CompactDFA: Scalable Pattern Matching Using Longest Prefix Match Solutions

Anat Bremler-Barr, *Member, IEEE,* David Hay, *Member, IEEE,* and Yaron Koral, *Student Member, IEEE*

***Abstract*—A central component in all contemporary Intrusion Detection Systems (IDS) is their pattern matching algorithms, which are often based on constructing and traversing a *Deterministic Finite Automaton (DFA)* that represents 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 used by common DFA-encoding 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 are 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 Match (LPM)*, which 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.

***Index Terms*—Aho-Corasick, Deep Packet Inspection, IP Lookup, NIDS, Pattern Matching, TCAM.**

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 (AC) [1] algorithm, which uses a Deterministic Finite Automaton (DFA) to represent the pattern set. Pattern matching algorithms, such as Aho-Corasick, are often used [18], [23] as a first step to identify regular expressions, which become popular in recent signature-sets. Each regular expression contains one or more (exact) string tokens (a.k.a anchors), which can be extracted offline. Thus, the DFA may be used to represent these anchors and to inspect the input traffic. A regular-expression engine is invoked only when all the anchors of a regular expression are found. We note that when the network is not under attack, these regular-expression engine-invocations are rare and do not induce high overhead on the pattern-matching algorithm [6], [18]. Yet, under certain scenarios, this overhead might become significant, leading some IDS to implement additional mechanisms that deal with regular expressions directly (e.g., [3], [15], [18], [19], [21]); these mechanisms are out of the scope of this paper.

After constructing the DFA, 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 naïve 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 [7], which consists of about 27 000 patterns, and 73 MB for Snort IDS [23], 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. An alternative approach is to implement an AC automaton using the concept of failure transitions. In such implementations, only part of the outgoing transitions from each state are stored explicitly. While traversing the automaton, if the transition from state s with symbol x is not stored explicitly, one will take the failure transition from s to another state s' and look for an explicit transition from s' with x. This process is repeated until an explicit transition with x is found, resulting in failure paths. Naturally, since only part of the transitions are stored explicitly, these implementations (sometimes referred to as AC NFAs) are more compact, but incur higher processing time.[1]

Recently, there were many efforts to compress the Aho-Corasick DFA and by that to improve the algorithm's performance [3], [15], [16], [20], [27]–[31]. While most of these works either suggest dedicated hardware solutions or introduce non-constant higher processing time, in this paper we present a generic DFA compression algorithm and store the resulting DFA in an off-the-shelf hardware. Our novel algorithm works on a large class of *Aho-Corasick–like DFA*s, whose unique properties are defined in the sequel. This algorithm reduces the rule set to *only one rule per state*. A key observation is that in prior works the state codes were chosen arbitrarily; we

Partial and preliminary versions of this paper appeared in Proc. Infocom'10 [4].

Manuscript received April 26, 2012; revised July 26, 2012; revised January 2, 2013; accepted February 19, 2013.

Anat Bremler-Barr is with the Department of Computer Science, Interdisciplinary Center, Herzliya, Israel. email:bremler@idc.ac.il.

David Hay is with the School of Engineering and Computer Science, Hebrew University, Jerusalem, Israel. email:dhay@cs.huji.ac.il.

Yaron Koral is with the Department of Computer Science, Tel Aviv University, Tel Aviv, Israel email:yaronkor@post.tau.ac.il.

This work was supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013)/ERC Grant agreement nº 259085 and by The Israeli Centers of Research Excellence (I-CORE) program, (Center No. 4/11).

[1] A classical result states that the longest failure path is at most the size of the longest pattern, and that, regardless of the traffic pattern, the total number of transitions (failure and explicit) is at most twice the number of symbols. This result does not take into account the representation of each single state, which determines the time it takes to figure out whether explicit rule exists.

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* that 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 Match (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 two extensions to our basic scheme. Our first extension, called *CompactDFA for total memory minimization*, aims at minimizing the product of the number of rules and the code width, rather than only the number of rules. The second extension deals with *variable-stride DFAs*, which were created to speed up the inspection process by inspecting more than one symbol at a time.

One of the main advantages of CompactDFA is that it fits into commercially available IP-lookup solutions, implying they may 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., [24]) 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.

We also deal with two obstacles in using TCAMs: the power consumption and the latency induced by the pipeline in the TCAM chip, which is especially significant since CompactDFA works in a closed-loop manner (that is, where the input for one lookup depends on the output of a previous lookup). To overcome the latency problem, we propose two kinds of interleaving executions (namely, inter-flow interleaving and intra-flow interleaving). We show that combining these executions provides low latency (in the order of few tens of microseconds) at high throughput. 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.

The small memory requirement of the compressed rules and the low power consumption enables the usage of multiple TCAMs simultaneously, where each performs pattern matching over different sessions or packets (namely, inter-flow interleaving). Furthermore, one can take advantage of the common multi-processors architecture of contemporary security-tools and design a high throughput solution, applicable to the common case of multi sessions/packets. Notice that while state-of-the-art TCAM chips are of size 5 MB, high throughput may be achieved using multiple small TCAM chips. Specifically, for Snort pattern set, we achieve a throughput of **10 Gbps** and latency of less than 60 microseconds by using 5 small TCAMs of 0.5 MB each, and as much as **40 Gbps** (and the same latency) with 20 small TCAMs.

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

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-scan 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 (each accepting state indicates that patterns were found in the input), 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'.

We use the following simple definitions to capture the meaning of a state $s \in S$: The *depth* of a state s, denoted $\text{depth}(s)$, is the length (in edges) of the shortest path between s_0 and s. The *label* of a state s, denoted $\text{label}(s)$, is the concatenation of the edge symbols of the shortest path between s_0 to s. Further, for every $i \leq \text{depth}(s)$, $\text{suffix}(s, i) \in \Sigma^*$ (respectively, $\text{prefix}(s, i) \in \Sigma^*$) is the suffix (prefix) of length i of $\text{label}(s)$. The *code* of a state s, denoted $\text{code}(s)$, is the unique number

that is associated with the state, i.e., the number that encodes the state. Traditionally, this number is chosen arbitrarily; in this paper we take advantage of this degree of freedom.

We use the following classification of DFA transitions (cf. [25]):

- **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 that 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 naïve approach stores the rules in a two dimensional matrix (see, e.g, Snort implementation [23]), 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 naïve encoding requires a matrix of $|\Sigma| \cdot |S|$ entries, one entry per DFA edge. In the typical case, when the input is inspected one byte at a time, the number of edges, and thus the number of entries is $256 \cdot |S|$. For example, Snort patterns required 73.5 MB for 6 423 patterns that translate into 75 256 states (see Section VIII). 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. Related Work

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

Van-Lunteren [29] 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 [29] that eliminates all cross transitions. Song et al. [25] 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 [20], [27], 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 [28]).

