CompactDFA: Generic State Machine Compression for Scalable Pattern Matching

Anat Bremler-Barr
Computer Science Dept.
Interdisciplinary Center, Herzliya, Israel
Email: bremler@idc.ac.il

David Hay
Dept. of Electrical Engineering
Columbia University, NY, USA
Email: hdavid@ee.columbia.edu

Yaron Koral
Computer Science Dept.
Interdisciplinary Center, Herzliya, Israel
Email: koral.yaron@idc.ac.il

***Abstract*—Pattern matching algorithms lie at the core of all contemporary Intrusion Detection Systems (IDS), making it intrinsic to reduce their speed and memory requirements. This paper focuses on the most popular class of pattern-matching algorithms, the Aho-Corasick–like algorithms, which are based on constructing and traversing a *Deterministic Finite Automaton (DFA)*, representing the patterns. While this approach ensures deterministic time guarantees, modern IDSs need to deal with hundreds of patterns, thus requiring to store very large DFAs which usually do not fit in fast memory. This results in a major bottleneck on the throughput of the IDS, as well as its power consumption and cost.**

We propose a novel method to compress DFAs by observing that the name of the states is meaningless. While regular DFAs store separately each transition between two states, we use this degree of freedom and encode states in such a way that all transitions to a specific state can be represented by a single prefix that defines a set of current states. Our technique applies to a large class of automata, which can be categorized by simple properties. Then, the problem of pattern matching is reduced to the well-studied problem of *Longest Prefix Matching (LPM)* that can be solved either in TCAM, in commercially available IP-lookup chips, or in software. Specifically, we show that with a TCAM our scheme can reach a throughput of 10 Gbps with low power consumption.

I. Introduction

Pattern matching algorithms lie at the core of all contemporary Intrusion Detection Systems (IDS), making it intrinsic to reduce their speed and memory requirements. The most common algorithm used today is the seminal Aho-Corasick [1] algorithm, which uses a Deterministic Finite Automaton (DFA) to represent the pattern set. Then, the input is inspected symbol by symbol by traversing the DFA. Since given the current state and the symbol from the input, the DFA should determine which state to transit to, a naive implementation stores a rule for each possibility; namely, one rule per each pair of a state and a symbol. This resolves in prohibitively large memory requirement: 1.5 GB for ClamAV virus signature database that consists of about 27 000 patterns, and 73 MB for Snort IDS which has approximately 6 400 patterns. A direct impact of this high memory requirement is that the DFA must be stored in slow memory; this translates to a maximum throughput of 640 Mbps assuming 100 ns memory access time for DRAM-like memory.

Recently, there were many efforts to compress Aho-Corasick DFA and by that improve the algorithm performance [2]–[7]. While most of these works suggest dedicated hardware solutions, in this paper we present a generic DFA compression algorithm and store the resulting DFA in off-the-shelf hardware. Our novel algorithm works on a large class of *Aho-Corasick–like DFA*s, whose unique properties will be defined in the sequel. This algorithm reduces the rule set to the minimum possible size, *only one rule per state*. A key observation is that in prior works the state codes were chosen arbitrarily without a special meaning; we take advantage of this degree of freedom and add information about the state properties in the state code. This allows us to encode all transitions *to* a specific state by a single *prefix* which captures a set of current states. Moreover, if a state matches more than one rule, the rule with the longest prefix is selected. Thus, our scheme reduces the problem of pattern matching to the well-studied problem of *Longest Prefix Matching (LPM)*.

The reduction in the number of rules comes with a small overhead in the number of bits used to encode a state. For example, a DFA based on Snort requires 17 bits when each state is given an arbitrary code, while when using our scheme it requires 36; for ClamAV's DFA the code width increases from 22 bits to 59 bits.

In addition, we present an extension to our basic scheme, called *CompactDFA for total memory minimization*, that aims at minimizing the product of the number of rules and the code width, rather than only the number of rules. This captures situations in which, at some point, the reduction in the number of rules is not worth the additional bits in the state code. Specifically, 26 bits for Snort's DFA and 38 bits for ClamAV's DFA reduce the memory requirement to 0.7 MB and 16.4 MB, respectively.

One of the main advantages of CompactDFA is that it fits into commercially available IP-lookup solutions, implying they can be used also for performing fast pattern matching. We demonstrate the power of this reduction by implementing the Aho-Corasick algorithm on an IP-lookup chip (e.g., [8]) and on a TCAM.

Specifically, in TCAM, each rule is mapped to one entry. Since TCAMs are configured with entry width that is a multiple of 36 bits or 40 bits, minimizing the number of bits to encode a state is less important and the basic CompactDFA that minimizes the number of rules is more adequate. In our TCAM solution, we reduce the power consumption of the

TCAM by taking advantage of the fact that today's vendors partition the TCAM to blocks and give the ability to activate, in every lookup, only part of the blocks. We suggest to divide the rules to different blocks, where each block is associated with different subset of the symbols. By dividing the rules to blocks in this way, we also reduce the amount of bits required for encoding the symbol field of the rule to the logarithm of the number of symbols that are mapped to the same block.

The small memory requirement of the compressed rules and the low power consumption enables the usage of multiple TCAMs simultaneously, where each perform pattern matching for different sessions or packets. Furthermore, one can take advantage of the common multi-processors architecture of contemporary security tools and design high throughput solution, applicable to the common case of multi sessions/packets. Notice that while state-of-the-art TCAM chip are of size 5 MB, high throughput can be achieved using multiple small TCAM chips. Specifically, for Snort pattern set, we achieve a throughput of **10 Gbps** by using 5 small TCAMs of 0.5 MB each, and as much as **40 Gbps** with 20 small TCAMs.

Paper Organization: The paper is organized as follows: Section II gives background information. In Section III, we describe our new CompactDFA scheme. Its implementation with IP-lookup techniques is discussed in Section IV. In Section V we show experimental results using the two databases of Snort and ClamAV. An overview on the related work is in Section VI. Concluding remarks and future work appear in Section VII.

II. Background

This paper focuses on the seminal algorithm of Aho-Corasick (AC) [1], which is the de-facto standard for pattern matching in *network intrusion detection systems* (NIDS). Basically, the AC algorithm constructs a *Deterministic Finite Automaton* (DFA) for detecting all occurrences of a given set of patterns by processing the input in a single pass. The input is inspected symbol by symbol (usually each symbol is a byte), such that each symbol results in a state transition. Thus, the AC algorithm has deterministic performance, which does not depend on the specific input and therefore is not vulnerable to various attacks, making it very attractive to NIDS systems.

