A DFA with Extended Character-set for Fast Deep Packet Inspection

Cong Liu[†] Ai Chen[‡] Di Wu[†] Jie Wu[§]
[†]Department of Computer Science, Sun Yat-sen University
[‡]Shenzhen Institutes of Advanced Technologies, Chinese Academic of Science
[§]Department of Computer and Information Sciences, Temple University

***Abstract*—Deep packet inspection (DPI), based on regular expressions, is expressive, compact, and efficient in specifying attack signatures. We focus on their implementations based on general-purpose processors that are cost-effective and flexible to update. In this paper, we propose a novel solution, called deterministic finite automata with extended character-set (DFA/EC), which can significantly decrease the number of states through slightly extending the character-set. Different from existing state reduction algorithms, our solution requires only a single memory access for each byte in the traffic payload, which is the minimum. We perform experiments with the Snort rule-sets. Results show that, compared to DFA, a DFA/EC can be over four orders of magnitude smaller, has smaller memory bandwidth, and runs faster. We believe that DFA/EC will lay a groundwork for a new type of state compression technique in fast packet inspection.** [1]

***Index Terms*—Deep packet inspection, regular expression, deterministic finite automata, extended character-set.**

I. Introduction

Deep packet inspection (DPI) processes packet payload content in addition to the structured information in packet headers. DPI is becoming increasingly important in classifying and controlling network traffic. Well-known internet applications of DPI include: network intrusion detection systems that identify security threats given by a rule-set of signatures, content-based traffic management that provides quality of service and load balancing, and content-based filtering and monitoring that block unwanted traffic. Due to their wide application, there is a substantial body of research work [1], [2], [3], [4], [5] on high-speed DPI algorithms in the literature, in which different automata for single-pass high-speed inspection are proposed based on either software or hardware implementations.

Traditional packet inspection algorithms have been limited to comparing packets to a set of strings. Newer DPI systems, such as Snort [6], [7] and Bro [8], use rule-sets consisting of regular expressions, which are more expressive, compact, and efficient in specifying attack signatures. Hardware-based approaches exploit parallelism and fast on-chip memory, and are able to create compact automata. However, it is more cost-effective and flexible to update when small on-chip lookup engines or general-purpose processors are used together with automata stored in off-chip commodity memory. In this paper, we focus on a general-purpose processor approach.

The throughput of general-purpose processor approaches are limited by memory bandwidth. Therefore, it is critical to minimize the number of memory (off-chip memory) accesses for each byte in the traffic payload. Implementations of regular expressions, such as *non-deterministic finite automata* (NFAs), have a non-deterministic number of memory accesses per byte. Another critical issue is to reduce the size of the automata stored in memory in order to reduce the cost of memory, improve the scalability for a larger number of rules, and increases the inspection speed (with the use of cache memory). While *deterministic finite automata* (DFAs) implementations of regular expressions take only one memory accesses per byte, they often require a very large memory capacity, which undermines their scalability in real applications. Therefore, conventional DFA and NFA are impractical in real systems.

Recent research effort has been focused on reducing the memory requirement of DFAs, and they can be divided into the following categories: (1) reducing the number of states [1], [9], [10], (2) reducing the number of transitions [2], (3) reducing the bits encoding the transitions [3], [11], and (4) reducing the character-set [12]. Unfortunately, all of these approaches compress DFAs at the cost of increased memory accesses. The amount of compression in transition reduction and character-set reduction is bounded by the size of the character-set since there is at least one transition in each state. We focus on state reduction, a more potential approach in reducing memory requirement, and our approach can incorporate the other approaches to achieve further memory reduction.

