in traffic volume, growing signature database size, and increased signature complexity have turned signature matching into a performance-limiting bottleneck.

IPS performance is limited to the speed at which network traffic can be matched against a set of signatures. Thus, the *language* used to express signatures and, correspondingly, the data structures and procedures used to represent and match input to strings in that language have a tremendous impact on performance. Modern IPSes use regular expressions as the language for writing signatures due to their greater expressiveness over strings [7, 19], and Deterministic Finite Automata (DFAs) are a commonly used representation for matching to input. In the signature matching context DFAs have two major advantages: matching to input requires only a single table lookup per input byte, and it is possible to compose the DFAs corresponding to multiple signatures into a combined DFA that recognizes all signatures in a single pass over the input.

Unfortunately, regular expressions common to intrusion detection interact poorly when their DFAs are combined, yielding a composite DFA that is typically much larger than the sum of the sizes of the DFAs for individual signatures and often significantly exceeds available memory. On the other hand, approaches that use Nondeterministic Finite Automata (NFAs) have modest memory requirements but are too slow for high-speed signature matching. Thus, DFAs and NFAs introduce a space-time trade-off between memory usage and performance. Many techniques have been proposed in the literature for compressing either the states or the transitions. Often, these techniques require hardware support, specialized architectures, or more complex matching procedures.

In this paper, we propose a lightweight transition compression technique for reducing the memory requirements of combined DFAs. We start from the observation that for IPS signatures, distinct input symbols often have identical behavior in their DFAs. In these cases, an *Alphabet Compression Table (ACT)* [11] can be used to map such groups of symbols to a single symbol that is retrieved by a table

lookup. Alphabet compression tables were first proposed for use in compiler-writing tools such as YACC [2, 11] and have been recently explored in the signature matching context as well [4]. In this work we refine this technique by introducing *multiple* alphabet compression tables. Specifically, we develop heuristics for partitioning the set of states in a DFA and creating compression tables for each subset in a way that yields further reductions in memory usage.

Using compression tables does require more processing time, since the per-byte cost now includes lookups into these tables. However, our experiments using real-world signature sets show that once the overhead of the first compression table has been paid for, inclusion of additional compression tables is essentially free. Further, although alphabet compression tables are not always the fastest and do not always have the smallest memory footprints, when considering both run-time and memory usage requirements simultaneously they consistently yield the best trade-off when compared to other common transition compression techniques. In summary, this paper makes the following contributions:

- we introduce the use of *multiple* alphabet compression tables to reduce the memory footprint of combined DFAs by a factor of up to 70 in the best case, without the need for custom hardware assistance;
- we present efficient heuristics for constructing multiple alphabet compression tables;
- we perform a comprehensive evaluation comparing our technique with another recent proposal, D^{2}FAs [13], using recent real-world signature sets from two popular IPSes, Snort and Cisco IPS.

The rest of this paper is structured as follows. After the background and related work in Section 2, Section 3 defines alphabet compression tables and gives algorithms for their construction. Section 4 describes another technique, D^{2}FAs, used extensively in our evaluation. In Section 5, we present our experimental results. Finally, Section 6 concludes.

2. BACKGROUND AND RELATED WORK

Early signature matching techniques were string-based and built upon multi-pattern string matching algorithms [1,9,21] to efficiently match multiple signatures against the payload. In particular, the Aho-Corasick algorithm [1] constructs an automaton-like structure that simultaneously matches all strings in a single pass over the input.

More recently, regular expressions have become the *de facto* standard for expressing signatures, in large part because they are strictly more expressive than strings. Like strings, though, they have representations and matching procedures that can match multiple expressions simultaneously. Indeed, Sommer and Paxson [19] argue that the increased expressivity of regular expressions combined with their efficient matching procedures yields fundamental improvements in signature matching capabilities. Unfortunately, common representations such as DFAs and NFAs reside at opposing extremes from a memory-performance perspective. When combined, DFAs are fast but too large whereas NFAs are small but too slow.

Many techniques have been proposed in the literature recently to reduce the memory usage of combined automata while maintaining acceptable matching performance. These techniques typically fall into one of two categories: state compression, which reduces the number of states, and transition compression, which compresses the footprint of a single state. State compression has the potential for larger memory savings since total memory is a function of the number of states. However, in principle, state and transition compression techniques are orthogonal so that in many cases both can be applied simultaneously.

Yu *et al.* proposed *DFA Set-Splitting* [22], a state reduction technique that combines signatures into multiple DFAs (instead of a single DFA) such that total memory usage is below a supplied threshold. Regular expressions are heuristically selected for combination until the resulting automaton exceeds its "fair share" of memory, at which time it becomes immutable and a new combined automaton is begun. The process repeats until all expressions have been included. Set-splitting traces a curve between NFAs and DFAs in space-time: as the memory threshold is increased, the number of DFAs that must be simultaneously matched shrinks, decreasing the inspection time.

Several techniques have been proposed recently to reduce the number of states in a combined automaton. Smith *et al.* [17,18] introduced *Extended Finite Automata (XFAs)*, which augment DFAs with auxiliary variables such as bits and counters that can track computation state more compactly than explicit DFA states alone can do. Simple instructions are attached to states and/or edges for manipulating these variables and testing their value. By carefully incorporating auxiliary variables, DFAs are transformed so that structurally they look like multi-pattern string-matching automata and therefore do not blow up when combined. XFAs perform state compression and are thus orthogonal to the transition compression technique developed in this work. Kumar *et al.* [12] have also performed work incorporating bits and counters. Alternatively, Becchi and Cadambi [3] have proposed *State Merging*, which pushes information from states themselves into labeled transitions, thereby allowing states to be combined together.

With regard to transition compression, Kumar *et al.* [13] introduced *Delayed Input DFAs* (D^{2}FAs), based on the observation that many states had similar (or identical) transition tables. In their approach, each transition table retains only the transitions that are distinct to the corresponding state. Transitions common to many states are stored in a single transition table linked to via chains of *default* transitions that consume no input and eventually lead to the proper state. This technique is orthogonal to ours. We perform extensive comparisons to D^{2}FAs in this work and describe them in more detail in Section 4. Further work [4, 14] extends this technique to eliminate hardware dependencies and reduce default transition traversals. Recently, Becchi and Crowley [4] have also employed alphabet compression tables to reduce memory requirements, although they limit compression to a single compression table.