The construction of AC's DFA is done in two phases. First, the algorithm builds a *trie* of the pattern set: All the patterns are added from the root as chains, where each state corresponds to a single symbol. When patterns share a common prefix, they also share the corresponding set of states in the trie. In the second phase, additional edges are added to the trie. These edges deal with situations where the input does not follow the current chain in the trie (that is, the next symbol is not an edge of the trie) and therefore we need to transit to a different chain. In such a case, the edge leads to a state corresponding to a prefix of another pattern, which is equal to the longest suffix of the previously matched symbols.

Formally, a DFA is a 5-tuple structure $\langle S, \Sigma, s_0, F, \delta \rangle$, where S is the set of states, Σ is the alphabet, $s_0 \in S$ is the initial state, $F \subseteq S$ is the set of accepting states, and $\delta : S \times \Sigma \mapsto S$ is the transition function. It is sometimes useful to look at the DFA as a directed graph whose vertex set is S and there is an edge between s_1 and s_2 with label x if and only if $\delta(s_1, x) = s_2$. The input is inspected one symbol at a time: Given that the algorithm is in some state $s \in S$ and the next symbol of the input is $x \in \Sigma$, the algorithm applies $\delta(s, x)$ to get the next state s'. If s' is in F (that is, an accepting state) the algorithm indicates that a pattern was found. In any case, it then transits to the new state s'. Upon beginning of the input, the algorithm is in state s_0.

We use the following simple definitions to capture the meaning of a state $s \in S$: The *depth* of a state s, denoted depth(s), is the length (in edges) of the shortest path between s and s_0. The *label* of a state s, denoted label(s), is the concatenation of symbols of the edge of the shortest path between s_0 to s. Further, for every $i \leq$ depth(s), suffix$(s, i) \in \Sigma^*$ (respectively, prefix$(s, i) \in \Sigma^*$) is the suffix (prefix) of length i of label(s). The *code* of a state s, denoted code(s), is the unique number that associated with the state, i.e., the number that encodes the state. Traditionally, this number is meaningless and is chosen arbitrarily; in this paper we take advantage on this degree of freedom.

We use the following classification of DFA transitions (cf. [9]):

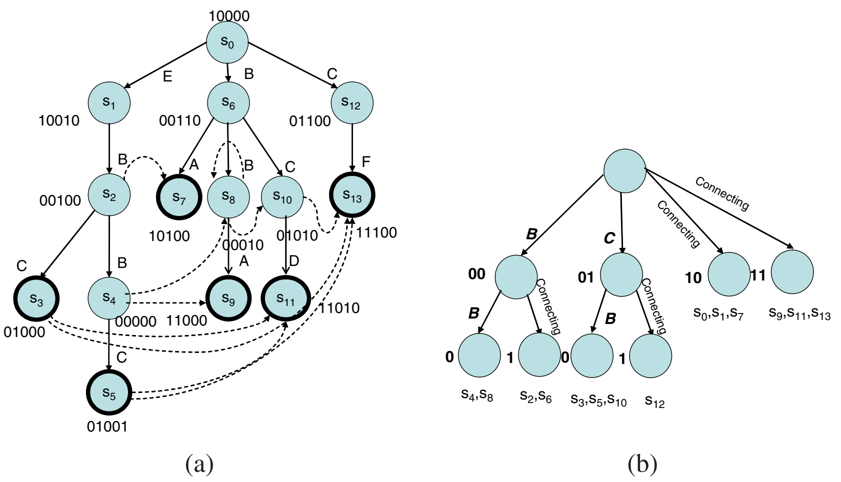
- **Forward transitions** are the edges of the trie; each forward transition links a state of some depth d to a state of depth $d + 1$.
- **Cross transitions** are all other transitions. Each cross transition links a state of depth d to a state of depth d' where $d' \leq d$. Cross transitions to the initial state s_0 are also called **failure transitions**, and cross transitions to states of depth 1 are also called **restartable transitions**.

The DFA is encoded and stored in memory, which is accessed by the AC algorithm when inspecting the input. A straightforward encoding is to store the set of rules (one rule for each transition) with the following fields:

Current state field	Symbol field	Next state field

We denote each rule as a tuple $\langle s_i, x, s_j \rangle$. The rule $\langle s_i, x, s_j \rangle$ is in the DFA rule set if and only if $\delta(s_i, x) = s_j$. A naive approach stores the rules in a two dimensional matrix (see, e.g, Snort implementation [10]), where a row represents a current state and a column represents a possible symbol. Thus, upon inspecting a symbol of the input, the algorithm reads entry number (s_i, x) and obtains the next state s_j. Fig. 1(a) depicts the DFA for the pattern set: CF, BCD, BBA, BA, EBBC, and EBC. The resulting automaton has 14 states (corresponding to the nodes of the graph). For readability, only the forward and cross transitions to depth 2 and above are shown, while all failure transitions (e.g., $\langle s_1, D, s_0 \rangle$) and restartable transitions (e.g., $\langle s_1, C, s_{12} \rangle$) are omitted; thus, Fig. 1(a) presents only 24 out of $14 \cdot 256 = 3\,584$ transitions (these transitions correspond to the edges of the graph).

Note that this naive encoding requires a matrix of size $|\Sigma| \cdot |S|$ with one entry per DFA edge. In the typical case, when the input is inspected one byte at a time, the number of edges,



	Current State	Symbol	Next State
1	00000	C	01001(s_5)
2	00100	C	01000(s_3)
3	00100	B	00000(s_4)
4	10010	B	00100(s_2)
5	010**	D	11010(s_{11})
6	000**	A	11000(s_9)
7	01***	F	11100(s_{13})
8	00***	C	01010(s_{10})
9	00***	B	00010(s_8)
10	00***	A	10100(s_7)
11	*****	E	10010(s_1)
12	*****	C	01100(s_{12})
13	*****	B	00110(s_6)
14	*****	*	10000(s_0)

(c)

Fig. 1. A toy example. (a) AC-DFA for the patterns {EBC,EBBC,BA,BBA,BCD,CF}. Failure and restartable transitions are omitted for clarity. The CompactDFA code of the states is the 5 bits near each state; (b) The Common Suffix Tree; (c) The Rules of the compressed DFA.

and thus the number of entries is $256|S|$. For example, Snort patterns required 73.5 MB for 6 423 patterns that translate into 75 256 states (see Section V). Reducing the space requirement of the DFA has a direct impact on the algorithm performance, since smaller DFAs can fit into faster memory, and thus require significantly less time to access.