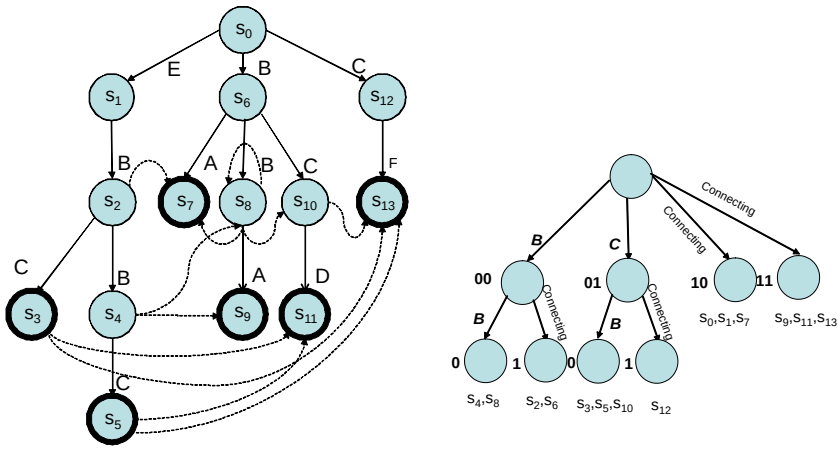
Next, we compare our implementation of CompactDFA in TCAM hardware to previous works that use TCAM. The papers [31] and [26], [30] encode segments of the patterns in the TCAM and do not encode the DFA rules. However both solutions require significantly larger TCAM (especially [30]) and more SRAM memory (order of magnitude more; see Section VIII for more details). The work of [2] encodes the DFA rules in TCAM as CompactDFA does. CompactDFA and [2] share the same basic observation, that we can eliminate cross transitions by using information from the next state label. However, [2] 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 to eliminate cross transitions to depth m. Thus, for depth 4, ClamAV [2] requires 62 bits while CompactDFA needs only 36 bits, and hence the solution is not scalable.

A recent work presented a method for state encoding in TCAM-based implementation of AC NFA rather than AC DFA [32]. While such a method, which was concurrently developed with ours, shares some of the insights of our work (e.g., it also eliminates all failure transitions), it is limited to the TCAM implementations where CompactDFA may be used with any known IP-lookup solution.

Following our work, several methods that perform regular expression matching using TCAM [19], [21] were suggested. These methods rely on the same high-level principle of our work, exploiting the degree of freedom in the way states are encoded. Since these methods deal with regular expression rather than exact string matching, they do not use AC-DFA, but other automata that are geared to handle regular expressions. Specifically, [19] uses D^{2}FA, while [21] uses both a DFA and a knowledge derived from a corresponding NFA; both methods then construct a tree (or a forest) structure, which is encoded similarly to CompactDFA. Finally, unlike our work, the methods in [19], [21], [32] do not deal with pipelined TCAM implementations (which are common in contemporary TCAM chips) and therefore suffer from significant performance degradation if such TCAMs are used.

Two additional methods of using TCAMs to handle regular expression matching were presented in [18], [33], showing orthogonal improvements to utilizing TCAMs. Specifically, [18] is based on implementing a fast and cheap pre-filter, so that only portion of the traffic should be fully-inspected; [33], on the other hand, suggests a technique that parallelizes the process of using TCAM by smartly dividing the pattern rule set and the flows to different TCAM blocks. Naturally, these two approaches can be easily combined with ours.

Finally, we note that [19] introduces the *table consolida-*



(a) (b)

	Current state	Symbol	Nxt State
1	00000	C	01001 (s_5)
2	00010	C	01000 (s_3)
3	00010	B	00000 (s_4)
4	10001	B	00010 (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	00001 (s_8)
10	00***	A	10100 (s_7)
11	*****	M	10010 (s_1)
12	*****	C	01100 (s_{12})
13	*****	B	00011 (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. (b) The Common Suffix Tree; (c) The Rules of the compressed DFA.

tion technique, which combines entries even if they lead to different states. This technique trades TCAM memory with a cheaper SRAM memory that stores the different states of each combined entry. Table consolidation, which requires solving complicated optimization problems, can be applied also to our results to further reduce the TCAM space.

IV. 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 IV-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 $\langle 010^{**}, D, 11010(s_{11})\rangle$, 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^{***} is a prefix of the current state. In addition, Rule 6, with current state 000^{**}, 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 IV-D), Common Suffix Tree Construction (Algorithm 2, Section IV-E), and State and Rule Encoding (Algorithm 3, Section IV-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. Note 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 captures 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 "***B". On the other hand, code$(N(s_9))$ = code$(\{s_4, s_8\})$ = "**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.

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.

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

To group 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[2]. 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 $\langle ** \cdots ** \text{CS}(s), x, \text{label}(s) \rangle$[3] 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 most of 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.

The following lemma captures the relation between the CS and LCS parameters (proof is in Appendix B).

Lemma 3 *For every state $s \in S$:*
(i) *LCS(s) is a suffix of label(s).*
(ii) *For every state $s' \in S$ for which there is a symbol $x \in \Sigma$ such that $\delta(s,x) = s'$, CS(s') is a suffix of LCS(s).*

E. Stage II: Common Suffix Tree

Next, we 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

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

[3] The number of "$*$" 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: $\mathrm{CS}(s) :=$ label(s) without the link symbol
4: **else**
5: $\mathrm{CS}(s) := \perp$ ▷ $\perp$ 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: $\mathrm{LCS}(s) := \mathrm{CS}(s')$ 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 $\ell_1, \ell_2 \in L$, ℓ_1 is an ancestor of ℓ_2 if and only if ℓ_1 is a suffix of ℓ_2
3: **for all** suffix tree node $\ell \in L$ **do**
4: **if** ℓ is not a leaf in the Common Suffix Tree **then**
5: $n :=$ number of node ℓ's children ▷ $n > 0$
6: add x *connecting nodes* as a children to ℓ 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** $\mathrm{LCS}(s)$ is a leaf in the Common Suffix Tree **then**
11: link s as a child of $\mathrm{LCS}(s)$
12: **else**
13: link s as a child of the connecting node of $\mathrm{LCS}(s)$ with the least number of children ▷ all children of an LCS are balanced between its connecting nodes
14: **end if**
15: **end for**

a suffix of ℓ_2 (Algorithm 2, Lines 1-3). Note that it suffices to consider only LCS values since the set L contains all the possible CS values, as captured by the following lemma (proof in Appendix B).

Lemma 4 *For any $s' \in S$, $CS(s') \in \{LCS(s) | s \in S\}$.*

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, to map a suffix rule to a prefix rule, all labels are written in a reverse order. For clarity, we label 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.

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 ℓ_1's children⌉ ▷ the number of bits required to enumerate over all outgoing edges of ℓ_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 ℓ of the Common Suffix Tree **do**
7: assign the code of ℓ as the concatenation of all edges' codes on the path between the ℓ and the root. Pad with 0 to fit width w.
8: **end for**
9: $S' := \{s \in S \mid \mathrm{CS}(s) = \perp\}$
10: **for all** states $s \in S \setminus (S' \cup \{s_0\})$ **do**
11: **add** the following rule to the rule set:
12: current-state set is the code of node $\mathrm{CS}(s)$
13: symbol is the label on the incoming link of s
14: next state is the code of node s
15: **end for**
16: **for all** states $s' \in S'$ **do**
17: **for all** $s \in S, x \in \Sigma$ such that $\delta(s,x) = s'$ **do**
18: **add** the following rule to the rule set:
19: current-state set is the code of node s
20: symbol is x
21: next state is the code of node s'
22: **end for**
23: **end for**
24: **add** the following rule for s_0:
25: current-state set is w bits of $\star$
26: symbol is $\star$
27: next state is the code of node s_o

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 $\mathrm{CS}(s)$ 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 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 $\langle \mathrm{code}(\mathrm{CS}(s)), x, \mathrm{code}(s) \rangle$, if there are more than one such edges; otherwise, we encode the

[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.