Hardware-based solutions can parallelize the processing required to achieve high performance by processing many signatures in parallel rather than explicitly combining them. Sidhu and Prasanna [16] provide a hardware-based NFA architecture that updates the set of states in parallel during matching. Sourdis and Pnevmatikatos [20] employ content-addressable memories (CAMs) to increase the performance further, and Clark and Schimmel [8] present optimization techniques (such as examining multiple bytes per clock cycle) and achieve regular expression matching at rates of up to 25 Gbps. Brodie *et al.* [5] also employ multi-byte transi-

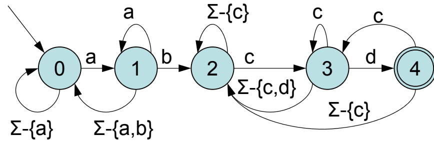


Figure 1: A DFA recognizing the regular expression (.*)ab(.*)cd. Starting in state 0, the expression is accepted whenever state 4 is reached.

tions and apply compression techniques to reduce the memory requirements. These techniques show promise for high performance matching. However, replication of NFAs introduces scalability issues as resource limits are reached (Clark and Schimmel are able to fit only 1500 signatures on their prototype). In addition, hardware techniques in general lack the flexibility for evolving signature sets that is implicit to intrusion detection, and they restrict applicability to those instances where the hardware cost can be justified and custom hardware support is available.

Finally, in another context, the compiler construction community has also investigated compression techniques for regular expressions. Dencker *et al.* [10] describe a variety of techniques for compressing parse tables. Johnson [11] introduced the use of default transitions for DFA compression during development of the YACC parser generator. Instead of hashing, Johnson used two auxiliary arrays and performed careful placement of state identifiers to achieve compression. Similar techniques to these are currently used by the Flex Scanner Generator [15]. Our experiments with this technique using Flex have yielded memory reductions of up to 50×, although the reduction comes at a high execution time cost involving multiple memory accesses and default transition lookups that are not suitable for high-speed matching. Furthermore, tools like Flex perform matching using repeated invocations of the DFA and assume different semantics. Thus, they are not directly applicable without modification.

3. ALPHABET COMPRESSION TABLES

A DFA is a directed graph with labeled edges used for efficiently matching regular expressions to input. Nodes are termed *states*, edges between nodes are called *transitions*, and each edge is labeled with a symbol from the input alphabet Σ. For each state S in the DFA, there is an edge for each input symbol in Σ from S to some state S' in the DFA. The set of transitions out of S is referred to as the *transition table* for S, and each state has its own table. A non-empty subset of the states are marked as *accepting*, and there is a distinct starting state s_0. Figure 1 shows a DFA that recognizes the regular expression `(.*)ab(.*)cd` (read as: an arbitrary number of characters, followed by `ab`, followed by an arbitrary number of characters, followed by `cd`). The start state is state 0, and the corresponding regular expression is recognized when state 4 is reached. In general, for each regular expression R, there is a DFA D such that D accepts exactly the language described by R.

The DFA matching procedure keeps a *current state* variable that is initialized to state s_0. During matching, the DFA reads input characters one at a time and updates the current state by following the appropriate transition out of

```
SingleAlphabetPartition(StateSet States):
1  CrtBestPartition = {Σ}
2  foreach state s ∈ States do
3  |  NextPartition = {}
4  |  foreach character group g ∈ CrtBestPartition do
5  |  |  NextToChars = EmptyHashTable
6  |  |  foreach character c ∈ g do
7  |  |  |  NextToChars[s.next[c]] ∪= {c}
8  |  |  foreach state n ∈ NextToChars.keys() do
9  |  |  |  NextPartition ∪= {NextToChars[n]}
10 |  CrtBestPartition = NextPartition
11 return CrtBestPartition
```

Algorithm 1. Compression algorithm that finds the partition of the input alphabet Σ with the smallest number of equivalence classes.

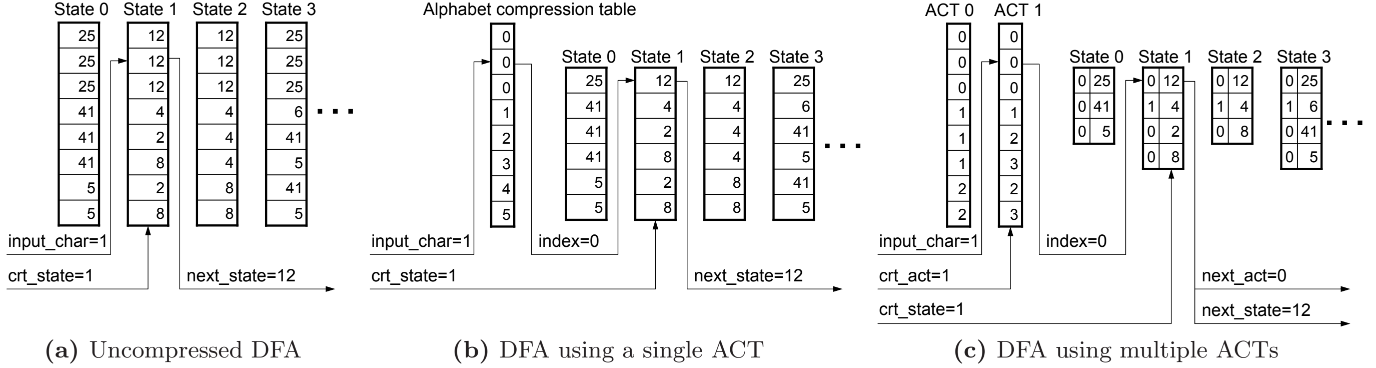
the current state to the destination state. Reaching an *accepting* state indicates that the input thus far is a string in the language defined by the regular expression.[1] Figure 2a depicts this procedure at a specific state.

Alphabet compression tables for DFAs arise from the observation that for any given transition table, there are often many input characters that lead to the same next state. Such identical behavior forms a binary relation between input symbols and partitions them into equivalence classes. Individual transition tables can then store a single entry for an entire equivalence class, and a shared lookup table can be used to map from the observed input character to the appropriate equivalence class entry in the compressed transition table (Figure 2b). Since this alphabet compression table (ACT) is shared by all states, it will be accessed for every input character, and thus likely reside in the cache of the processor. Therefore, while alphabet compression adds one extra lookup to the per-byte processing, it does not have a significant negative performance impact as there is no need for an extra off-chip memory access.