Our compression algorithm succeeds to compress the number of rules to a minimum of one rule per state. In the toy example presented in Fig. 1(a) our algorithm requires 14 rules (see Fig. 1(c)).

III. The CompactDFA Scheme

In this section we explain our *CompactDFA Scheme*. We begin by explaining the scheme output, namely a compact encoding of the DFA and continue by describing the algorithm and the intuition behind it.

A. CompactDFA Output

The output of the CompactDFA scheme is a set of compressed rules, such that there is only one rule per state. This is achieved by cleverly choosing the code of the states. Unlike traditional AC-like algorithms, in our compact DFA each rule has the following structure:

Set of current states	Symbol Field	Next state code

The set of current states of each rule is written in a prefix style, i.e., the rule captures all states whose code matches a specific prefix. Specifically, for each state s, let $N(s)$ be the incoming neighborhood of s, namely all states that has an edge to s. For every state $s \in S$, we have one rule where the current state is the common prefix of the code of the states in $N(s)$ and the next state is s. Note that the symbol that transfers each state in $N(s)$ to a state s is common for all the states in $N(s)$ due to AC-like algorithm properties (see Property 2 in Section III-C).

Fig. 1(c) shows the rules produces by CompactDFA on the DFA of Fig. 1(a). For example, Rule 5 in Fig. 1(c), which is $\langle 010{\star}{\star}, D, 11010(s_{11})\rangle$, is the compressed rule for next state s_{11} and it replaces three original rules: $\langle 01000(s_3), D, 11010(s_{11})\rangle$, $\langle 01001(s_5), D, 11010(s_{11})\rangle$, and $\langle 01010(s_{10}), D, 11010(s_{11})\rangle$.

In the compressed set of rules, a code of a state may match multiple rules. Very similar to forwarding table in IP networks, the rule with the *Longest Prefix Match (LPM)* determines the action. In our example, this is demonstrated by looking at Rules 6 and 10 in Fig. 1(c). Suppose that the current state is s_8, whose code is 00010, and the symbol is A. Then, Rule 10 is matched since $00{\star}{\star}{\star}$ is a prefix of the current state. In addition, Rule 6, with current state $000{\star}{\star}$, is also matched. According to the longest prefix match rule, Rule 6 determines the next state.

B. CompactDFA Algorithm

This section describes the encoding algorithm of CompactDFA and gives the intuition behind each of its three stages: State Grouping (Algorithm 1, Section III-D), Common Suffix Tree Construction (Algorithm 2, Section III-E), and State and Rule Encoding (Algorithm 3, Section III-F).

The first stage of our algorithm is based on the following insight. Suppose that each state s is encoded with its label; our goal is to encode with a single rule the incoming neighborhood $N(s)$, which should appear in the first field of the rule corresponding to next state s. This is done by observing that the labels of all states in $N(s)$ share a common suffix, which is the label of s without its last symbol. Thus, by assigning code$(N(s))$ to be label(s) without its last symbol, padded with "don't care" symbols in its beginning, and applying a *longest suffix match* rule, one capture correctly the transitions of the DFA.

For example, consider Fig. 1(a). The code of state s_7 is BA. $N(s_7) = \{s_6, s_2\}$, label(s_6) = B and label(s_2) = EB; their common suffix is B, and indeed the code of $N(s_7)$ is "${\star}{\star}{\star}$B". On the other hand, code$(N(s_9))$ = code$(\{s_4, s_8\})$ = "${\star}{\star}$BB"; thus, if the current state is s_4, whose label is EBB, and the symbol is A, the next state is s_9 whose corresponding rule has longer suffix than the rule corresponding to s_7.

As demonstrated above, the longest suffix match rule should be applied to resolve conflicts when more than one rule is matched. Intuitively, this encoding is correct since all incoming edges to a state s (alternatively, all edges from $N(s)$) share the same suffix which is code$(N(s))$. Moreover, a cross transition edge from a state s with symbol x always ends up at a state s' whose label is the longest suffix (among all state labels) of the concatenation of label(s') with x.

However, this code is, first and foremost, extremely wasteful (and thus unpractical), requiring a 32 bit code for the automaton of Fig. 1(a) (namely, to encode 4 byte labels) and hundreds of bits for Snort's DFA. In addition, it uses a longest suffix match rule, while current off-the-shelf IP lookup chips employ longest prefix match. Thus, in the second stage, we transform our code to fit longest *prefix* match; this transformation is done with a dedicated data structure, the Common Suffix Tree. Finally, the third stage translates the wasteful code to a compact code, with a significantly smaller number of bits.

In the rest of this section, we first discuss the necessary properties of the AC-like DFA and then describe the three stages of the algorithm. We conclude by presenting a variation of CompactDFA that aims at minimizing total memory requirement (Section III-G).

C. The Aho-Corasick Algorithm-like Properties

Our proposed CompactDFA algorithm works for AC-like DFAs. In this section we give a short description of the four DFA properties that are necessary for CompactDFA's correctness. The formal definitions and proofs appear in [11].

Property 1: Given a set of patterns and an input string, the DFA accepts the input string if one of the patterns exists in the input. The AC-like algorithm is also required to *trace back* which pattern (or patterns) is matched by the input.

Property 2: All the incoming transitions to a specific state are labeled with the same symbol. Notice that this property is crucial to our CompactDFA algorithm.

Property 3: There are no redundant states at the DFA. This implies that every state has a path of forward transitions to an accepting state.

Property 4: Any specific input leads to exactly one specific state; this is a by-product of the determinism of the automaton.

Note that Property 1 and 3 imply the following two simple lemmas, which we later use to prove that our scheme is correct:

Lemma 1: For any state s of depth d, the last d symbols on any path from s_0 to s are equal to the last d symbols in label(s).