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 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 LCS(i) is a suffix of LCS(j) if and only if code(LCS(i)) is a prefix of code(LCS(j)). Hence, the correctness of this encoding is straightforward from the correctness of our wasteful intermediate encoding (of Section IV-D). This is formally captured by the next theorem, whose proof appears in Appendix B.

Theorem 5 *For every state $s \in S$ and symbol $x \in \Sigma$ such that $\delta(s,x) \neq s_0$, the* longest prefix match *rule of code(s) and x is either $\langle code(s), x, code(s_1) \rangle$ or $\langle code(CS(s_1)), x, code(s_1) \rangle$, where $s_1 = \delta(s,x)$. If $\delta(s,x) = s_0$, then the only rule matched is the default rule $\langle \star \ldots \star, \star, s_0 \rangle$.*

V. CompactDFA for total memory minimizations

Minimizing the number of rules increases the number of bits used to encode a state. For some applications, however, we 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 prefer using 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 VIII, 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).

Algorithm 4 Truncated Rule Encoding

```
1: for all leaf vertexes v of the Common Suffix Tree do
2:     if v's depth > D then                    ▷ v corresponds to a state
3:         attach v to its ancestor of depth D in the tree
4:     end if
5: end for
6: truncate the Common Suffix Tree at depth D
7: execute lines 1 − 9 of Algorithm 3
8: S' := S' ∪ {s ∈ S | CS(s) does not appear in the tree}
9: execute lines 10 − 27 of Algorithm 3
```

Algorithm 5 State Grouping. Calculating the family of sets H for a stride of k edges

```
1: H := ∅
2: for all state s ∈ S do
3:     if depth(s) > k and s is not an accepting state then
4:         S_s := {s' | there is a path of length k from s' to s}
5:         if |S_s| > 1 then
6:             H := H ∪ {S_s}
7:         end if
8:     else if depth(s) > k and s is an accepting state then
9:         for j = 1, ..., k do
10:            S_s(j) := {s' | there is a path of length j from s' to s}
11:            if |S_s(j)| > 1 then
12:                H := H ∪ {S_s(j)}
13:            end if
14:        end for
15:    end if
16: end for
```

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. Algorithm 4 depicts the pseudo-code of this algorithm.

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 VIII).

VI. CompactDFA for DFA with strides

One way to increase the speed of DPI using CompactDFA is to perform a lookup on a sequence of characters (a.k.a. *words* or *strides*) of the input. More specifically, we propose that each such stride will have a predetermined size of k thus potentially boosting up the performance by a factor of k. For example, considering the DFA in Fig. 1(a) with strides of at most $k = 2$, s_2 would have three additional edges that support strides with labels: BC, BA and CD to states s_5, s_9 and s_{11} respectivley. However, two problems arise: (i) When striding over the automaton, one might skip over *accepting states* within a stride. For that purpose, we propose to have variable

Algorithm 6 Rule Encoding for stride of length k

for all states s such that depth$(s) \leq k$ **do**
  add the following rule to the rule set:
    current-state set is w '$*$' symbols
    symbol is label(s), padded with $k-$depth(s) '$*$' symbols to the left.
    next state is the code of node s
end for
for all non-accepting states s with depth$(s) > k$ **do**
  if $S_s \in H$ **then** ▷ Set S_s as defined in Algorithm 5
    add the following rule to the rule set:
      current-state set is the code of S_s
      symbol is the last k symbols of label(s)
      next state is the code of node s
  else ▷ $S_s = \{s'\}$
    add the following rule to the rule set:
      current-state is the code of s'
      symbol is the last k symbols of label(s)
      next state is the code of node s
  end if
end for
for all accepting states s **do**
  for $j = 1,\ldots,k$ **do**
    if $S_s(j) \in H$ **then** ▷ $S_s(j)$ as defined in Algorithm 5
      add the following rule to the rule set:
        current-state set is the code of $S_s(j)$
        symbol is the last j symbols of label(s), padded with $k-j$ '$*$' symbols to the right.
        next state is the code of node s
    else ▷ $S_s(j) = \{s'\}$
      add the following rule to the rule set:
        current-state set is the code of s'
        symbol is the last j symbols of label(s), padded with $k-j$ '$*$' symbols to the right.
        next state is the code of node s
    end if
  end for
end for

size strides (of length less than k) for transitions that go to accepting states. Given the above example, such a transition would be represented by an edge from s_2 to the accepting state s_7 with label A. (ii) For states s whose depth d is smaller than k, there are several different words of size k that goes to s, however these words share the same suffix of length d, as states s_1, s_6 and s_{12} with suffixes E, B and C, respectively.

Given the following obstacles we first define a set H of sets that should be encoded (See Algorithm 5): For each set $s \in S$, let set S_s be the set of states from which there is a path of length k to s; in case s is an accepting state, we maintain k such sets, denoted $S_s(i)$, for each possible path length in $i \in \{1,\ldots,k\}$. For instance, using the above example based on the DFA in Fig. . 1 with strides of at most $k = 2$, the set $S_{s_{11}}(2)$ is $\{s_2, s_4, s_6, s_8\}$, and $S_{s_{11}}(1) = \{s_3, s_5, s_{10}\}$. H is the set of all such states whose size is greater than 1. In the above-mentioned example, $H = \{S, \{s_2, s_4, s_6, s_8\}, \{s_6, s_2\}, \{s_4, s_8\}, \{s_3, s_5, s_{10}\}, \{s_3, s_5, s_{10}, s_{12}\}\}$. Note that in this state grouping algorithm we assume that the shortest pattern is of length greater than k, implying that a state with depth at most k cannot be an accepting state.

Each state $S' \in H$ may be represented by the suffix that is common to all states in S'. Thus, H is viewed as a set of suffixes. For coding purposes we *attach* each state $s \in S$ to the minimal-size set $S' \in H$ such that $s \in S'$. Then, we build a common suffix tree (that is, a set S_1 is an ancestor of S_2 in the tree if and only if S_1 contains S_2) and encode it (as well as the states attached to it) using one of the methods proposed for suffix tree encoding (as in Algorithm 3). Finally, given the codes obtained for all sets $s \in S$ and $S' \in H$ and the code-width w, we derive the rules as depicted in Algorithm 6. The less '$*$' symbols in the current state field, the higher its priority; ties are broken by the position of the first non-'$*$' symbol in the symbol field: the further left the position is, the higher the priority is.

The input is the code of the current state and the first k characters. After applying each rule, we advance the input sequence by k' characters, where k' is the last proper index (that is, non-'$*$') in the symbol field of the matched rule. This implies that unless there is a match, the input sequence advances in k characters per lookup. The number of extra entries required is $(k-1)\cdot|F|$, where F is the set of accepting states (namely, 6 extra entries in the above-mentioned example).

VII. 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* [22]. 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 uses a 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. In addition, the second approach does not require to encode the symbol in the rule. Furthermore, hybrid solution may 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.