Before discussing the algorithm for building alphabet compression tables, we clarify some of the notation used in the algorithms in this paper. Our notation relies heavily on the use of sets whose elements can be characters, states, or other sets (with all elements of a set being of the same type). We use the standard definition for set equality: $\{1,2\} = \{2,1\}$, but $\{\{1,2\},\{3,4\}\} \neq \{1,2,3,4\}$ (actually two such sets would never even get compared by our algorithms since their elements are of different types). As usual, the size of a set S given by $|S|$ only counts the number of elements in the top-level set, and does not give a recursive count of all atomic elements. For sets A and B, the statement $A \cup= B$ is shorthand for $A = A \cup B$. Finally, we represent hash tables as associative arrays and use standard notations (*e.g.* *hashtable*[*key*]) for performing lookups, using sets both as keys and values in some cases. To simplify the algorithms, we introduce the convention that for hash tables whose values are sets, looking up a non-existent value returns the empty set rather than explicitly signaling failure.

We say that a state *distinguishes between* two characters if the transitions corresponding to those characters go to different states. Thus in Figure 1, characters `b` and `d` are dis-

[1] In the more traditional definition, a DFA signals a match only if it is in an accepting state after reading the last input character. All the results we present apply to that definition as well.



(a) Uncompressed DFA **(b)** DFA using a single ACT **(c)** DFA using multiple ACTs

Figure 2: The core operation of DFA matching is to look up the next state based on the current state and the input character. By using one or more alphabet compression tables, we can reduce the size of the transition tables attached to individual states.

tinguished between by each of states 1 and 3, but not by 0, 2 and 4. On the other hand, characters `e` and `f` are not distinguished between by any state. When using a single compression table there is a unique partition of the symbols in the input alphabet that minimizes the total memory usage. Algorithm 1 gives the procedure that computes this partition in a single traversal of the states of the DFA. Starting with a partition of size one whose single entry is the full set of input characters, the algorithm progressively refines the partition to account for distinctions between input characters that manifest themselves as transitions to distinct states out of the same source state. Upon completion, the algorithm finds the fewest number of sets σ of input symbols where all the elements in each set induce the same sequence of traversed states in the automata. Per-state transition tables are correspondingly reduced from $|\Sigma|$ to σ entries. Conversely, for any two characters that are in different sets, there is at least one state that has transitions to different states for these two characters. Given the output of Algorithm 1, building the actual alphabet compression table and the compressed transition tables is straightforward. Note that the complexity of this algorithm is $O(n|\Sigma|)$ where n is the number of states and $|\Sigma|$ is the size of the input alphabet.

3.1 Multiple alphabet compression tables

It is often the case that many characters behave identically for a large fraction of states $\mathcal{S}$ but are individually distinguished between by a small (perhaps overlapping) set of states. When using a single ACT for all states as in the previous section, individual characters of such groups will need separate entries in each of the compressed transition tables, limiting the memory savings that can be achieved. If instead we compute an ACT to apply only to the large subset of states $\mathcal{S}$, the transition tables are smaller since the groups of characters treated identically are larger and fewer. Thus, further reductions in memory usage can be obtained by using multiple ACTs, each over a disjoint subset of states.

To build a DFA with m ACTs, we first divide the states of an automaton into m subsets (discussed below) and then compute a separate ACT for each subset. During matching, the lookup function needs not only the current state and current input symbol but also the identity of the correct ACT to use (in the range $\{1..m\}$). Thus, in the transition table we don't just encode the next state but also the corresponding ACT. Figure 2c shows the matching process extended for multiple ACTs. Since the number of ACTs is small (up to 8 in our experiments), for all currently feasible configurations a 32-bit word can encode both the ACT number and the pointer to the next state so that sizes of transition table entries are not increased. Since entries of the transition tables are decoded efficiently and all the ACTs are typically cached, the matching process is not significantly slower than in the case of a single ACT.

BestStatePartition(StateSet *States*):

1 $AlphaPartToStates$ = EmptyHashMap
2 **foreach** state $s \in States$ **do**
3 $\quad Groups$ = SingleAlphabetPartition ($\{s\}$) `// Alg. 1`
4 $\quad AlphaPartToStates[Groups] \cup= \{s\}$
5 $Result = \{\}$
6 **foreach** partition $ap \in AlphaPartToStates$.keys() **do**
7 $\quad Result \cup= AlphaPartToStates[ap]$
8 **return** $Result$

Algorithm 2. Algorithm to partition a set of states so that the sum of the sizes of transition tables is minimized when the number of ACTs is unlimited.

In constructing multiple alphabet compression tables, we must first divide the states into subsets that will be covered by the same ACT. For any of these subsets, we can then use Algorithm 1 to build the corresponding ACT. If there are no restrictions on the number of ACTs we can use, the partition that minimizes the total size of the transition tables is the one in which all states that distinguish between the same input symbols use the same ACT. Algorithm 2 finds this best partition of the set of states in $O(n|\Sigma|)$ time. Unfortunately, for practical automata, the number of ACTs required to achieve the optimum is unfeasibly large, so we need algorithms that can guarantee that the number of state subsets produced by the partition of the states is bounded above by a given m.

There are $S(n, m)$ ways to partition n elements into m disjoint subsets, where $S(n, m)$ is a Stirling number of the second kind [6], given as:

$$S(n,m) = \frac{1}{m!}\sum_{i=0}^{m}(-1)^i \binom{m}{i}(m-i)^n$$

```
FastPartitionStates(StateSet States, Int m):
1  CrtPartition = {}
2  RemainingStates = States
3  for i = 1 to m − 1 do
4     StatesCovered = RemainingStates
5     TargetSize = |RemainingStates|/2
6     Groups=SingleAlphabetPartition (StatesCovered)
7     while |StatesCovered| > TargetSize do
         /* Choose the pair of groups distinguished between by
            the fewest states, discard the states, combine the
            groups.                                          */
8        StatesCut = StatesCovered
9        for j = 0 to |Groups| − 1 do
10          for k = j + 1 to |Groups| − 1 do
11             Candidates = {}
12             foreach state s ∈ StatesCovered do
13                if s.next[Groups[j][0]] ≠ s.next[Groups[k][0]]
                  then
14                   Candidates ∪= {s}
15             if |Candidates| < |StatesCut| then
16                StatesCut = Candidates
17                bestj = j
18                bestk = k
19       if |StatesCut| == |StatesCovered| then
20          return CrtPartition ∪ {RemainingStates}
21       NewGroup = Groups[bestj] ∪ Groups[bestk]
22       Groups = Groups ∪ {NewGroup} -
         {Groups[bestj], Groups[bestk]}
23       StatesCovered = StatesCovered - StatesCut
24    CrtPartition ∪= {StatesCovered}
25    RemainingStates = RemainingStates - StatesCovered
26 return CrtPartition ∪ {RemainingStates}
```

Algorithm 3. Fast heuristic algorithm for partitioning a set of states into m subsets such that when ACTs are computed separately for each subset, total memory usage is low.