Lemma 2: Cross-transitions from a state $s \in S$ with symbol $x \in \Sigma$ are always to the state s_1 whose label label(s_1) is the longest possible suffix of label$(s)x$.

D. Stage I: State Grouping

In order to group the states, we calculate two parameters for each state: its *common suffix* (*CS*) and its *longest common suffix* (*LCS*). We describe here the motivation behind these parameters, using our wasteful code that encodes each state with its label[1]. Notice that with this code it is easy to identify all the states that go to a specific state s, since they must share a common suffix, which equals label s without its last symbol (by Lemma 1). We define this suffix as the parameter CS(s), which is calculated for Algorithm 1, Lines 1-7. Note that if a state has only one incoming transition, it should not be compressed and therefore its CS value is set to $\perp$.

In the toy example of Fig. 1(a): CS(s_{13}) = C, which is the common suffix of s_3, s_5, s_{10} and s_{12}. CS(S_{11}) = BC, which is the common suffix of s_3, s_5 and s_{10}. On the other hand, CS$(s_5) = \perp$ since s_5 has only one incoming transition.

Observe now that all the rules *to* a next state s can be encoded together since all current states have a common suffix CS(s) (Lemma 1) and all the incoming transitions are with the same symbol, denoted by x (Property 2). Thus, we can compress the rules to $\langle \star\star\cdots\star\star\text{CS}(s), x, \text{label}(s)\rangle$[2] and use the label of the current state while looking for the right rule. The encoding is correct, that is, yielding to the same next state as in the original DFA if and only if when more than one rule is matched, the rule with the longest suffix (i.e. the longest CS) is chosen. This stems from the fact that in an AC-like algorithm the cross transitions are always to the longest prefix of the pattern that exists in the DFA (see Lemma 2).

Hence, encoding the state by its label has almost the desired properties, except that it works on suffixes rather than prefixes and uses too many bits. To shorten each state code, we calculate our second parameter LCS, which identifies the important suffix in the label state (see Algorithm 1, Lines 8-11). LCS(s) stands for *longest common suffix* of state s, which is the longest CS(s') such that s has outgoing edges to s'; calculating CS and LCS for all states can be easily done in linear time. The main idea is that the prefix bytes in a state label that are not part of its LCS do not affect which rule matches the state.

In the toy example of Fig. 1: LCS(s_5) = BC since it has outgoing edges to s_{11} with CS(s_{11}) = BC and s_{13} with CS(s_{13}) = C.

E. Stage II: Common Suffix Tree

Next, we show how to encode the rules with smaller number of bits and transform them from being defined on suffixes to being defined on prefixes. For this we first construct an auxiliary data structure, the *Common Suffix Tree*, whose nodes correspond to the different LCS values (the set L) and for every two such values $\ell_1, \ell_2 \in L$, ℓ_1 is an ancestor of ℓ_2 if and only if ℓ_1 is a suffix of ℓ_2 (Algorithm 2, Lines 1-3). Note

[1]By Property 4, this is a valid code since it assigns a unique value for each state.

[2]The number of "$\star$" symbols is chosen arbitrarily for this toy demonstration.

Algorithm 1 State Grouping. Calculating the parameters CS and LCS for all states.

```
1: for all state $s \in S$ do
2:     if $s$ has more than one incoming link then
3:         CS($s$) := label($s$) without the link symbol
4:     else
5:         CS($s$) := $\bot$                    ▷ $\bot$ is the empty word
6:     end if
7: end for
8: for all state $s \in S$ do
9:     $O_s$ := states within distance 1 of $s$ on the DFA graph
           ▷ $s$ has an outgoing transition to each $s' \in O_s$
10:    LCS($s$) := CS($s'$) with maximal length among all $s' \in O_s$
11: end for
```

Algorithm 2 Constructing the Common Suffix Tree

```
1: $L := \{\mathrm{LCS}(s) | s \in S\}$          ▷ the set of all LCS values
2: build a Common Suffix Tree with vertex set $L$ such that
     for every $\ell_1, \ell_2 \in L$, $\ell_1$ is an ancestor of $\ell_2$ if and only if $\ell_1$ is a suffix of $\ell_2$
3: for all suffix tree node $\ell \in L$ do
4:     if $\ell$ is not a leaf in the Common Suffix Tree then
5:         $n$ := number of node $\ell$'s children          ▷ $n > 0$
6:         add $x$ *connecting nodes* as a children to $\ell$ such that $n+x$ is a power of 2 and $x \geq 1$
7:     end if
8: end for
9: for all state $s \in S$ do
10:    if LCS($s$) is a leaf in the Common Suffix Tree then
11:        link $s$ as a child of LCS($s$)
12:    else
13:        link $s$ as a child of the connecting node of LCS($s$) with the least number of children   ▷ all children of an LCS are balanced between its connecting nodes
14:    end if
15: end for
```

that it suffices to consider only LCS values since the set L contains all the possible CS values[3].

For every *internal* node in the tree we add *connecting nodes*, such that the total number of its children is a power of two and there is at least one such connecting node (Algorithm 2, Lines 3-8). Next, we link the states of the DFA. A state is linked to the node that corresponds to its LCS: If the node is a leaf, we link the state directly to the node. Otherwise, we link the state to one of the connecting nodes, such that the number of states linked to sibling connecting nodes is balanced (Algorithm 2, Lines 9-15). The main purpose of these connecting nodes is to balance as much as possible the states so that the tree can be encoded with smaller number of bits.

The Common Suffix Tree of the toy example is shown in Fig. 1(b). Note that the common suffix LCS="BC" translates to "CB". Roughly, in order to map a suffix rule to a prefix rule, all labels are written in a reverse order. For clarity, we label

[3]We can prove that for any $s' \in S$, $\mathrm{CS}(s') \in \{\mathrm{LCS}(s)|s \in S\}$: Consider a forward transition from state s to state s' with symbol x. By the definition of CS, $\mathrm{CS}(s') = \mathrm{label}(s)$. Note that from the LCS definition, $\mathrm{LCS}(s) = \mathrm{label}(s)$. This is due to the fact that all other outgoing edges from s are to depth $\leq \mathrm{depth}(s')$ and therefore their CS parameter is shorter or equal to $\mathrm{CS}(s')$. We can also prove that $\bot \in \{\mathrm{LCS}(s)|s \in S\}$.