We propose a novel state reduction solution, called *deterministic finite automata with extended character-set* (DFA/EC). We first introduce DFA/EC as a general model of DFA, which removes part of a DFA state and incorporates this part with the set of input characters into the extended character set. However, simply doing this cannot reduce the size of the transition table since the increase in the size of the extended character set can be more significant than the decrease in the number of states. Our main contribution is an efficient implementation of DFA/EC, which contains an encoding method. This encoding method encodes the part of

[1]This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grants 61003241,61003242, 61003296, Natural Science Foundation of Guangdong Province (10451027501005630), Fundamental Research Funds for the Central Universities (09LGPY56), Doctoral Fund of Ministry of Education of China (20100171120047), SRF for ROCS, SEM (2011-508), National S&T Major Project of China under Grant No. 2011ZX03005-001, Guangdong S&T Major Project under Grant No. 2009A080207002, Shenzhen Fundamental Research Program under Grant No. JC201005270332A, and by the National Science Foundation (NSF) under Grants CCF 1028167, CNS 0948184 and CCF 0830289.

TABLE I
NOTATIONS USED IN THIS PAPER.

Notation	Meaning	Note
$N(D)$	The set of NFA(DFA) states	$D \subset 2^N$
N_1	The set of main states of a DFA/EC	$N_1 \cup N_2 = N$
N_2	The set of complementary states	$N_1 \cap N_2 = \phi$
D_1	The set of main DFA states	$D_1 = 2^{N_1}$
D_2	The set of complementary program states	$D_2 = 2^{N_2}$
T^e	The transition function of the main DFA	
$C(C^e)$	The original (extended) character-set	$C^e = C \times \mathbb{B}$
$H^e\&A$	The complementary program	

the removed DFA state into a single bit so that the the size of the extended character set merely doubles as the number of states drops by orders of magnitude. The main contributions of this paper are summarized as follows:

1) We introduce DFA/EC, a general DFA model that incorporates partial DFA state into the set of input characters.
2) We provide an efficient implementation of inspection program based on our DFA/EC model, which results in a compact transition table and a fast inspection speed.
3) We prove that DFA/EC is equivalent to DFA.
4) We perform extensive evaluation to compare DFA/EC with related algorithms with the Snort rulesets.

Compared with existing state reduction algorithms, DFA/EC significantly increases inspection speed by keeping the number of per-byte memory access to one, which is the minimum. The size of our inspection program is also small enough to be stored entirely in the cache memory. Evaluations with Snort rule-sets demonstrate that DFA/EC can be very compact and achieves high inspection speed. Specifically, DFA/EC can be over four orders of magnitude smaller than DFA, and also has a smaller memory bandwidth than DFA. It is also significantly smaller than MDFA [1], and runs faster than DFA and MDFA. The advantages of DFA/EC are summarized in the following:

1) DFA/EC requires only one memory access for each byte in the packet payload, while significantly reducing memory in terms of table size.
2) DFA/EC is conceptually simple, easy to implement, and easy to update due to fast construction.
3) DFA/EC can be combined with other approaches to provide a better level of compression.

The rest of this paper is organized as follows: Related work is briefly covered and compared in Section II. Section III introduces the concept of DFA/EC with a motivating example. Section IV presents the formal model of DFA/EC and the its DFA-equivalency condition. Section Vdescribes our efficient implementation of DFA/EC and prove its DFA equivalency. Section VI, evaluates DFA/EC using the Snort rule-sets and synthetic traffics. Section VII concludes the paper. Notations used are summarized in Table I.

II. RELATED WORK

Prior work on regular expression matching at line rate can be categorized by their implementation platforms into FPGA-based implementations [13], [14], [15], [16], [17] and general-purpose processors and ASIC hardware implementations [1], [2], [9], [10], [18], [19].

Existing DFA state compression techniques (e.g. MDFA [1], HFA [9], XFA [20]) and transition compression techniques (e.g. D^{2}FA [2], CD^{2}FA [18]) effectively reduce the memory storage but introduce additional memory access per byte.

Delayed Input Deterministic Finite Automata (D^{2}FA) [2] uses default transitions to reduce memory requirements. If two states have a large number of transitions in common, the transition table of one state can be compressed by referring to that of the other state. Unfortunately, when a default transition is followed, memory must be accessed once more to retrieve the next state.