Note that $S(n, m)$ is bounded above by $m^n/m!$. We found no criterion for easily determining the optimal partition and instead focused on heuristic techniques. We evaluated two methods for partitioning the states into m subsets. First, a "bottom-up" approach starts with the partition produced by Algorithm 2 and combine subsets until the total number of subsets is reduced to m. The combination routine iteratively combines subsets two at a time, selecting at each iteration the two subsets that yield the smallest increase in total memory usage. Unfortunately, this algorithm operates in $O(n^3|\Sigma|)$ time. We found that for large rule sets typical to those found in signature matching, this algorithm was unacceptably slow (each run required over a day to complete) with results no better than those from the top-down approach described below. Thus, we do not consider it further in this paper.

An alternative method to producing a partition with m subsets, and the one we employ, is to use a "top-down" approach that starts with a single set and iteratively subdivides until m subsets are produced. In Algorithm 3, we present such a top-down approach that completes in only $O(mn|\Sigma|^3)$ time. At each step, the algorithm sets a target for the size of the subset to remove (line 5: we found that setting this target to half the remaining states works well) and finds a subset that is large enough and has a small ACT. To do so it uses a greedy heuristic (lines 7-23) that starts with the set

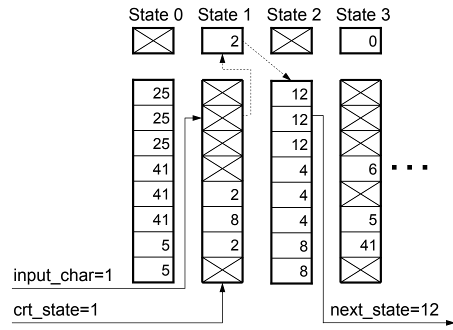


Figure 3: For each state, D^{2}FAs employ a default transition that is followed whenever its transition table contains no entry for the current symbol.

of all remaining states and removes states from the set until the desired size is reached.

The greedy heuristic implemented by the loop between lines 7 and 23 tries to find a large set of states with an ACT that results in small transition tables. Each iteration of the loop reduces the size of the transition table by one by removing all states that distinguish between two groups of characters. To remove the fewest possible states, the nested inner loops (lines 9 to 18) go through all pairs of groups of characters and pick the two groups that can be combined by removing the fewest states. Note that if at each step of the outermost loop we chose the smallest subset larger than the target size (as opposed to the largest below it), the complexity of the algorithm would reduce to $O(n|\Sigma|^3)$. In practice the difference between the sizes of the two sets is not significant, and the actual running times do not depend much on m since as m increases the loop in lines 7 to 23 works with exponentially smaller sets of states and the processing requirements are dominated by the cost of the first few iterations through the outermost loop.

4. D^{2}FAs

Kumar *et al.* proposed Delayed Input DFAs (D^{2}FAs) [13] as another solution for compressing the transition tables used by DFAs. Whereas alphabet compression exploits the fact that for a given state the transitions for many input characters point to the same next state, D^{2}FAs build on the fact that for a given input character, many states transition to the same next state. Thus, if two states have the same transitions for many characters, one can reduce memory by storing for one of the states only the transitions that differ. Default transitions that consume no input link states with elided transitions to states that contain the proper transition table entry. As shown in Figure 3, if the transition table entry for the input symbol is not stored in the current state, the default transition points to the state whose transition table should be consulted. Multiple states can have default transitions pointing to the same state, and one may need to follow multiple default transitions when processing a single input character. Following chains of default transitions comes at a processing cost, so the maximum length of default transition is given and fixed during construction. Kumar *et al.* show that D^{2}FAs lead to large reductions in memory usage, but there are two limitations to consider.

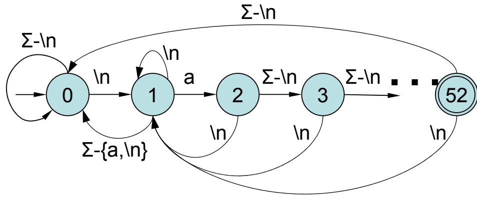


Figure 4: A DFA recognizing the signature `.*\na[^\n]{50}` .

First, memory savings achieved by D^{2}FAs can vary widely among different kinds of signatures. Figure 4 shows the signature `.*\na[^\n]{50}` (read as an arbitrary number of characters followed by a newline and an `a` followed by 50 non-newline characters) for which D^{2}FAs cannot achieve significant memory savings, but ACTs can. Such signatures are commonly used to detect buffer overflow attacks. States 2 to 52 have very similar transition tables: for the newline character the next state is 1, and for all others the next state is the state with the next number. Applying an ACT for these states can reduce the size of their transition tables to 2, but D^{2}FAs cannot produce significant memory reductions since most of transitions are to distinct next states.

Second, software implementations of D^{2}FAs can be slow. The original D^{2}FA proposal is targeted to custom hardware environments where content addressable memories can be used. Software implementations must use a hash table-like data structure to compress transition tables, but without careful design this can result in unacceptable run-time and memory overheads resulting from computing hash functions and handling collisions.

To adapt to software-based environments, we designed a solution that combines a bitmap and an array to achieve good performance and low memory overhead. Each state has a bitmap as large as the alphabet (256 bits or 8 words) to indicate whether the transition corresponding to a given input character is stored or not, and an array to store the actual transitions. To determine the position of the transition in this array during matching, we need to count the number of bits set to 1 in the bitmap prior to the position of the bit corresponding to the input character. For our signature sets this solution uses between 0.1% and 148% more memory compared to an idealized solution that has no memory overhead. Compared to an idealized solution that performs array lookups instead of hashed lookups, the runtime is between 2.6 and 6 times larger.