Algorithm 3 State and Rule Encoding

```
1: for all edges $\langle \ell_1, \ell_2 \rangle$ of the Common Suffix Tree do
2:     length($\langle \ell_1, \ell_2 \rangle$) := log⌈number of $\ell_1$'s children⌉
         ▷ the number of bits required to enumerate over all outgoing edges of $\ell_1$
3: end for
4: $w$ := the length of the longest path in the tree
     ▷ $w$ is the *code width*, the number of bits required to encode the tree
5: enumerate all edges of each node
         ▷ each edge get a code of width equals to its length
6: for all node $\ell$ of the Common Suffix Tree do
7:     assign the code of $\ell$ as the concatenation of all edges' codes on the path between the $\ell$ and the root. Pad with 0 to fit width $w$.
8: end for
9: for all state $s \in S$, $s \neq s_0$ do          ▷ Encoding the rules
10:    add the following rule to the rule set:
11:    if CS($s$) $\neq \bot$ then
12:        current-state set is the code of node CS($s$), padded with $\star$ to fit $w$.
13:    else
14:        current-state is the code of node $s'$, where $s$ has its single incoming link from $s'$
15:    end if
16:    symbol is the label on the incoming link of $s$
17:    next state is the code of node $s$
18: end for
19: Add rule for $s_0$:
20:   current-state set is $w$ bits of $\star$
21:   symbol is $\star$
22:   next state is the code of node $s_o$
```

each edge with a string representing the difference between the LCS of its two endpoints. Note that the Common Suffix Tree is in fact a trie if all edges are labeled with a single symbol. However, this does not hold in general.

F. Stage III: State and Node Encoding

In the third step we encode the Common Suffix Tree and then the states and rules.

We first find w, the *code width*, which is the number of bits required to encode the entire tree (Algorithm 3, Lines 1-4)[4]. The edges are encoded by simply enumerating over all sibling edges (edges that originate from the same node); each edge is encoded with its binary ordinal number and a code width of $\lceil \log n + 1 \rceil$ bits, where n is the number of sibling edges. Then, each node is encoded using the concatenation of the edge codes of the path between the root s_0 and the node. Let code(v) be the code given to node v. The CS of a state s is encoded by the code of the node that corresponds to CS(s) padded with $\star$ symbols to reach a code width w. The states are encoded using their corresponding node in the Common Suffix Tree; in order to expand the code to w bits, "0" symbols

[4]This computation is done in a bottom-up approach and is best described in a recursive manner. For a given node a in the tree, assume $a_0, \ldots, a_{n-1}$ are its direct children. Furthermore, assume that from previous computation steps it is known that the subtree whose root is a_i requires b_i bits. The encoding of the entire subtree of a will require $b = b_{\max} + \log_2(n)$ bits where $b_{\max}$ is the maximum over $b_0, \ldots, b_{n-1}$ or 0 if no descendant nodes exist. We set w to be $b_{\max}$ of the root vertex.

are added to the code. Finally, we compress all the rules that share the same next state $s \neq s_0$ in the following way: Let x be the symbol of all the incoming edges to s (Property 2). We encode the single rule $\langle \text{code}(\text{CS}(s)), x, \text{code}(s) \rangle$, if there are more than one such edge; otherwise, we encode the rule $\langle \text{code}(s'), x, \text{code}(s) \rangle$, where s' is the origin of the single edge to s. A transition from current state s' is determined by searching the rule set with $\text{code}(s')$ and applying the longest prefix match rule. We also add explicitly the default rule for s_0 (Algorithm 3, Lines 19-22). Fig. 1(c) shows the compressed rules for the toy example.

Note that as a direct consequence of encoding and building the Common Suffix Tree we get that $\text{LCS}(i)$ is a suffix of $\text{LCS}(j)$ if and only if $\text{code}(\text{LCS}(i))$ is a prefix of $\text{code}(\text{LCS}(j))$. Hence, the correctness of this encoding is straightforward from the correctness of our wasteful intermediate encoding (of Section III-D).

G. *CompactDFA for total memory minimizations*

While CompactDFA minimizes the number of rules, this minimization is achieved at the cost of increasing the number of bits used to encode a state. For some applications, however, we might desire to minimize the *total memory requirement*, namely, the product of the number of rules and the total number of bits required to encode a rule. Hence, we might prefer to use less bits per state code, even at the cost of slightly increasing the number of rules.

One way to reduce memory requirements is to encode by one rule only the rules whose next state is of depth at most D, where D is a parameter of the algorithm. As shown by the experiments in Section V, most of the rules are due to transitions to a state with small depth, implying that compressing them might be enough. In addition, even though only a small fraction of the rules corresponds to states with large depth, they might increase the code width significantly due to the structure of the Common Suffix Tree. The optimal D for total memory requirement minimization can be found simply by checking linearly possible D values and choosing the optimal one.

Our algorithm for a given depth D—which we call the *Truncated CompactDFA*—is a simple variation on the original CompactDFA and is done by truncating the Common Suffix Tree at depth D. Nodes that appear in the original Common Suffix Tree in depth more than D are connected to their ancestor at depth D if and only if they are state nodes. All other nodes are omitted (we count the depth in nodes; specifically, for $D{=}0$ we get an empty structure). Notice that the *Truncated CompactDFA* works on the Common Suffix Tree and not on the DFA. However, it is easy to verify that for every state s, its depth in the DFA is at least the depth of the corresponding node in the Common Suffix Tree; thus, if the DFA depth of s is smaller than D then all rules with "next state" s are compressed.

The same encoding algorithm is used as in the original CompactDFA, handling just CS values that appear in the (truncated) Common Suffix Tree (and all states of the DFA). By construction, the resulting suffix tree is more "fat", thus requiring a smaller code width. The main change is in the rules encoding, since we compress only rules with next state s whose corresponding CS appears in the truncated Common Suffix Tree. For all other rules, we explicitly encode all transitions, thus for these specific states the number of rules is equal to their number in the uncompressed form. Note that this is usually a small number since the number is proportional to the *in-degree* of the next-state rather than to its out-degree which is always $|\Sigma|$ (typically, 256). This rule encoding is correct due to the longest prefix match rule. Comparing to the original CompactDFA, this variant *decompresses* some prefixes to their full (and thus longer) exact values. The formal proof and the pseudo code of this algorithm appear in [11].