Using auxiliary variables and devising a compact and efficient inspection program is challenging and is most related to our work. Two seminal papers [9], [20] use auxiliary variables to represent the "factored out" auxiliary states and reduce the DFA size. However, the auxiliary variables are manipulated by auxiliary programs associated with each state or transition, resulting in extra memory accesses to obtain the auxiliary programs in addition to the state indexes. Secondly, H-FA [9] uses conditional transitions that require a sequential search. Moreover, the number of conditional transitions per character can be very large in general rule-sets, which results in a large transition table and a slow inspection speed. XFA uses several automata transformations to remove conditional transitions. However, to preserve semantics, XFA is limited to one auxiliary state per regular expression, which is unsuitable for complex regular expressions. On the other hand, EFA/EC uses a single piece of program to generate its extended characters, and it requires a single memory access for each byte in the payload.

Hybrid-FA [9], [10] prevents state explosion by performing partial NFA-to-DFA conversion. The outcome is a hybrid automaton consisting of a head-DFA and several tail-automata. The tail-automata can be NFA or DFA. However, maintaining multiple DFA/NFA may introduce a large per-flow state and scarify inspection speed. In [10], a character set is expanded to represent conditional transitions. However, they used alphabet compression [12] to compress the character set, which cannot effectively reduce the size of the expanded character set when there are multiple conditions on the transitions. Differently, DFA/EC is a general model of DFA proposed to compress transition table size. More importantly, we propose a new encoding method to limit the extended character set to twice its size, which is the key to make our DFA/EC model practical.

CompactDFA [3] and HEXA [11] compress the number of bits required to represent each state, but they are only applicable to exact string matching. Alphabet compression [12] maps the set of characters in an alphabet to a smaller set of clustered characters that label the same transitions for a substantial amount of states in the automaton.

Recent security-oriented rule-sets include patterns with advanced features, namely bounded repetitions and back-references, which add to the expressive power of traditional regular expressions. However, they cannot be supported

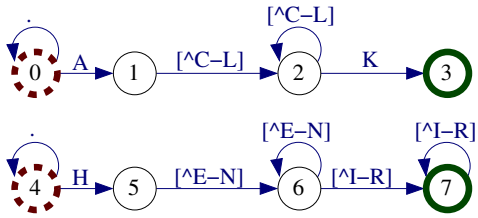


Fig. 1. The NFAs for ".*A[^C-L]+K" and ".*H[^E-N]+[^I-R]+", respectively.

through pure DFAs [10], [21]. The bounded repetition, or counting constraint, is a pattern that repeats a specific number of times. The back-reference [5] is a previously matched substring that is to be matched again later. DFA/EC can be extended to support the above features in regular expressions using the techniques in [9], [20]. We omit these advanced features in this work for simplicity.

III. Overview of DFA/EC

In this section, we will first review some preliminaries on automata used in packet inspection, i.e. NFA and DFA. Then, we provide a motivating example of DFA/EC.

A. Preliminaries

A *regular expression* describes a pattern of strings. Features of regular expressions that are commonly used in network intrusion detection systems include exact match strings, character-sets, wildcards, and repetitions. As an example throughout this paper, we use a rule-set consisting of two regular expressions: ".*A[^C-L]+K" and ".*H[^E-N]+[^I-R]+". An exact match substring, such as "C", is a pattern that occurs in the input text exactly as it is. Character-sets, such as "[E-N]", matches any character between "E" and "N", and "[^E-N]" is the complement of "[E-N]". A wildcard "." is equal to "[^]" and matches any character. Repetition ".*" matches any strings with a length from zero to infinity, and repetition "[^E-N]+" matches nonempty strings containing characters in "[^E-N]". For instance, pattern "H[^E-N]+[^I-R]+" matches string "HAT" and "HADST".