4.1 Combining D^{2}FAs and ACTs

Since ACTs and D^{2}FAs exploit orthogonal properties of DFA transitions, it is natural to ask whether it is possible to combine them into a solution that combines the strengths of both approaches. We evaluated a straightforward hybrid of the two methods that applies alphabet compression to D^{2}FAs. We extend the input alphabet to include a "not handled here" symbol, and we increase the size of the alphabet compression table by one to also include the default transition. With these extensions, we can directly apply our procedures for building the alphabet compression tables to D^{2}FA-compressed automata like that shown in Figure 3. As in Figure 2c, the entries in the transition table indicate the next state and the ACT to use. Also as with D^{2}FAs, the algorithm may need to follow multiple default transitions when processing an input character. Our experiments show that for some signature sets this hybrid solution results in the most compact automata.

Rule Set	# of ACTs	Memory (KB)	Exec Time cycles/byte	Trans. per state
Cisco SMTP	0	630,669	46.3	256
	1	231,573	50.6	94
	8	165,234	48.8	67
Cisco HTTP	0	106,771	43.0	256
	1	81,329	54.7	195
	8	24,124	52.0	57
Snort SMTP	0	810,711	22.2	256
	1	139,340	30.0	44
	8	67,761	29.8	21
Snort HTTP	0	163,114	38.6	256
	1	36,955	46.1	58
	8	15,150	43.9	23

Table 1: Measuring the cost of multiple compression tables. The biggest reductions come after the first table is employed, but additional tables yield further memory reductions.

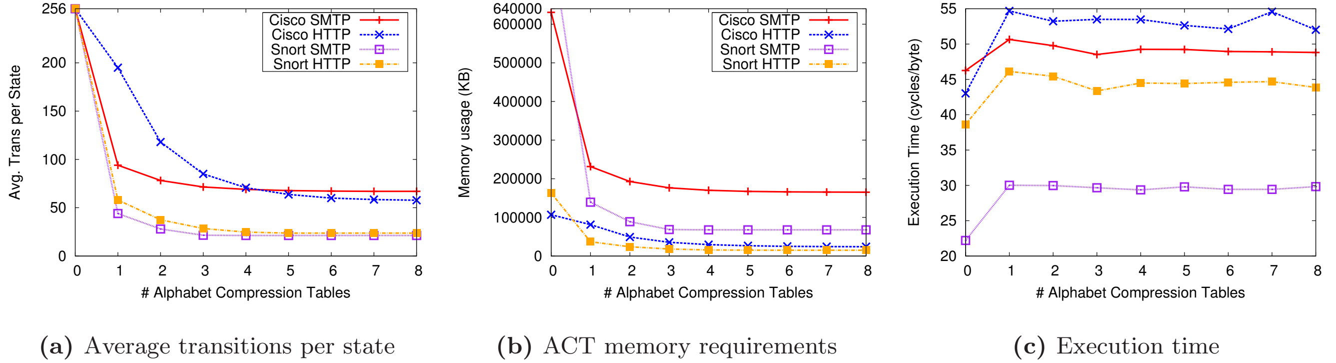
5. EXPERIMENTAL RESULTS

We performed a comparative evaluation using multiple signature sets to better understand the behavior of ACTs in practice. We extracted regular expressions from the FTP, HTTP, and SMTP signatures from the Snort and Cisco IPS rule sets and grouped them by protocol, collecting 1550 regular expressions in total. In addition to the algorithms and techniques described in this paper, we also implemented the DFA Set Splitting algorithm [22] (termed *mDFA* here, for "Multiple DFA") for combining a set of signatures to a group of DFAs. Finally, our comparative evaluation of D^{2}FAs was performed using the D^{2}FA source code obtained from its authors. Modifications discussed in Section 4 were built upon this as well. Test results involving execution time were obtained using a 10GB trace collected on the edge of a university departmental network. All experiments were performed on a Linux workstation with a 3.0 GHz Pentium IV processor and 3.4 GB of memory that was otherwise idle. We used cycle-accurate performance counters to measure the number of cycles required by the matching operations.

5.1 Multiple alphabet compression tables

The first set of experiments looks at the behavior of ACTs as the number of compression tables is increased. For each of our rule sets, we combined a subset of the regular expressions and converted them to a large, single DFA. We then repeatedly invoked Algorithm 3 with values of m (the number of alphabet compression tables) increasing from 0 to 8. Table 1 presents the memory requirements, execution time, and average transitions per state for 0, 1, and 8 compression tables. Figure 5 presents the results graphically for all tested values of m. For clarity of presentation, we show detailed results for only four of the six data sets. The omitted data sets have similar behavior.

The case $m = 0$ is the combined DFA without any alphabet compression applied and serves as the baseline for comparison. Consequently, the number of transitions per state is 256, the size of the alphabet. As m is increased,



(a) Average transitions per state **(b)** ACT memory requirements **(c)** Execution time

Figure 5: Effect of multiple ACTs on memory usage and matching execution time. Incorporating ACTs induces an initial runtime cost, but subsequent increases in the number of tables is free.

the memory requirements (also counting the memory used by the compression tables themselves) decreases. With 8 tables, the Cisco signature sets exhibit approximately a 4× reduction in memory usage, whereas for the Snort signature sets a 12× to 15× reduction is observed. As Figures 5a and 5b show, the memory usage experiences the largest decreases after the first alphabet compression table is applied, but using multiple ACTs reduces memory requirements further.

ACTs do carry an increased execution cost, adding 5 to 10 cycles per byte to the execution time on average. Fortunately, in Figure 5c we see that this cost is incurred only when the first alphabet compression table is introduced; adding multiple ACTs does not incur significant additional run-time costs. Thus, even though we observe diminishing returns in memory savings as the number of ACTs increases, the increased savings come for free, essentially, after the initial cost of including compression tables has been paid. For the remainder of the experiments, we use $m = 8$ ACTs.

5.2 ACTs, D²FAs and uncompressed DFAs

Next, we compare ACTs to D²FAs, D²FAs + ACTs, and uncompressed DFAs. Combining all regular expressions into a single DFA exceeds feasible memory limits, so we used set splitting [22] to produce sets of combined DFAs that cover all the rules. For the construction, we supplied memory budgets ranging from 4 MB to 128 MB.[2] As shown in columns 2 and 3 of Table 2, smaller memory budgets result in large numbers of DFAs to match. We use the term *protocol set* to refer to the set of DFAs produced by the algorithm for a given protocol and a given total memory setting. We then built a distinct set of eight alphabet compression tables for each protocol set. Thus, for example, a rule set such as Snort SMTP combined into six DFAs would contain eight ACTs that are shared among the six DFAs. Finally, we repeated the construction process to produce D²FAs for each of the DFAs in the protocol sets.