A. Implementing CompactDFA with non-TCAM IP-lookup solutions

One may wonder if applying IP-lookup solutions for pattern matching algorithms does not raise scalability issues. The cur-

rent 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 may split the solution to two components executed in parallel. The first component performs an exact-match search (which may 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 [24]. This recently-suggested chip requires on average 6 ns per lookup operation. Assuming that it takes 2 ns to resolve the exact-match value, we get a pattern-matching process with throughput of 1 Gbps. We note that some of the LPM solutions use pipeline architecture and hence have higher latency. In such cases, in order to cope with that latency, we need to use interleaving mechanisms, as we explain in details in Section VII-B.

B. Implementing CompactDFA with TCAM

One of the most promising off-the-shelf solutions for high-rate packet-classification and IP-lookup is the *Ternary Content-Addressable Memory (TCAM)* device, which consists of a table of fixed-width (yet configurable) entries, each of which is a sequence of symbols in the ternary alphabet {"0","1","*"}. The main functionality of the TCAM is a parallel matching of a key against all entries, where the result is the index of the first (that is, highest priority) matching entry. Current high-end TCAM devices work at rate of up to 500 MPPS (that is, perform 500M packet lookups per second) and store in their entries up to 72 Mb of data.

TCAM enables parallel matching of a key against all entries and thus provides high throughput that is unparalleled by software-based (or SRAM-based) solutions.

While TCAM has a high throughput, it has also an inherent high latency, which is an order of magnitude larger than its cycle time. For example, the cycle time of a 500 MHz TCAM (that performs 500 MPPS) is 2 ns, where its latency is typically over 100 ns [9][5]. The root cause of the high latency is that pipelined architecture is used to boost up the throughput of the TCAM devices, thus sacrificing the latency. This tradeoff is common in many other high-throughput hardware devices [12]–[14], [17].

[5]e.g., the partly-TCAM device NLA900 has a latency of 160 ns and PLUG has a latency of 148 ns [9].

The effect of the latency is negligible when implementing solutions that require only a single lookup per packet, such as in packet classification and IP lookups, where TCAMs were used traditionally. However, CompactDFA works on the payload of the packet assuming closed-loop lookups, where the input to a TCAM lookup depends on the result of a previous TCAM lookup. Thus, CompactDFA[6] cannot be efficiently implemented as-is due to the high latency of the device. A straightforward naïve implementation, which stalls the TCAM until the result of the lookup is available, yields high processing time per packet and a low throughput. Specifically, assuming a CompactDFA that works byte by byte, the processing time of such a solution is $\frac{L \cdot N}{R}$ seconds and its throughput is $\frac{8 \times R}{L}$, where L denotes the latency of the TCAM (in cycles), N is the number of payload 8-bit characters in the input, and R is the rate of the TCAM (in Hz),

A natural solution is to perform *inter-flow interleaving*, i.e., interleave queries from different flows. However, this solution increases the throughput (to $8R$) but does not improve the processing time per packet, as in the point of view of each flow, nothing had changed. Therefore, we suggest an *intra-flow interleaving* solution, where queries from different parts of the same flow are interleaved. This interleaving takes advantage of a delicate independence between different parts of the flow, which are harder to identify. We show that this solution reduces the processing time with negligible overhead in the throughput.

Specifically, the basic idea is to divide the packets to L equal size chunks and preform interleaving between the different chunks. For simplicity, we assume first that the chunk size $\lfloor N/L \rfloor$ is larger than some threshold T. We will later discuss how to choose the threshold and how to deal with packets of size less than $T \cdot L$.

Dividing the flow into chunks raises two different questions: (i) In which state to start scanning each chunk? (ii) How to detect patterns that span several chunks.

We propose the following simple solution, which is depicted in Algorithm 7: The scanning of each chunk starts at the initial state s_0 (Line 3). We continue scanning beyond the chunk boundaries until we are sure that there cannot be a pattern that starts at the current chunk and spans several chunks. (Lines 15–16).

We next discuss the stopping condition of our algorithm (Line 15): We continue scanning until we reach a state s whose depth $\text{depth}(s)$ is at most j, where j is the number of extra characters (from the consecutive chunk) scanned. The correctness is derived from the following basic lemma, which is a direct consequence of Properties 1 and 4.

Lemma 6 *Let $x_1, \ldots x_N$ be the input sequence, and let s_i be the state after scanning x_j. Then, $x_{j-depth(s_i)+1}, \ldots, x_j$ is a prefix of some pattern in the pattern set, and for any $j' \leq j - depth(s_i)$, $x_{j'}, \ldots, x_j$ is not a prefix of some pattern in the pattern set.*

[6]As well as other newly-proposed solutions that use closed-loop lookups with TCAMs, such as [2], [19], [30], and in some sense also [31].

Algorithm 7 Scanning chunk X whose characters sequence is $x_0, x_1, \ldots, x_n$. Denote by $y_1, y_2, \ldots y_\ell$ the characters after x_n in the flow. s_0 is the code of the initial state (10000 in the toy example).

Upon beginning of the scan of chunk X:

$scan\_index \leftarrow 0$
$within\_boundary \leftarrow true$
TCAM.lookup($s_0||x_0$)
▷ Apply TCAM lookup with key s_0 concatenated before x_0.

When the output register has a result i for chunk X:

$scan\_index++$
$entry \leftarrow$ The i-th entry of the external memory
if *entry.Match* **then**
  report matched pattern corresponding to entry i
end if
if *within_boundary* **then**
  TCAM.lookup($entry.NextState||x_{scan\_index}$)
  if $scan\_index == n$ **then**
    $scan\_index \leftarrow 0$; $within\_boundary \leftarrow false$
  end if
else
  if $entry.depth > scan\_index$ and $scan\_index \leq \ell$ **then**
    TCAM.lookup($entry.NextState||y_{scan\_index}$)
  else
    stop scanning chunk X
  end if
end if

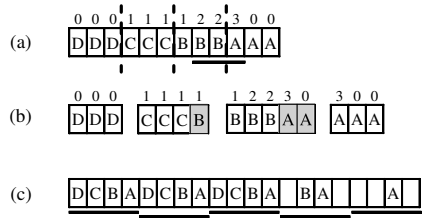


Fig. 2. Illustration of the intra-flow interleaving on a single packet against the pattern-set depicted in Fig. 1 (a) The packet is divided to $L = 4$ chunks of size 3, the underline indicates a matching: "BBA". The numbers above the packets are the depths of states reached in a DFA scan from the packet's beginning. (b) The chunks and the additional characters processed by our algorithm. The numbers above the packets are the state depths in a scan from the chunk's beginning. (c) The actual sequence of characters that the TCAM processes; each line below the figure is L consecutive cycles.

Lemma 6 implies that if the depth of the current state is d, then only the last d characters of the input are relevant for the matching. Thus, if $\text{depth}(s) \leq j$, then all the relevant characters are part of the consecutive chunk, and hence are detected in the scan of the consecutive chunk. Note that the algorithm requires storing additional information, which is the depth of each next state.

Fig. 2 illustrates the intra-flow interleaving. Specifically, Fig. 2(a) shows the entire packet, how it is divided into chunks, and what were the state depths had the scan was not divided; Fig. 2(b) shows the actual characters our algorithm scans in each chunk, including characters that are beyond the chunk boundary. Fig. 2(c) shows the characters as the stream of queries to the TCAM (the current state information is omitted for readability). Note that there are empty cycles towards the end of the stream, in which the TCAM stalls. These empty slots are due to the fact that the number of extra characters scanned by our algorithm varied from chunk to chunk. Notice that these cycles may be used for processing chunks of different flows, i.e., combining the inter-flow and intra-flow interleaving approaches, which are in fact orthogonal.