NFA and DFA are popular pattern matching programs for a set of one or more regular expressions. Figure 1 shows the NFAs accepting the example regular expressions. In NFAs, the number of states is not greater than the number of characters in the regular expressions in the rule-set, even when the regular expressions contain repetitions and character-sets. States 0 and 4 are initially active, and a match is reported when any accepting state, e.g. 3 and 7, is active. In NFAs, multiple states can be active simultaneously, and multiple memory accesses are required to obtain the next transitions for all active states. The sequence of the sets of active states when the example NFA matches string "HAT" is:

$$(0,4) \xrightarrow{H} (0,4,5) \xrightarrow{A} (0,1,4,6) \xrightarrow{T} (0,4,6,\underline{7})$$

A DFA can be constructed from NFAs using the subset construction routine, in which a DFA state is created to represent a set of NFA states that can be simultaneously active in some matching process. Therefore, the number of DFA states can be very large. Although, in practice, indexes are assigned to DFA states to reduce space, *we will regard a DFA state as a set of NFA states* in this paper. Let N be the set of NFA states, D be the set of DFA states, and we have (1) $d \subseteq N$, for any DFA state $d \in D$, (2) $D \subset 2^N$ (the power set of N), and (3) usually $|N| \ll |D| \ll |2^N| = 2^{|N|}$. $|N| \ll |D|$ is usually true due to the *state explosion problem*. For example, the DFA (which is not shown in this paper) constructed for the example NFA with 8 states contains 18 states. On the other hand, $|D| \ll 2^{|N|}$ because not all combinations of NFA states can be simultaneously active.

TABLE II
The cases of each NFA state duplicating DFA states and the scores for the NFA states in Figure 1.

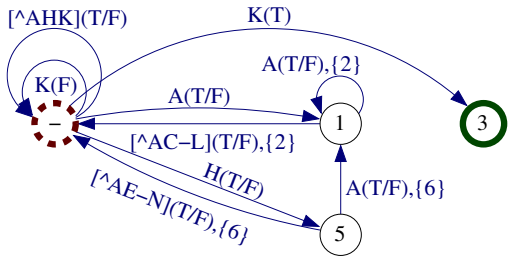
state	0	1	2	3	4	5	6	7
cases	0	7	7	1	0	2	7	8
scores	0	1	246	1	0	1	246	491

B. Motivation and overview

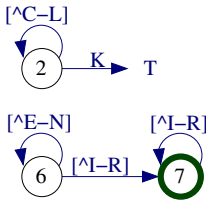
Methods to resolve the state explosion problem have been discussed in [1], [9], [20]. NFA states corresponding to the repetitions of large character-sets, such as states 2, 6, and 7 in our example NFA, cause state explosion. The explanations are that (1) these states are more likely to be active, and (2) a frequently active NFA state is more likely to be active simultaneously with other sets of states, which consequently increases the number of simultaneously active sets of NFA states, i.e. the number of DFA states. For example, state (0,1,4) is in D, and the frequently active NFA state 2 duplicates it and adds another state (0,1,2,4) into D. In Table II, for each states n in our example NFA (Figure 1), we show the number of cases where there exists a $d \in D$ that satisfies $d \cap \{n\} \in D$.

To reduce DFA size, we propose a novel method, called *DFA with extended character-set* (DFA/EC). In a DFA/EC, we select some of the most frequently active NFA states and incorporate them into the character-set (or the alphabet) of the DFA to form a slightly larger *extended character-set*. There is a *main DFA* in DFA/EC that implements the rest of the infrequently active NFA states and, therefore, the main DFA has a small number of states. We call those NFA states that are selected and incorporated into the character-set the *complementary states* (N_2); and we call the remaining NFA states the *main states* (N_1). As we will see in Section V, we have constraints which exclude some of the frequently active NFA states from the set of complementary states in order to enable the single-bit encoding of the complementary states in the extended character set and the efficient DFA/EC implementation.

While the main DFA represents and processes the main states, we call the remaining functionality in the DFA/EC the *complementary program*, which deals with the complementary states. The challenge in the design of DFA/EC is in the selection of a proper implementation such that the complementary



(a) Main DFA



(b) complementary states

Fig. 2. The DFA/EC for ".*A[^C-L]+K" and ".*H[^E-N]+[^I-R]+".

program is very fast and that the main states N_1 can be implemented by a compact main DFA whose size, ideally, is equal to $|N_1|$.