We performed signature matching using protocol sets with uncompressed DFAs, DFAs with ACTs, D²FAs, and D²FAs with ACTs, recording execution time and memory usage. Table 2 shows the results for three memory settings: 16 MB, 48 MB, and 128 MB. Execution times are higher in these results principally because we must repeat the matching procedure for each DFA in a protocol set. Note also that in some cases (Cisco SMTP), increasing the amount of available memory does not decrease execution time. This behavior is an artifact of the greedy algorithm [22] for building the protocol sets. In general, the table shows that increasing total available memory reduces the number of DFAs in the protocol set, decreasing execution time.

Compared to uncompressed DFAs (column 4 in Table 2), the table shows a sharp reduction in memory costs when eight ACTs are employed (column 5). For 16 MB total memory, ACTs are between 66× smaller (Snort SMTP) and 4× smaller. At 128 MB, DFAs with ACTs are between 19× and 4× smaller. As expected, however, there is a slight increase in execution time: execution times with ACTs are typically between 35% to 85% slower, the largest slowdown approaches a factor of 3×. Figure 6 (Cisco rules) and Figure 7 (Snort rules) show the memory usage (top graph) and execution time (bottom graph) for all supplied values of available memory for three signature sets.

D²FAs (column 6) exhibit wider variability in their performance and memory usage than ACTs. For Cisco rule sets, our tests give an 11× to 17× reduction in memory usage. These results are generally consistent with those reported in [13]. For the Snort signature sets, however, which were not included in the original D²FA evaluation, the memory reduction is always less than a factor of 8 and often less than a factor of 2. This is consistent with our observation that D²FAs are designed to optimize DFAs in which certain symbols in the alphabet (almost) always go to the same state. This is not characteristic of the Snort sets, and thus there is little opportunity for compression.

The hybrid algorithm that combines D²FAs and ACTs (rightmost column in Table 2) always achieves low memory (often the lowest of all solutions), and run-times that are close to, but larger than those of ACTs. ACTs are faster because the matching algorithm does not need to follow default transitions. Interestingly, in one of the signature sets D²FAs use less memory than the hybrid approach. The reason is that after applying ACTs to D²FAs, for a given state there may be multiple entries in the actual transition table storing the "not handled here" symbol, resulting in higher memory usage than D²FAs that do not store these entries.

[2]Although 128 MB may seem rather small in relation to modern memory capacities, our tests are performed using a single protocol. In reality, DFAs for many protocols must reside in memory simultaneously.