Let C be the average number of characters per chunk we scan twice due to the intra-flow interleaving technique (as shown in Section VIII, C is approximately 1.75). C does not depend on the size of the chunk or the size of the flow. Since we have L chunks, the total number of bytes scanned by our algorithm is $N + L \cdot C$; thus the processing time is at most $\frac{L+L\cdot C+N}{R}$ seconds, since in every cycle a character from the packet is analyzed, and the output is delayed by L cycles. When $C = 0$, we get the optimal processing time of $\frac{L+N}{R}$ seconds, while the worst-case processing time is $\frac{L+N(M-1)}{R}$, where M is the length of the longest pattern. The algorithm's throughput is $8R \cdot (1 - \frac{L\cdot C}{L\cdot C+N}) = 8R\frac{N}{N+LC}$ b/s, since $\frac{L\cdot C}{L+N}$ of the characters are scanned twice and they are the overhead of the algorithm. Again the smaller C is, the closer we get to the optimal throughput of $8R$ b/s.

Utilizing TCAM blocks: Modern TCAM devices allow applying K different keys simultaneously on different "blocks" of the entries (this is semantically equivalent to having K parallel TCAMs); typically K is around 4. This implies that, for large packets, the processing time can be further reduced by a factor of approximately K by performing simultaneous parallel processing of chunks in different K TCAM blocks (or parallel TCAMs). To fully utilize this solution, one need to divide the packet to $L \cdot K$ chunks, thus occurring over-head of C on each of them. In addition, this blocked solution requires that the TCAM entries are duplicated in each such block.

An alternative way to utilize the TCAM blocks is to selectively address one block at time. As we discussed earlier, we may partition the rules among the blocks by their next-symbol field. Note that TCAM 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. On the other hand, this solution significantly reduces the power consumption of the TCAM, which is often considered more important.

Finally, we note that both ways of utilizing TCAM blocks may be used simultaneously. In such a case, there is an hierarchy of blocks: *K super-blocks* contain the entire rule sets and work in parallel on different chunks of the packet; within each super-block, the entries are divided into blocks by their next symbol. At each cycle, K different keys are applied on the super-blocks, but within each super-block only one block is searched.

Dealing with Small Packets: Recall the throughput of the intra-flow interleaving approach is given by $\frac{N}{N+LC}8 \cdot R$. Thus, for small packet for which N is close to L there is a significant loss of throughput. The reason behind this is that we divide the

packet to L chunks no matter what its size is, and the overhead is C bytes per chunk. For example, for a small packet of size L, and a real-life value of $C = 1.75$, we lose about two thirds of the throughput.

Thus, we set up a threshold value T, which is the minimal chunk size. This ensures that the throughput is at least $\frac{N}{N+\lceil \frac{N}{T} \rceil C} \cdot 8R$. In the rest of the paper we use a threshold $T = 10$, implying a maximum loss of throughput of 15% for $C = 1.75$.

Notice that for packet of size less than $T \cdot L$, we end up with less than L chunks, and therefore, unlike large packets, small packet cannot utilize by itself the entire TCAM pipeline. This implies that in order to maximize the utilization *online scheduling algorithm* among the chunks is needed. Such algorithms are usually analyzed theoretically by considering their *competitive ratio*; namely, the worst-case ratio between their processing time to the processing time of an optimal offline algorithm that knows all future packet arrivals in advance.

The most straight-forward way to schedule the chunks is by taking a greedy approach and trying to minimize the time the last chunk of a packet completes. Algorithm 8 depicts the code for this greedy approach. Utilizing a pipeline of depth L is equivalent to schedule jobs to L machines, whose queues (and corresponding counters) are defined in Lines 1–2. The pipelined execution is realized by assigning a different starting time to each machine (Line 2). When a packet arrives, we first calculate the number of characters each machine handles (Lines 5–17). Then, we construct the specific chunk using these numbers (Lines 19–23). Note that to ensure the throughput guarantees, a special care should be taken so that all chunks will be at least of size T (see the conditions in Lines 7 and 10; and dealing with last leftover characters of a packet at Line 14). Notice that Algorithm 8 may result in variable-size chunks. Variable-size chunks increase the algorithm flexibility and therefore may improve its performance.

Our experiments shows that the algorithm works well in practice, however there are no theoretical bounds on its competitive ratio. Alternatively, one may map this scheduling problem to a *machine scheduling problem for aggregated links* [11]. For this problem, there is an online algorithm, whose objective is to minimize the maximum latency of a packet and has an optimal competitive ratio. The algorithm works in *epochs* in which at the beginning of each epoch the algorithm assigns greedily all the packets, currently existing in the system, to the different queues. During the epoch the algorithm deals only with assigned packets, and an epoch ends when there are no more such packets. The competitive ratio of the algorithm is $O(\sqrt{n/L})$ where n is to the maximum number of chunks assigned in any epoch [11]. We note that the design of the algorithm is aimed at enabling one to prove its performance guarantees; however, it is likely that in practice Algorithm 8 outperforms it.

Inspecting flows that span many packets: Up until now we discussed how to divide a *packet* into chunks. However, the data inspected by a DPI algorithm may span many packets, and one may wonder whether to perform intra-flow interleaving between chunks of different packets of the same flow.

Algorithm 8 The greedy approach for splitting up a packet to chunks and schedule the TCAM lookups.

```
1:  Let Q be array of L queues; initially all empty.
2:  Let B be array of L counters; initially B[i] = i.

    Upon arrival of packet P of size N:
                ▷ Let P = [p_0, ... p_{N-1}], where p_i is a character
3:  sched ← 0; last ← −1
4:  Let S be array of L counters; initially all 0
5:  while sched < N do
6:      v ← max_i B[i]; w ← min_i B[i]; j ← arg min_i B[i]
7:      if v − w + S[j] > T then
8:          S[j] ← S[j] + (v − w)
9:          sched ← sched + (v − w); last ← j
10:     else if sched < N − T then
11:         S[j] ← S[j] + T
12:         sched ← sched + T; last ← j
13:     else
14:         S[last] ← S[last] + N − sched
15:         sched ← N
16:     end if
17: end while
18: sched ← 0
19: for all Queues Q[i] in increasing order of counters B[i] do
20:     create chunk X of characters [p_sched, ..., p_{sched+S[i]}]
21:     enqueue X to Q[i]
22:     B[i] ← B[i] + S[i] · L; sched ← sched + S[i]
23: end for

    When processing chunk X at queue i completes:
24: Let C be the number of extra characters scanned
25: B[i] ← B[i] + C · L; dequeue chunk X from Q[i]

    At time slot t:
26: apply Algorithm 7 on the chunk at head of Q[t mod L]
```

Naturally, the challenge is not to miss patterns that lie on the boundaries between two consecutive packets.

We note that many network devices handle similar problem in the case of fragmentation and segmentation [10]. Most of the solutions require to store some state per flow. In our case, interleaving chunks of different packets would require to store a much more complex state on each flow, capturing the scanning state of the first and last characters of a packet; e.g. a straight-forward solution is to store information about the first and last $maxPattern - 1$ characters in each packet (where $maxPattern$ is to the maximum pattern length).

Dealing with fixed-width entries: 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 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.