In our implementation (see Section V), the main DFA of the DFA/EC can be smaller than the corresponding conventional DFA by over four orders of magnitude while the extended character-set only doubles the size of the original character-set. When a DFA/EC consumes a byte, the complementary states are processed and the extended character-set is generated efficiently by only a few instructions without any memory access (Section V). Experiment results in Section VI show that DFA/EC can be the fastest among the automata in comparison, including DFA.

C. An illustration example of DFA/EC

From the NFAs in Figure 1, we construct a DFA/EC, which is shown in Figure 2. Since we have not presented how to select complementary states, we simply assume that the complementary states are the NFA states 2, 6, and 7. The main DFA constructed from the main states is shown in Figure 2(a). For clarity, we make the following simplifications in this figure: (1) in each DFA state, we remove the NFA states 0 and 4, which are always active in the state labels: state "1" should actually be labeled with "0,4,1" and states "-" should actually be labeled with "0,4", and (2) some transitions to state "-", "1", and "5" are removed. From this example, we can see that the DFA/EC, which has only 4 states in its main DFA, is very compact compared to the corresponding conventional DFA (not shown) that has 18 states.

In our implementation, the extended character-set includes the original character-set and an extra bit. This extra bit represents a boolean value, which is encoded from the complementary states. For example, the label "K(T)" on the transition from state "-" to "3" indicates that the transition is taken when the next byte is "K" and the extra bit is true.

We require that the transitions in the main DFA can make some complementary states active. For example, the transition labeled by "[^AE-N](T/F),{6}" from states "-" to "5", which is taken when the next byte is in the character-set "[^AE-N]" and the extra bit is either true or false, makes the complementary state "6" active.

In Figure 2(b), the complementary states are temporarily shown as an NFA, which is to be replaced by our efficient implementation in Section V. One of the transitions from state 2 does not make any state active, but it sets the extra bit in the extended character-set to true.

A DFA/EC maintains two states, one for the main DFA, and another for the complementary program. In our example, we represent the initial DFA/EC states as (-){}. For each byte in the payload, the DFA/EC functions as follows: (1) It calculates the extra bit for the extended character using the complementary program. (2) It calculates the new main DFA state using the current main DFA state and the extended character. (3) It calculates the new state of the complementary program using the current state of the complementary program together with the label on the main DFA transition.

As an example, the sequence of the sets of active states when the example DFA/EC matches string "ABK" is:

$$(-)\{\} \xrightarrow{A} (1)\{\} \xrightarrow{B} (-)\{2\} \xrightarrow{K} (\underline{3})\{\}$$

Initially, the state of the main DFA is (-) and that of the complementary program is {}. For the first input character 'A', the extra bit is F (since no complementary state is active), the transition labeled by "A(T/F)" from states "-" to "1" in the main DFA is token, and no transition in the complementary program is token (since no complementary state is active). So, the resulting DFA/EC state is "(1){}". For the second input character 'B', the extra bit is F, the transition labeled by "[^AC-L](T/F),{2}" from states "1" to "-" in the main DFA is token, and no transition in the complementary program is token. So, the resulting DFA/EC state is "(-){2}". For the third input character 'K', the extra bit is set to T since the transition labeled by 'K' in (the NFA representation of) the complementary program is token, and the transition labeled by "K(T)" from states "-" to "3" in the main DFA is token. So, the resulting DFA/EC state is "($\underline{3}$){}".

In the above example, we have illustrated how the main DFA and the complementary states interact with each other. We will define a formal model for DFA/EC and show the equivalence between DFA/EC and DFA in Section IV. An efficient implementation of DFA/EC will be presented in Section V.

IV. The Formal Model of DFA/EC

This section presents a formal model of DFA/EC and discusses the correctness of DFA/EC in terms of its equivalence to a DFA.