Finally, it should be emphasized that D in real-life data is small, 5 for the Snort pattern set and 6 for ClamAV pattern set (see Section V).

IV. Implementing CompactDFA using IP-lookup Solutions

This section discusses how input inspection on our compressed rule set is performed. Basically, this boils down to looking up at all the prefixes and finding the longest prefix match.

Although this is not a trivial task, it is similar to the well-studied *IP-lookup Problem* [12]. Since IP-lookup is a fundamental problem in networking, and, in fact, one of the most basic tasks of routers, it was extensively studied in the past and many satisfactory solutions are commercially available such as IP-lookup chips, TCAM memory (which is the most common solution), or special software solution.

The IP lookup problem is to determine where to forward a packet. A router maintains forwarding table which consists of networks, represented by the common prefix of the network's addresses, and their corresponding next hop. Upon receiving a packet, the router looks up in its forwarding table for the longest prefix that matches the packet destination address.

A straightforward reduction of CompactDFA output to IP-lookup forwarding-table is done by treating the concatenation of the symbol and the prefix of current states as the network prefix, and the next state as the next hop. A second approach is to use separate IP-lookup forwarding table for every symbol. Namely, the next symbol indicates the number of the table containing its corresponding rules (and only these rules). The second approach is more applicable to a software solution or a TCAM with its ability to use blocks (see Section IV-A). In addition, the second approach does not require to encode the symbol in the rule. Furthermore, hybrid solution can be implemented by partitioning the symbols to sets and performing a separate IP-lookup for each set; in this case, only part of the symbol should be encoded.

In this paper we concentrate on TCAMs—the common IP-lookup hardware solution (Section IV-A). In Section IV-B, we discuss preliminary results on using other IP-lookup solutions.

A. Implementing CompactDFA with TCAM

Ternary Content-Addressable Memory (TCAM) devices are increasingly used in the industry for performing high-speed packet classification and IP-lookup [13]. TCAM enables parallel matching of a key against all entries using one access cycle and thus provides high throughput that is unparalleled by software-based solutions. A TCAM is an associative memory hardware device that can be viewed as an array of fixed-width entries. Each TCAM entry consists of ternary digits: 0, 1, or $\star$ (don't care). When a key matches multiple TCAM entries, the TCAM returns the index of the first matching entry. This index is then used to locate the information specifying which actions to apply to the packet.

Two main disadvantage of TCAM are its high cost per bit and its high power consumption requirement [14]. We discuss now how CompactDFA is able to deal with these two problems.

In CompactDFA, we minimize the rule set and thus the space requirement. The encoding of a rule is straightforward, since each rule can be viewed as a ternary string. Similarly to actions in packet classifications, the information of the next state is kept in a direct-access array stored in a separate SRAM. Within the TCAM, rules are sorted in a way such that the longer prefixes come first as shown in Fig. 1(c). If the number of rules to encode is t and each TCAM rule requires b bits for state encoding and 8 bits to encode the next symbol (in the common case of byte by byte DFA traversal), our implementation requires $(b + 8) \cdot t$ bits of TCAM and $b \cdot t$ bits of SRAM.

However, commercially available TCAMs have fixed entry widths, which are multiples of either 36 bits or 40 bits. Hence, minimizing the code width is significant only in the case where this minimization enables the usage of smaller entry widths (e.g., in a 40 bit TCAM, reducing the entry width $b + 8$ from 81 to 80 is important, while reducing it from 100 to 81 has no effect). Our experimental results consider this property of commercial TCAM devices (see Section V).

The high power consumption of TCAM devices is one of their major disadvantages, and there has been an intensive research effort to reduce the power consumption of TCAMs in IP-Lookup settings [7], [15]–[17]. Those solutions use the fact that TCAM vendors provide a mechanism that reduces power consumption by selectively addressing smaller portions of the TCAM at a time. This mechanism is applicable also to our setting. As we discussed earlier, we can partition the rules among the blocks by their next symbol field. Note that these blocks are of fixed size, hence it is preferable to balance the number of rules between the blocks, according to the number of rules corresponding to each $x \in \Sigma$. Moreover, this method requires less bits because there is no need to encode all the bits of the next symbol. For example, if the TCAM is divided into 64 different blocks and $|\Sigma| = 256$, only 2 bits (rather than 8) are required to represent a symbol within its block. Note that the total memory saving might be smaller due to an imperfect balancing of the rules among the TCAM blocks. Nevertheless, if there is sufficient memory, power consideration may be more important than space, therefore dividing the TCAM into several blocks would be crucial.

TCAM clock rate is generally 266 MHz, implying its throughput is 2 Gbps. Furthermore, smaller and cheaper TCAMs are of size 576 KB and are able to store about $128\,000$ rules; high-end TCAMs are of size 5 MB and can store up to a million rules. The small memory requirement of CompactDFA and its low power consumption enable the usage of multiple TCAMs in parallel, each performing pattern matching for a different session or packet (a multi-processors architecture, handling multiple sessions or packets in parallel, is common in contemporary security tools). Specifically, considering pattern matching based on Snort database, our implementation resolves in a throughout of 10 Gbps, when 5 small (and therefore cheap) 0.5 MB TCAMs are used; using 20 small 0.5 MB TCAMs results in a throughput of 40 Gbps (see Section V). For the much larger database of ClamAV, a throughput of 2 Gbps, is achieved by two 5 MB TCAM chips. We expect the power consumption to be moderate, since only the relevant block in each of the TCAM chips is activated.

Finally, it should be emphasized that TCAM devices exist in modern routers anyway, and in many cases they are under-utilized. This implies that our space and power-efficient solution can fit in these devices and provide pattern matching in parallel with any other application running on the TCAM (few extra bits in each entry might be needed to associate rules to applications).

B. Implementing CompactDFA with other IP-lookup solutions

One may wonder if applying IP-lookup solutions for pattern matching algorithms does not raise scalability issues. The current solutions of IP-lookup assume around a million prefixes and IP addresses of up to 64 or 128 bits (due to IPv6 requirements), which is significantly more than the requirements of CompactDFA for Snort rule set. As for ClamAV, there might be a problem with IP-lookup solution that cannot scale to a large number of rules, since ClamAV requires more than 1.5 million rules (namely, the number of states in its DFA).