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).

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

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

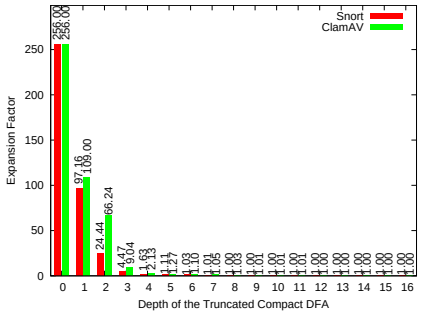


Fig. 3. The expansion factor of Snort's and ClamAV's DFAs.

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

VIII. EXPERIMENTAL RESULTS

We evaluate our CompactDFA technique only for the pattern matching process. We use two common pattern sets: Snort [23] (May 2009) and ClamAV [7] (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, unless table consolidation [19] is used, the optimal expansion factor is 1, while traditional DFA implementations have expansion factor of $|\Sigma|$. Fig. 3 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 that are multiple of 36 (recall Section VII-B); (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 may be divided into separate blocks (see Section VII-B). Table II presents the optimal depths and 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 set 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 may 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 result 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 VII-B), without strict balancing the rules among the blocks.

For ClamAV, CompactDFA uses a code width of 59 bits for achieving expansion factor of 1. 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.

We have compared the above results, using the same pattern sets, with the NIPS scheme presented by Weisenberg et al. [30]. We note that this work is based on an earlier work of Yu et al. [31] and provides significant improvements in terms of memory. For Snort pattern set NIPS uses 127914 rules, of width 192 symbol each. This translates to a TCAM memory usage of 2.93 MB, which is order of magnitude higher than CompactDFA; NIPS's SRAM memory usage is 0.1 MB, which is slightly lower. For the ClamAV pattern set, NIPS uses 613481 rules (of width 192 symbols each), yielding a TCAM memory footprint of 14.04 MB, which is almost double the memory usage of the best CompactDFA.

Next, we deal with CompactDFA with strides. Table III depicts the code width, number of rows, and memory usage required to implement various strides value for the Snort pattern set.[7] We considered only the CompactDFA implementation that aim at minimizing the number of entries, and has variable width entries (see Table II). As expected, both the code width and the number of entries increase as the stride increases. To test the effectiveness of this technique, we ran CompactDFA

[7]Recall that the maximum stride is bounded by the length of the shortest pattern; therefore, we have modified the pattern set to include only patterns of length 5 or more.

TABLE III
THE EFFECTS OF STRIDES ON THE MEMORY USAGE AND SPEEDUP.

Stride	Code width (bits)	Number of entries	TCAM size (MB)	SRAM size (MB)	Avg. stride	Std. Dev.	Min. observed avg. stride per packet
2	43	80468	0.56	0.41	1.999	0.03	1.897
3	45	86787	0.72	0.47	2.998	0.07	2.681
4	47	93106	0.88	0.52	3.996	0.11	3.434
5	48	99425	1.04	0.57	4.993	0.17	4.047

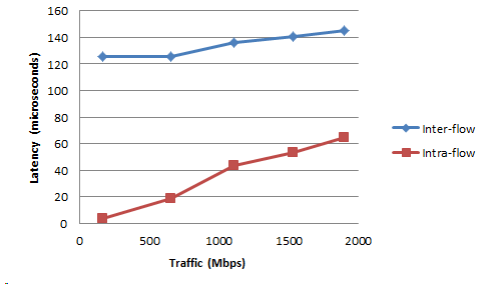


Fig. 4. The latency of inter-flow and intra-flow interleaving as function of traffic volume.

on traffic captured in the gateway of our university over the course of one day in November 2012. In all our experiments, the average stride taken by the DFA was within 99.9% of the maximum possible stride, and the standard deviation shows very high concentration around this value. This is explained by the fact that shorter than maximum strides are taken only when a match is found. We also report the minimum observed average stride in a packet in the trace. Evidently, even the processing of these rare worst-case packets gains a significant speedup as the maximum stride value is increased.

Moreover, we used another input traffic that was extracted from [8] to compare TCAM implementations with inter-flow and intra-flow interleaving. We assume that the TCAM rate is 500 MHz and its latency is 50 cycles.

Inter-flow interleaving is straightforward to implement and does not require any fine tuning of the algorithm. On the other hand, intra-flow interleaving requires a threshold T on the chunk size, so as to guarantee a minimal throughput (recall Section VII-B and Algorithm 8). The throughput, in turn, depends also on the per-chunk overhead C. Thus, we first calculated the distribution of C by using a trace as a training data (taken also from [8], yet different than the one used for the latency analysis). For the m-th character in the file, we found the minimal value j for which the DFA state after scanning the $(m+j)$-th character is of depth at most j; this value corresponds to the value of C had m was the last character in the chunk. Our experiments show that C is 1.75 on average, while for 96.54% of the characters it is at most 4 (this coincides with the fact that most of time the DFA is in a state with small depth). Accordingly, we defined the threshold to $T = 10$, yielding a maximum overhead of around 15%.

Fig. 4 depicts the latency (which include queuing time and processing time) as function of traffic volume. Specifically, we took another log file from [8] and run the log file several times simulating different volume of traffic. This is done by manipulating the arrival time of the packets by a constant factor: dividing all arrival times by K implies increasing the traffic rate by K. We focused on HTTP signatures and HTTP traffic. The average packet size in our logs was 1207 bytes, where most of the packets are of size 1514 bytes.

Fig. 4 shows that intra-flow interleaving is superior to inter-flow interleaving, especially in low loads. The experiment was performed on *TCAMimic* [5], a simulator we designed in order to simulate a real TCAM hardware.

In the logs we used, the gap between the approaches decreases as the load increases. The reason behind that is that our logs had less than L simultaneous flow. Recall that intra-flow aims to reduce the processing time and not the queuing delay of packets. Thus, its benefit is reduced in high load as the queueing delay becomes the dominant factor in the latency. On the other hand, in case there are less than L simultaneous flows, negligible queuing is needed when using inter-flow interleaving. We expect that as the number of these flows exceeds L, the inter-flow latency will increase significantly.

IX. CONCLUSIONS

This paper shows a reduction of the pattern matching problem to the IP-lookup problem. Our scheme, CompactDFA, gets as an input a DFA and produces compressed rule sets; each compressed rule defines the set of states that the rule applies 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 case.

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 achieve 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, achieving a total throughput of 10 Gbps and beyond.

ACKNOWLEDGMENT

The authors would like to thank Dr. Z. Guz, Dr. I. Keslassy and B. Zayith of the Knowledge Center on Chip MultiProcessors (CMP), Dept. Electrical Engineering, Technion for providing us computing resources. We would also like to thank A. Mor, which implemented and designed TCAMimic.

REFERENCES

[1] A. Aho and M. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, no. 6, pp. 333–340, 1975.

[2] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network IDS/IPS," in *IEEE ICNP*, 2006, pp. 187–196.

[3] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *ACM/IEEE ANCS*, 2007.

[4] A. Bremler-Barr, D. Hay, and Y. Koral, "CompactDFA: Generic state machine compression for scalable pattern matching," in *IEEE INFOCOM*, 2010, pp. 657–667.