A DFA/EC is a novel model of automata that generalizes the conventional DFA. We denote $D_1 \subset 2^{N_1}$ as the set of simultaneously active sets of main states, $D_2 \subset 2^{N_2}$ as the set of simultaneously active sets of complementary states, C

as the original character-set (or alphabet), C^e as the extended character-set, and D as the set of conventional DFA states. For any state d in a DFA, a semantically equivalent DFA/EC has a corresponding state $\{d_1, d_2\}$, such that $d_1 \in D_1$, $d_2 \in D_2$, and $d_1 \cup d_2 = d$. Here, d_1 is a state of the main DFA and d_2 is a state of the complementary program. That is, $D_1 = \{d_1 | d_1 = d \cap N_1, d \in D\}$ is the set of main DFA states and $D_2 = \{d_2 | d_2 = d \cap N_2, d \in D\}$ is the set of complete program states. A DFA/EC can be defined by $(D_1, D_2, C, C^e, H^e, T^e, A)$, where:

$$\begin{cases} H^e: & D_2 \times C \to C^e \\ T^e: & D_1 \times C^e \to D_1 \times D_2 \\ A: & D_2 \times C \to D_2 \end{cases} \quad (1)$$

For each byte $c \in C$ in the packet payload, a DFA/EC with its current state being $\{d_1, d_2\}$ functions as the following : (1) Function H^e generates an extended character $c^e \in C^e$ using d_2 and c. (2) With d_1 and c^e, function T^e generates a new partial state $\{d_1, d_2'\}$ where d_1 is the new state of the main DFA. (3) With d_2 and the original character c, function A generates a new partial state d_2'', and $d_2 = d_2' \cup d_2''$ is the new state of complementary program. In our implementation, T^e is the transition function of the main DFA, which is implemented by a transition table and a lookup function. H^e and A together form the complementary program, which is implemented by several efficient instructions without any memory access.

Theorem 1 (The equivalence between DFA and DFA/EC): For any DFA, there exists an equivalent DFA/EC.

Proof: If we let $T : D \times C \to D$ be the transition function of a DFA, we need to prove that for any DFA, there is an equivalent DFA/EC, as defined by Equation 1. First, the DFA defined by (D, C, T) can be equivalent to another form of DFA $(D_1, D_2, C, T_{11}, T_{12}, T_{21}, T_{22})$, with $D_1 \cap D_2 = \phi$, $D_1 \cup D_2 = D$, and $T_{11}, T_{12}, T_{21}, T_{22}$ being transition functions:

$$\begin{cases} T_{11}: & D_1 \times C \to D_1 \\ T_{12}: & D_1 \times C \to D_2 \\ T_{21}: & D_2 \times C \to D_1 \\ T_{22}: & D_2 \times C \to D_2 \end{cases} \quad (2)$$

The equivalence holds, as long as, for any $d \in D$, there exists $d_1 \in D_1$, $d_2 \in D_2$ and $d_1 \cup d_2 = d$, such that $T_{11}(d_1, c) \cup T_{12}(d_1, c) \cup T_{21}(d_2, c) \cup T_{22}(d_2, c) = T(d, c)$. Here, we regard d_1 and d_2 as sets of simultaneously active NFA states and T_{11} is a transition function, which returns the set of new active NFA states in D_1 that is made active through transitions from the set of previously active states $d_1 \in D_1$ on character c. Equally, T_{12},T_{21}, and T_{22} are transition functions that return the sets of new active NFA states in D_2, D_1, D_2 that are made active through transitions from the sets of previously active NFA states in D_1, D_2, D_2, respectively. Obviously, these functions exist and can be easily constructed with the NFA corresponding to DFA (D, C, T).

In the following, we are going to construct a DFA/EC in terms of functions T_{11}, T_{12}, T_{21}, and T_{22}. Since we only need to prove the existence of such DFA/EC, we temporarily assume $C^e = D_2 \times C$, and use a trivial function $H^e(d_2, c) = \{d_2, c\}$. Also, we break T^e into two functions: $T_1^e : D_1 \times C^e \to D_1$ and $T_2^e : D_1 \times C^e \to D_2$. Then, we can define the functions in DFA/EC as follows:

$$T_1^e(d_1, c^e) = T_1^e(d_1, \{d_2, c\}) = T_{11}(d_1, c) \cup T_{21}(d_2, c),$$
$$T_2^e(d_1, c^e) = T_2^e(d_1, \{d_2, c\}) = T_{12}(d_1, c),$$
$$A(d_2, c) = T_{22}(d_2, c).$$