However, a more careful look at the compressed rules shows that the majority of rules are not prefixes but *exact matches*, which occur when a state has only one incoming edge: In ClamAV, only 5% ($\sim 70\,000$) of the rules are prefixes, while in Snort only 17% are prefixes; this seems like a common characteristic of any real-life pattern set. Hence, for ClamAV, we can divide the solution to two components executed in parallel. The first component performs an exact match search (which can be done easily by hashing with a combination of Bloom filters). The second component runs the LPM only on the rules containing prefixes. The output of the exact match component (if exists) is always preferred to the output of the LPM.

Notice that ClamAV is an extreme example of a very large pattern set; most practical pattern sets are much smaller.

Finally, we discuss a promising direction of using a state-of-the-art chip for fast IPv6-lookup [8]. This recently-suggested

TABLE I
STATISTICS OF THE PATTERN SETS USED IN SECTION V.

	Patterns	States	DFA Size (Naive Implementation)
Snort	6 423	75 256	73.5 MB
ClamAV	26 987	1 565 874	1.48 GB

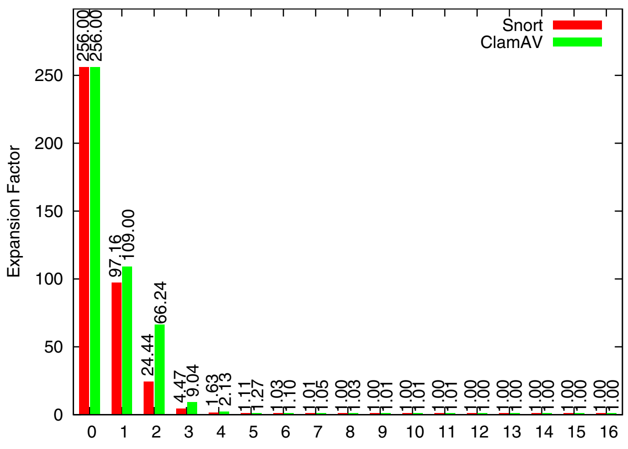


Fig. 2. The expansion factor of Snort's and ClamAV's DFAs under Truncated CompactDFA with different depths.

chip requires only 6 ns per lookup operation. Assuming that it takes 2 ns to resolve the exact match value, we get a pattern matching that requires at most 8 ns per byte (namely, a throughput of 1 Gbps). We intend to study the applicability of other IP lookup solutions in our future work.

V. EXPERIMENTAL RESULTS

We use two common pattern sets: Snort [10] (May 2009) and ClamAV [18] (February 2005). Snort contains mostly alphanumeric patterns, while ClamAV is constructed from binary virus signatures. Statistics of the pattern sets are detailed in Table I.

We first consider the *expansion factor* of CompactDFA, which is the ratio between the number of rules and the number of states. Note that the optimal expansion factor is 1, while traditional DFA implementations have expansion factor of $|\Sigma|$. Fig. 2 depicts the expansion factor of Snort and ClamAV DFAs, with *Truncated CompactDFA* of different depth values D.

Next, we calculate the memory requirement of Truncated CompactDFA with different depths, and consider the following TCAM implementations: (1) *Variable Entry Width (VEW)*, where each rule is stored with exactly the number of bits required to encode it; (2) *36-bit Width entry (W36)*, which considers that TCAM memory used in practice have entry widths which are multiple of 36 (recall Section IV-A); (3) *40-bit Width entry (W40)*, similar to W36 but with entry widths that are multiple of 40; and (4) *40-bit Width entry, 16 Blocks Implementation (W40B)*, considering the fact that modern TCAM devices can be divided into separate blocks (see Section IV-A). Table II presents the optimal depths and

TABLE II
SUMMARY OF EXPERIMENTAL RESULTS FOR SNORT AND CLAMAV PATTERN SETS.

Pattern Set	Objective	Impleme-ntation	Depth	Code Width (bits)	TCAM Size (MB)	SRAM Size (MB)
Snort	Minimizing Rules	VEW	16	36	0.39	0.32
		W36	16	36	0.65	0.32
		W40	16	36	0.72	0.32
		W40B	16	36	0.36	0.32
Snort	Minimizing Memory	VEW	5	26	0.34	0.26
		W36	5	26	0.36	0.26
		W40	10	32	0.36	0.29
		W40B	16	36	0.36	0.32
ClamAV	Minimizing Rules	VEW	30	59	12.51	11.01
		W36	30	59	13.44	11.01
		W40	30	59	14.93	11.01
		W40B	30	59	14.93	11.01
ClamAV	Minimizing Memory	VEW	7	38	8.99	7.43
		W36	30	59	13.44	11.01
		W40	30	59	14.93	11.01
		W40B	6	36	8.18	7.37

the resulting TCAM and SRAM size (recall that the "next state" field of each rule is stored in SRAM).

The optimal depth for Snort pattern based on W36 implementation is D=5, which requires a code width of only 26 bits; this results in 81 873 rules and translates to a memory of size 0.36 MB (that is, slightly more than the VEW memory size, which is 0.34 MB for this depth). Note that, as for depth D=6, there is a significant increase in W36 memory requirement, since the code width crosses the threshold of 36 bits, thus requiring TCAM entry width of 72 bits.

These results imply that a Snort pattern set can fit into a small 576 KB TCAM that supports 128 000 rules. Since CompactDFA is scalable power-consumption–wise, one can easily add more small TCAMs and gain linear performance boost. Assuming each TCAM provides throughput of 2 Gbps, using 5 TCAMs would results in a throughput of **10 Gbps** and 20 small TCAMs achieves a throughput of **40 Gbps**. In addition, notice that only two-thirds of the TCAM capacity is used. This extra TCAM space is beneficial in dividing the rules to different blocks (see section IV-A), without strict balancing the rules among the blocks.

For ClamAV, CompactDFA uses a code width of 59 bits for achieving expansion factor of 1. As for memory reduction, Truncated CompactDFA with depth D=6 is the optimal for W40B configuration. It uses code width of 36, has 1 716 390 rules, and requires 8.18 MB. Hence, a two 5 MB TCAM configuration may handle the large ClamAV pattern set at 2 Gbps.

VI. RELATED WORK

There has been an intensive effort for implementing compact AC-like DFA that can fit into faster memory.