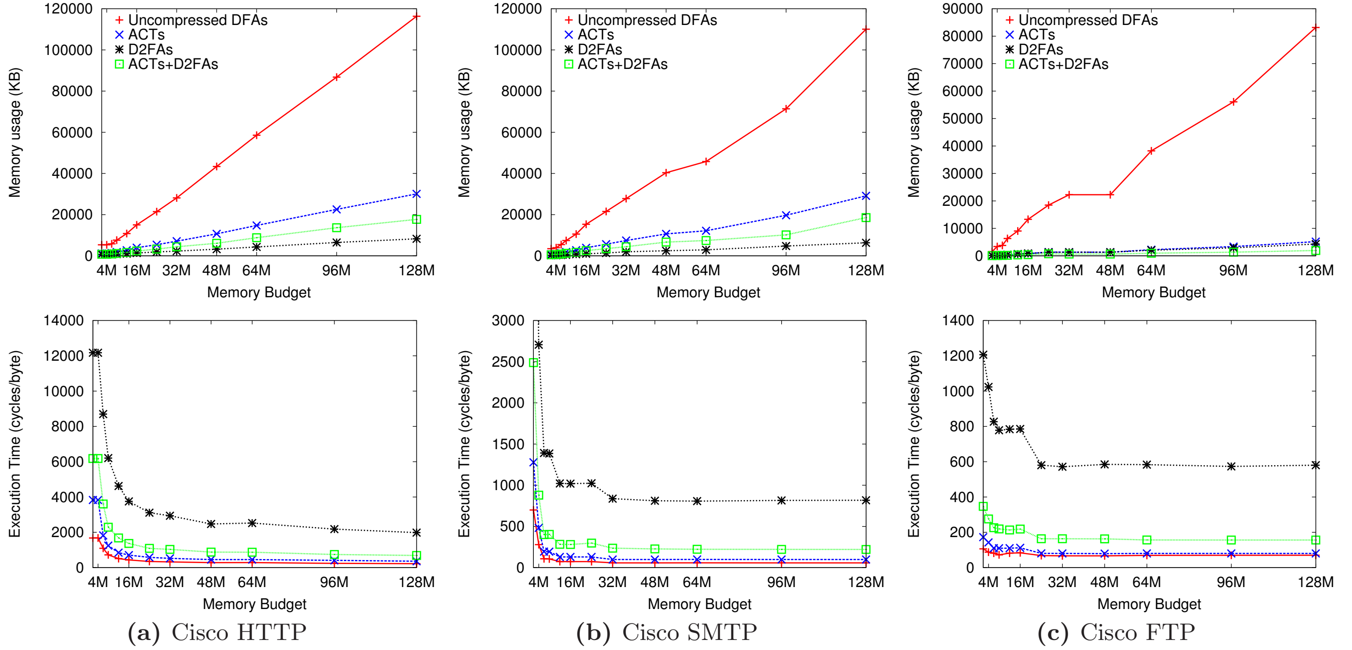


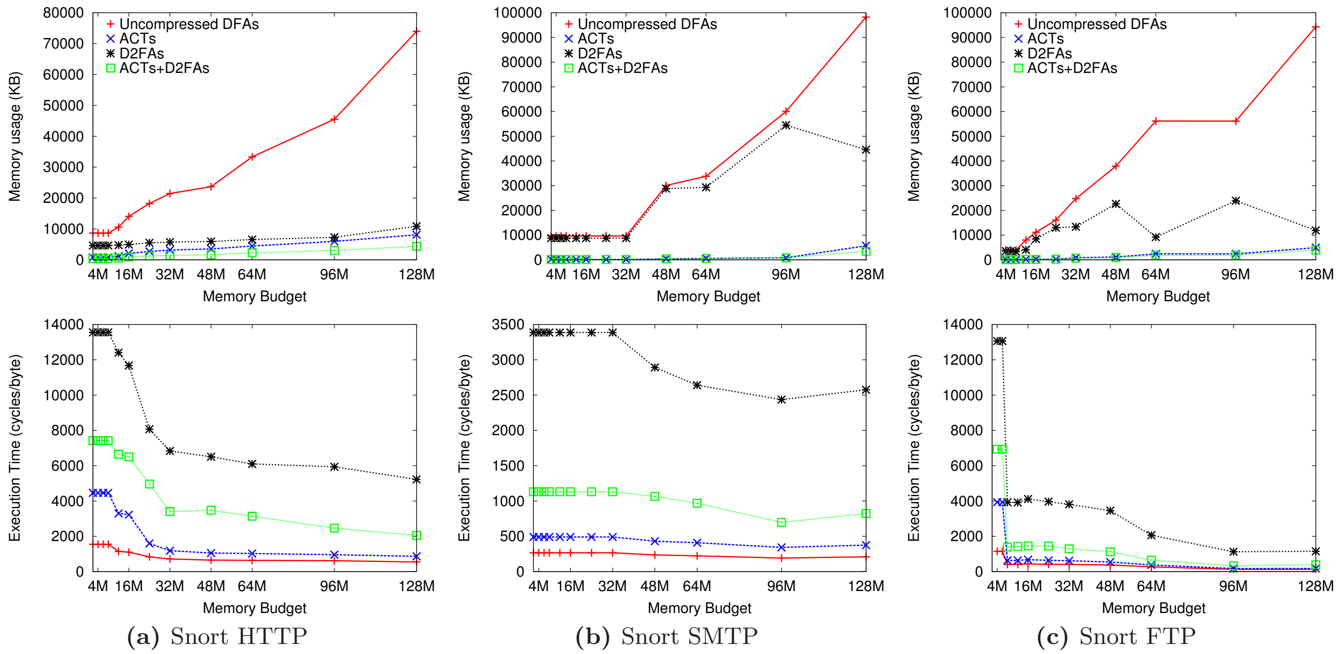
Figure 6: Comparing memory (top) and performance (bottom) of ACTs to mDFAs and D2FAs with Cisco rule sets.

Signature set	Memory budget (MB)	Number of DFAs	Uncompressed Runtime (cycles/byte)	Uncompressed Memory (KB)	multiple ACT Runtime increase	multiple ACT Memory decrease	D^2FA Runtime increase	D^2FA Memory decrease	mult. ACT+D^2FA Runtime increase	mult. ACT+D^2FA Memory decrease
Snort SMTP	16	15	266	9,667	1.84×	65.96×	12.70×	1.10×	4.24×	75.43×
	48	13	236	30,058	1.82×	70.66×	12.25×	1.04×	4.51×	73.66×
	128	11	209	98,236	1.79×	17.26×	12.31×	2.20×	3.94×	28.52×
Snort HTTP	16	45	1,103	14,065	2.92×	6.74×	10.58×	2.83×	5.90×	15.17×
	48	28	651	23,693	1.62×	6.71×	9.98×	4.00×	5.33×	14.44×
	128	23	543	73,988	1.59×	9.11×	9.60×	6.81×	3.78×	16.88×
Snort FTP	16	18	434	11,127	1.56×	50.50×	9.47×	1.32×	3.36×	62.10×
	48	14	374	37,920	1.45×	33.28×	9.23×	1.67×	3.02×	40.99×
	128	4	131	94,288	1.35×	19.13×	8.76×	7.92×	2.97×	23.71×
Cisco SMTP	16	4	72	15,316	1.78×	3.92×	13.98×	15.04×	3.85×	6.03×
	48	3	57	40,367	1.72×	3.79×	14.22×	16.38×	3.98×	6.05×
	128	3	57	110,063	1.72×	3.78×	14.34×	17.34×	3.86×	5.92×
Cisco HTTP	16	19	432	15,015	1.64×	3.81×	8.66×	11.03×	3.16×	6.42×
	48	12	282	43,389	1.62×	4.06×	8.76×	13.37×	3.12×	7.12×
	128	9	220	116,352	1.64×	3.87×	8.98×	14.08×	3.11×	6.57×
Cisco FTP	16	3	83	13,308	1.34×	16.41×	9.38×	15.43×	2.61×	31.56×
	48	2	66	22,254	1.19×	16.97×	8.76×	16.92×	2.44×	33.95×
	128	2	70	83,162	1.14×	16.09×	8.26×	19.29×	2.23×	42.86×

Table 2: Comparison of run times and memory usage for uncompressed DFAs, DFAs using multiple ACTs, D^2FAs, and D^2FAs using multiple ACTs.