[5] A. Bremler-Barr, D. Hay, and A. Mor, "TCAMimic," 2012. [Online]. Available: http://www.faculty.idc.ac.il/bremler/TCAMimic.htm

[6] A. Bremler-Barr, S. Tzur David, D. Hay, and Y. Koral, "Decompression-free inspection: DPI for shared dictionary compression over HTTP," in *IEEE INFOCOM*, Mar. 2012.

[7] ClamAV, 2010. [Online]. Available: http://www.calmav.net

[8] DARPA Intrusion Detection Data Sets, 1998. [Online]. Available: http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html

[9] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam, "PLUG: flexible lookup modules for rapid deployment of new protocols in high-speed routers," in *ACM SIGCOMM*, 2009.

[10] F. B. G. Varghese, J. A. Fingerhut, "Detecting evasion attacks at high speeds without reassembly," in *ACM SIGCOMM*, 2006.

[11] W. Jawor, M. Chrobak, and C. Dürr, "Competitive analysis of scheduling algorithms for aggregated links," *Algorithmica*, vol. 51, pp. 367–386, May 2008.

[12] W. Jiang and V. K. Prasanna, "Energy-efficient multi-pipeline architecture for terabit packet classification," in *IEEE GLOBECOM*, 2009.

[13] W. Jiang, Y. E. Yang, and V. K. Prasanna, "Scalable multi-pipeline architecture for high performance multi-pattern string matching," in *IEEE IPDPS*, 2010.

[14] W. Jiang, Q. Wang, and V. K. Prasanna, "Beyond TCAMs: an SRAM-based parallel multi-pipeline architecture for terabit IP lookup," in *IEEE INFOCOM*, 2008.

[15] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expression matching for deep packet inspection," in *ACM SIGCOMM*, 2006.

[16] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "HEXA: Compact data structures for faster packet processing," in *IEEE ICNP*, 2007.

[17] S. Kumar, M. Becchi, P. Crowley, and J. Turner, "CAMP: fast and efficient IP lookup architecture," in *ACM/IEEE ANCS*, 2006.

[18] T. Liu, Y. Sun, A. X. Liu, L. Guo, and B. Fang, "A prefiltering approach to regular expression matching for network security systems," in *ACNS*, 2011.

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

[20] D. Pao, W. Lin, and B. Liu, "Pipelined architecture for multi-string matching," in *Computer Architecture Letters*, vol. 7, no. 2, 2008, pp. 33–36.

[21] K. Peng, S. Tang, M. Chen, and Q. Dong, "Chain-based DFA deflation for fast and scalable regular expression matching using TCAM," in *ACM/IEEE ANCS*, 2011.

[22] M. Ruiz-Sanchez, E. Biersack, and W.Dabbous, "Survey and taxonomy of IP address lookup algorithms," *IEEE Network*, 2001.

[23] SNORT, 2010. [Online]. Available: http://www.snort.org

[24] H. Song, F. Hao, M. Kodialam, and T.-V. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards," in *IEEE INFOCOM*, 2009.

[25] T. Song, W. Zhang, D. Wang, and Y. Xue, "A memory efficient multiple pattern matching architecture for network security," in *IEEE INFOCOM*, April 2008, pp. 166 – 170.

[26] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim, "A multi-gigabit rate deep packet inspection algorithm using tcam," in *IEEE GLOBECOM*, 2005.

[27] L. Tan and T. Sherwood, "Architectures for bit-split string scanning in intrusion detection," *IEEE Micro*, pp. 110–117, 2006.

[28] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memoryefficient string matching algorithms for intrusion detection," in *IEEE INFOCOM*, 2004.

[29] J. van Lunteren, "High-performance pattern-matching for intrusion detection," in *IEEE INFOCOM*, April 2006, pp. 1–13.

[30] Y. Weinsberg, S. Tzur-David, D. Dolev, and T. Anker, "High performance string matching algorithm for a network intrusion prevention system (nips)," in *IEEE HPSR*, 2006.

[31] F. Yu, R. Katz, and T. Lakshman, "Gigabit rate packet pattern-matching using tcam," in *IEEE ICNP*, 2004.

[32] S. Yun, "An efficient TCAM-based implementation of multipattern matching using covered state encoding," *IEEE Trans. Comput.*, vol. 61, no. 2, pp. 213–221, Feb. 2012.

[33] K. Zheng, X. Zhang, Z. Cai, Z. Wang, and B. Yang, "Scalable nids via negative pattern matching and exclusive pattern matching," in *IEEE INFOCOM*, 2010.

APPENDIX A
PROPERTIES OF AHO-CORASICK- LIKE ALGORITHMS

In this section we give a formal definitions of the properties described in Section IV-C:

Property 1 *Let $P \subseteq \Sigma^*$ be the set of patterns that should be recognized by the DFA. Then, there is a mapping $\pi : F \mapsto 2^P$, such that for every accepting state $s \in F$ and a pattern $p \in \pi(s)$, if the DFA reached state s then p is a suffix of the inspected input. In addition, $label(s) \in \pi(s)$ and every pattern p with length larger than $depth(s)$ is not in $\pi(s)$.*

It is easy to prove that Property 1 is equivalent to demand that for every accepting state $s \in F$ and a pattern $p \in \pi(s)$ there is $i \leq \text{depth}(s)$ such that $p = \text{suffix}(s,i)$.

In addition, the DFA created by the Aho-Corasick Algorithm has the following property:

Property 2 *For any state $s \in S \setminus \{s_0\}$, there is a symbol $x \in \Sigma$, such that all incoming transitions to s are labeled with x (namely, for any $y \neq x$ and any $s' \in S$, $\delta(s',y) \neq s$). Specifically, $x = suffix(s,1)$.*

The next property implies that the DFA has no redundant states:

Property 3 *For every $s \in S \setminus F$ there is a path of forward edges to a state $s' \in F$.*

Finally, the determinism of the DFA yields the following property:

Property 4 *For every word $w \in \Sigma^*$ there is at most a single state s for which $label(s) = w$.*

APPENDIX B
OMITTED PROOFS

In this section we give formal proofs of the claims in the papers. We will use the formal definitions of the automata as described in Section II. In addition, we use the following function to capture the traversal of the DFA graph given an input sequence: $\hat{\delta} : S \times \Sigma^* \mapsto S$ generalizes the transition function δ to deal with a sequence of symbols (which we call a *word*), one by one. $\hat{\delta}$ is defined a recursive manner: $\hat{\delta}(s, x_0 x_1 \ldots x_n) = \hat{\delta}(\delta(s,x_0), x_1 \ldots x_n)$.

Proof of Lemma 1: Assume that there is a path w from s_0 to s for which $\text{suffix}(w,d) \neq \text{label}(s)$. Let s_1 be the intermediate state along this path d symbols before it ends (that is, $\hat{\delta}(s_0, \text{prefix}(w,|w|-d)) = s_1$ and $\hat{\delta}(s_1, \text{suffix}(w,d)) = s$). In addition, let $f \in F$ be the accepting state which is reached from s by forwarding edges (such a state exists, by Property 3) and let w' be the symbols along these forwarding edges (that is, $\hat{\delta}(s,w') = f$). Hence, by Property 1, $\text{label}(f) = \text{label}(s)w' \in \pi(f)$. On the other hand, $\hat{\delta}(s_0, ww') = \hat{\delta}(s_1, \text{suffix}(w,d)w') = \hat{\delta}(s,w') = f$, but $\text{label}(s)w' \in \pi(f)$ is not a suffix of the inspected input ww' since $\text{suffix}(w,d) \neq \text{label}(s)$, and hence a contradiction to Property 1. ∎

Proof of Lemma 2: Assume that there is a state s_2 such that $\text{label}(s_2)$ is a suffix of $\text{label}(s)x$ and $\text{label}(s_2)$ is longer than $\text{label}(s_1)$. Let $f_2 \in F$ be the accepting state which is reached from s_2 by forwarding edges, and let w_2 be the symbols along these edges (f_2 exists by Property 3). Furthermore, denote by $s_3 = \hat{\delta}(s_1, w_2)$; note that $\text{depth}(s_3) \leq \text{depth}(s_1) + |w_2| < \text{depth}(s_2) + |w_2| = \text{depth}(f_2)$.