Van-Lunteren [3] proposed a novel architecture that supports prioritized tables. His results are equal to CompactDFA with suffix tree that is limited to depth 2, thus having 25 (66) times more rules than the CompactDFA solution for Snort (ClamAV). CompactDFA in some sense is a generalization of [3] that eliminates all cross transitions. [9] proposed an n-step cache architecture to eliminate a part of the DFA's cross-transitions. This solution still has 4 (9) times more rules for Snort (ClamAV) than in CompactDFA. In addition, this solution, as other hardware solutions [2], [5], uses dedicated hardware and thus is not flexible.

As far as we know, this paper is the first to suggest a method of reducing the number of transitions in DFA to the minimum possible one, the number of DFA states. CompactDFA does not depend on any specific hardware architecture or any statistic property of data (as opposed to the work [19]).

Next, we compare our implementation of CompactDFA in TCAM hardware to the previous works using TCAM. The papers [20] and [21] encode segments of the patterns in the TCAM and do not encode the DFA rules. However both solutions require significantly larger TCAM (especially [21]) and more SRAM memory (order of magnitude more). The work of [6] encodes the DFA rules in TCAM as CompactDFA does. CompactDFA and [6] share the same basic observation, that we can eliminate cross transitions by using information from the next state label. However, [6] does not use the bits of the state to encode the information, on the contrary they just append to each state code the last m bytes of its corresponding label in order to eliminate cross transitions to depth m. Thus for depth 4 ClamAV [6] requires 62 bits while CompactDFA needs only 36 bits, and hence the solution is not scalable.

VII. Conclusion and Future Work

This paper shows a reduction of the pattern matching problem to the IP-lookup problem, two problems that at a first glance seems to be uncorrelated. We present an innovative scheme, CompactDFA, that performs this reduction. It gets as an input a DFA and produces compressed rule sets; each compressed rule defines the set of states that the rules apply to using a common prefix. A state may match more than one rule, and in this case the rule with the longest prefix determines the action—exactly as in the IP forwarding.

With this reduction, we have new arsenals of IP-lookup solutions, either in hardware or in software, that can boost up the performance of pattern matching, a major task in contemporary security tools. In this paper, we focus on the usage of TCAM for pattern matching, a hardware device that is commonly used for IP-lookup and packet classification and is deployed in many core routers. We show that with moderate size of TCAM space we can have fast pattern matching of 2 Gbps with low power consumption. Due to its small memory and power requirements, it is feasible to implement our architecture with several TCAMs working in parallel, such that each TCAM performs pattern matching on a different session. Such architecture can achieve very high throughput of 10 Gbps and beyond.

In the future, we would like to explore the performance of CompactDFA on AC-like DFA that advances few symbols at a time (and not just one symbol at a time, as traditional AC-like algorithms do). Processing a few symbols at a time increases the overall throughput of the pattern matching even on the same session. It is easy to build such a DFA that satisfies all the properties that our CompactDFA scheme requires.

Another promising research direction is to explore the applicability of other IP-lookup solutions. We aim at finding the fastest solution with good cost performance ratio that is able to cope with today increasing demands.

The jewel in the crown of our new architecture is that, with some modifications, it might have an important application on compression of DFAs resulting from *regular expression* pattern matching. This is part of our ongoing work.

Acknowledgment: We would like to thank Zvika Guz, Isaac Keslassy and Binyamin Zayith of the Knowledge Center on Chip MultiProcessors (CMP), Dept. Electrical Engineering, Technion for providing us computing resources.

References

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. of the ACM*, no. 6, pp. 333–340, 1975.
[2] L. Tan and T. Sherwood, "Architectures for bit-split string scanning in intrusion detection," *IEEE Micro*, pp. 110–117, 2006.
[3] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *Proc. IEEE INFOCOM*, Apr. 2006, pp. 1–13.
[4] R. T. Liu, N. F. Huang, C. N. Kao, C. H. Chen, and C. C. Chou, "A fast pattern-match engine for network processor-based network intrusion detection system," in *Proc. Information Technology: Coding and Computing (ITCC)*, 2004, pp. 97–101.
[5] D. Pao, W. Lin, and B. Liu, "Pipelined architecture for multi-string matching," *IEEE Comput. Archit. Lett.*, vol. 7, pp. 33–36, 2008.
[6] W. Lin and B. Liu, "Pipelined parallel AC-based approach for multi-string matching," in *Proc. IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2008, pp. 665–672.
[7] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz, "SSA: a power and memory efficient scheme to multi-match packet classification," in *Proc. ACM ANCS*, 2005.
[8] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100 gbps core router line cards," in *Proc. IEEE INFOCOM*, 2009.
[9] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *Proc. IEEE INFOCOM*, April 2008, pp. 166–170.
[10] "Snort," (May 2009). [Online]. Available: http://www.snort.org
[11] A. Bremler-Barr, D. Hay, and Y. Koral, "CompactDFA: Generic state machine compression for scalable pattern matching - technical report." [Online]. Available: www.tlc-networks.polito.it/hay/CompactDFA.pdf
[12] M. Ruiz-Sanchez, E. Biersack, and W. Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE NETWORK*, 2001.
[13] R. Panigrahy and S. Sharma, "Reducing TCAM power consumption and increasing throughput," in *Proc. IEEE HOT Interconnects*, 2002, pp. 107–112.
[14] D. E. Taylor, "Survey and taxonomy of packet classification techniques," in *ACM Computing Surveys*, September 2005.
[15] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: power-efficient TCAMs for forwarding engines," in *Proc. IEEE INFOCOM*, 2003.
[16] B. Agrawal and T. Sherwood, "Modeling TCAM power for next generation network devices," in *Proc. IEEE ISPASS*, 2006.
[17] D. Lin, Y. Zhang, C. Hu, B. Liu, X. Zhang, and D. Pao, "Route table partitioning and load balancing for parallel searching with TCAMs," in *Proc. IEEE IPDPS*, 2007.
[18] "ClamAV," (Version 0.82). [Online]. Available: http://www.calmav.net
[19] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, 2004.
[20] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proc. IEEE ICNP*, 2004, pp. 174–183.
[21] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker, "High performance string matching algorithm for a network intrusion prevention system (NIPS)," in *Proc. IEEE HPSR*, 2006.