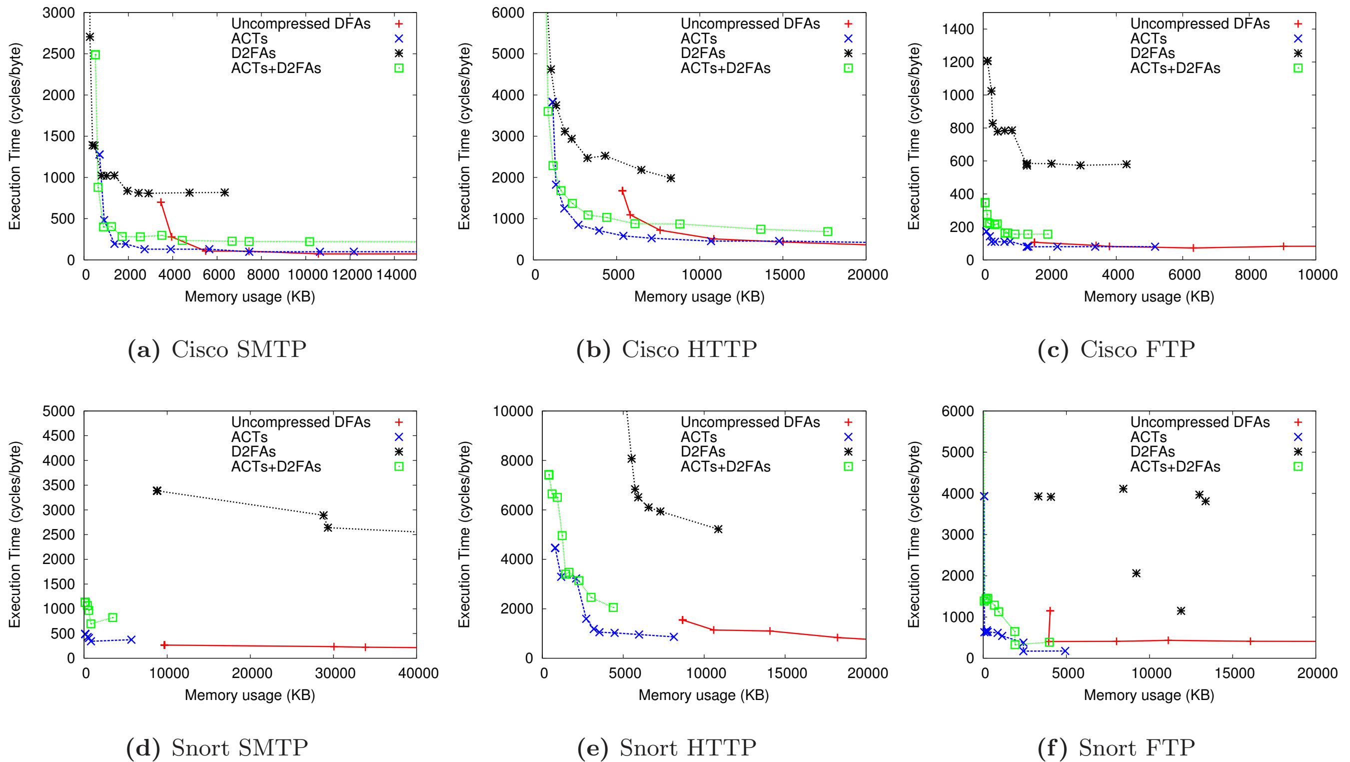
Both execution time and memory usage are critical resources in signature matching and induce a space-time tradeoff. Figure 8 depicts a space-time comparison for all six of our test sets, directly showing the trade-offs that occur between memory usage (the x-axis) and execution time (the y-axis). We have truncated the axes in some sets to more clearly highlight the data in the lower left-hand quadrant; this does not influence the interpretation. Each point on the plot refers to an observed total available memory setting. Data points belonging to the same compression technique trace out a curve that shows the trade-offs between execution time and memory for that technique. In the limit, large memory yields fast execution, and small memory requires large execution times. Entries toward the origin (the bottom left corner) require reduced resources in space and time and are thus preferred.

Most importantly, for all protocol sets ACTs provide the most favorable trade-offs between run time and memory usage. Admittedly, it may be surprising that ACTs can be faster than uncompressed DFAs despite the overhead of the compression table mapping. In reality, large available memory sizes (resulting in bigger but fewer DFAs) combined with excellent ACT memory reduction yields a memory footprint that is smaller than for uncompressed DFAs, and the time



(a) Snort HTTP (b) Snort SMTP (c) Snort FTP

Figure 7: Comparing memory (top) and performance (bottom) of ACTs to mDFAs and D2FAs with Snort rule sets.



(a) Cisco SMTP (b) Cisco HTTP (c) Cisco FTP

(d) Snort SMTP (e) Snort HTTP (f) Snort FTP

Figure 8: Comparing memory usage vs. run-time performance on several techniques. In all cases, multiple ACTs yield the best trade-offs between memory usage and run-time performance.

savings obtained from executing fewer DFAs more than compensates for the ACT overhead. Thus, a small number of highly compressed DFAs can be both smaller and faster than other alternatives.

6. CONCLUSION

In this paper we introduced multiple alphabet compression tables (ACTs) for reducing the memory footprint of DFA-based signature matching. This technique uses heuristics to partition the states of a DFA, computing a distinct ACT for each partition. Multiple ACTs achieves increased memory reduction over single ACTs with no additional runtime cost. We present algorithms for constructing multiple ACTs and demonstrate their effectiveness using signatures found in Cisco IPS and Snort. ACTs are applicable in software-only environments, although they may be easily included in hardware-based solutions.

Compared to uncompressed DFAs, multiple ACTs achieve memory savings of between a factor of 4 and a factor of 70 at the cost of an increase in run time that is typically between 35% and 85%. Compared to D^2FAs, multiple ACTs are between 2 and 3.5 times faster in software, and for some signature sets they use less than one tenth of the memory. Overall, for all signature sets and compression methods evaluated, ACTs offer the best memory versus run-time tradeoffs.

Acknowledgments

This work is sponsored by NSF grants 0546585 and 0716538 and by a gift from the Cisco University Research Program Fund at Silicon Valley Community Foundation. We thank the anonymous reviewers for suggestions that improved this paper.

7. REFERENCES

[1] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. In *Communications of the ACM*, June 1975.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[3] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *Proceedings of IEEE Infocom*, Anchorage, AK, May 2007. ACM.

[4] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 2007 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Orlando, FL, December 2007. ACM.

[5] Benjamin C. Brodie, Ron K. Cytron, and David E. Taylor. A scalable architecture for high-throughput regular-expression pattern matching. *SIGARCH Comput. Archit. News*, 34(2):191–202, 2006.

[6] Richard M. Brualdi. *Introductory Combinatorics, 2nd edition*. Prentice Hall, 1992.

[7] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, Oakland, California, May 2006.

[8] Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high-speed networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, Napa, California, April 2004.

[9] Beate Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th International Cooloquium on Automata, Languages, and Programming*, pages 118–132, 1979.

[10] Peter Dencker, Karl Durre, and Johannes Heuft. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst.*, 6(4):546–572, 1984.

[11] S.C. Johnson. Yacc – yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, 1975.

[12] Sailesh. Kumar, Balakrishnan. Chandrasekaran, Jonathan. Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS 2007*, pages 155–164.

[13] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proceedings of the ACM SIGCOMM*, September 2006.

[14] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architectures for networking and communications systems*, pages 81–92, New York, NY, USA, 2006. ACM Press.

[15] Vern Paxson. The flex fast scanner generator, 1995. Available at `http://flex.sourceforge.net/`.

[16] Reetinder Sidhu and Viktor Prasanna. Fast regular expression matching using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2001.

[17] Randy Smith, Cristian Estan, and Somesh Jha. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, August 2008.

[18] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *IEEE Symposium on Security and Privacy*, May 2008.

[19] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. In *ACM CCS*, Washington, DC, Oct. 2003.

[20] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching. In *FCCM*, April 2004.

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

[22] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, 2006.