Consider the input sequence $\text{label}(s)xw_2$. On one hand, $\text{label}(s_2)w_2$ is a suffix of the input sequence, $\text{label}(s_2)w_2 = \text{label}(f_2)$, and therefore $\text{label}(s_2)w_2 \in \pi(f_2)$ (Property 1). This implies that the DFA matches the pattern $\text{label}(s_2)w_2$. On the other hand, $\hat{\delta}(\text{label}(s)xw_2) = \hat{\delta}(s, xw_2) = \hat{\delta}(s_1, w_2) = s_3$ and $\text{depth}(s_3) < |\text{label}(s_2)w_2|$ implying that $\text{label}(s_2)w_2 \notin \pi(s_3)$ (Property 1). Thus, the DFA does not match the pattern $\text{label}(s_2)w_2$, which is a contradiction. ∎

Proof of Lemma 3: If $\text{LCS}(s) = \perp$ the claims hold triviality.

Let s_1 the states the determines $\text{LCS}(s)$ (that is, $\text{LCS}(s) = \text{CS}(s_1)$, see Algorithm 1, Line 10), and let x be the symbol such that $\delta(s,x) = s_1$. Note that $\hat{\delta}(s_0, \text{label}(s)x) = \hat{\delta}(s,x) = s_1$, thus by Lemma 1, $\text{suffix}(\text{label}(s)x, \text{depth}(s_1)) = \text{label}(s_1)$. This implies that the last symbol of $\text{label}(s_1)$ is x, and therefore $\text{CS}(s_1)$ (that is, $\text{label}(s_1)$ without its last symbol) is equal to $\text{suffix}(\text{label}(s), \text{depth}(s_1) - 1)$, proving *(i)*.

The proof of *(ii)* follows immediately, since for all $s' \in S$ with an edge from s, $\text{CS}(s')$ is a suffix of $\text{label}(s)$. Since $\text{LCS}(s)$ is the longest such suffix, it implies that $\text{CS}(s')$ must be a suffix of $\text{LCS}(s)$. ∎

Proof of Lemma 4: Assume s' has more than one incoming link and consider a *forward transition* from state $s \in S$ to state $s' \in S$ with symbol $x \in \Sigma$. By the definition of the labels and since we consider a forward transition, $\text{label}(s') = \text{label}(s)x$. Hence, by the definition of CS, it implies that $\text{CS}(s') = \text{label}(s)$. Furthermore, by LCS definition, $\text{LCS}(s) = \text{CS}(s')$, since all other outgoing edges from s reaches states with depth at most $\text{depth}(s) + 1 = \text{depth}(s')$, implying their CS is shorter or equal to $\text{CS}(s')$. This implies, $\text{CS}(s') \in \{\text{LCS}(s) | s \in S\}$.

If s' has only one incoming edge that $\text{CS}(s') = \perp$. However, $\text{LCS}(s_0) = \perp$ by definition, and the claim follows. ∎

Proof of Theorem 5: We distinguish between several cases according to the type of the DFA transition $\delta(s,x)$ (see Section II).

If $\delta(s,x)$ is a forwarding transition and s_1 has only one incoming edges, then by Algorithm 3 the rule $\langle \text{code}(s), x, \text{code}(s_1) \rangle$ is in the rule-set, and clearly it is the longest prefix match for $\text{code}(s)$ and x since its first field has no $\star$ symbols.

If $\delta(s,x)$ is a forwarding transition and s_1 has more than one incoming edge, then, by definition, $\text{label}(s_1) = \text{label}(s)x$ and thus $\text{CS}(s_1) = \text{label}(s) = \text{LCS}(s)$. By the construction of the Common Suffix Tree, state s is linked (possibly via a connecting node) to a node v corresponding to $\text{LCS}(s) = \text{CS}(s_1)$. Therefore, among all nodes, $\text{code}(v)$ has the longest prefix match to $\text{code}(s)$. Since $\text{code}(\text{CS}(s_1)) = \text{code}(v)$, it implies that the rule $\langle \text{code}(\text{CS}(s_1)), x, \text{code}(s_1) \rangle$ has the longest prefix match to $\text{code}(s)$ and symbol x.

If $\delta(s,x)$ is a cross transition then Lemma 3 implies that $\text{CS}(s_1)$ is a suffix of $\text{LCS}(s)$. Thus, the node corresponding to $\text{CS}(s_1)$ is an ancestor of the node corresponding to $\text{LCS}(s)$ implying that $\text{code}(\text{CS}(s_1))$ is a prefix of $\text{code}(\text{LCS}(s))$ and thus also of $\text{code}(s)$. This immediately implies that the rule $\langle \text{code}(\text{CS}(s_1)), x, \text{code}(s_1) \rangle$ is matched by $\text{code}(s)$ and x.[8]

It remains to show that this rule is the *longest prefix match* rule. Assume that there is a state s_2 such that: *i)* the rule $\langle \text{code}(\text{CS}(s_2)), x, \text{code}(s_2) \rangle$ is in the rule set; *ii)* $\text{code}(\text{CS}(s_2))$ is longer (in term of 0-1 symbols, excluding $\star$'s) than $\text{code}(\text{CS}(s_1))$; and *iii)* $\text{code}(s)$ matches $\text{code}(\text{CS}(s_2))$. By the structure of the tree, $\text{code}(s)$ matches only nodes on the path between $\text{LCS}(s)$ and the root. This implies that both $\text{CS}(s_2)$ and $\text{CS}(s_1)$ are on that path, and since $\text{code}(\text{CS}(s_2))$ is longer, $\text{CS}(s_1)$ is an ancestor of $\text{CS}(s_2)$ in the Common Suffix Tree. This in turn implies that $\text{CS}(s_1)$ is a suffix of $\text{CS}(s_2)$, which is either equal to or a suffix of $\text{LCS}(s)$. Since $\text{LCS}(s)$ is a suffix of $\text{label}(s)$ then both $\text{CS}(s_2)$ and $\text{CS}(s_1)$ are suffixes of $\text{label}(s)$, implying that both $\text{label}(s_1)$ and $\text{label}(s_2)$ are suffixes of $\text{label}(s)x$, and $\text{label}(s_2)$ is longer. This contradicts Lemma 2 that states that $\text{label}(\delta(s,x)) = \text{label}(s_1)$ is the longest possible suffix of $\text{label}(s)x$. ∎

[8] if $s_1 = s_0$ then the rule which is matched is $\langle \text{code}(\text{CS}(s_0)), \star, \text{code}(s_0) \rangle$.