Recall that, for each byte c, a DFA/EC updates its state $\{d_1, d_2\}$ with the following functions:

$$c^e = H^e(d_2, c),\ \{d_1, d_2'\} = T^e(d_1, c^e),\ d_2 = d_2' \cup A(d_2, c).$$

Therefore, for a new DFA/EC state $\{d_1, d_2\}$,

$$d_1 \cup d_2 = T_1^e(d_1, c^e) \cup (T_2^e(d_1, c^e) \cup A(d_2, c))$$
$$= (T_{11}(d_1, c) \cup T_{21}(d_2, c)) \cup (T_{12}(d_1, c) \cup T_{22}(d_2, c)) = T(d, c).$$

As a result, for any DFA, there is an equivalent DFA/EC. ■

In the above proof, we used trivial definitions for function H^e and its range C^e, but the size of the extended character-set $|C^e| = |C| \times |D_2|$ can be very large. To reduce $|C^e|$ and preserve functional equivalence, we can use other definitions for H^e and C^e, as long as the following equations are true:

$$T_1^e(d_1, H^e(d_2, c)) = T_{11}(d_1, c) \cup T_{21}(d_2, c),$$
$$T_2^e(d_1, H^e(d_2, c)) = T_{12}(d_1, c).$$

Theorem 2 summarizes the conditions when a DFA/EC is equivalent to a DFA. It will be used in Section V to prove the correctness of our efficient DFA/EC implementation.

Theorem 2 (The DFA/EC–DFA equivalence conditions): For a DFA defined by (D, C, T) with its equivalent form $(D_1, D_2, C, T_{11}, T_{12}, T_{21}, T_{22})$ (see Equations 2) and a DFA/EC defined by $(D_1, D_2, C, C^e, H^e, T^e, A)$, the equivalence conditions are:

$$T^e(d_1, H^e(d_2, c)) = \{T_{11}(d_1, c) \cup T_{21}(d_2, c)\} \times T_{12}(d_1, c), \quad (3)$$
$$A(d_2, c) = T_{22}(d_2, c). \quad (4)$$

Proof: It follows from the proof of Theorem 1. ■

V. An Efficient Implementation of DFA/EC

A. Overview

We have presented the formal model of DFA/EC, which removes part of a DFA state and incorporates this part with the set of input characters into the extended character set. However, this model cannot reduce the size of the transition table since the increase in the size of the extended character set $|C| \times |D_2|$ can be more significant than the decrease in the number of states $|D_1|$, i.e., $|C| \times |D_2| \times |D_1| > |C| \times |D|$.

This section presents an efficient implementation of DFA/EC, which contains an encoding method. The encoding method encodes the complementary state into a single bit so that the the size of the extended character set merely doubles as the number of states drops by orders of magnitude. Specifically, we define $H^e : D_2 \times C \to C \times \mathbb{B}$, which uses

a single bit to encode the current state of the complementary program given the next byte in the payload.

Our efficient implementation of DFA/EC consists of (1) a compact main DFA of size $|D_1| \times 2 \times |C|$, which requires only one memory access in its transition table for each byte in the payload, and (2) a complementary program which is efficient and runs without any memory access. Here, the complementary program is very succinct so that, together with the main DFA lookup program, it can be stored entirely in the cache memory or in the on-chip memory.

The key challenge in our implementation is the selection of the set of complementary states N_2 such that (1) the number of states $|D_1|$ of the main DFA is small, (2) we can encode the complementary state into a single bit, and (3) the equivalence condition in Theorem 2 holds.

B. Scoring the states

As discussed in Seciton III, the frequently active NFA states are more likely to cause state explosion. Therefore, to get a compact main DFA, we try to systematically identify those NFA states and add them to the set of complementary states N_2. Our method is to score the states and add the states with the highest scores into N_2. It is noticeable that the NFA states associated with repetitions of a large character-set, e.g. states 2, 6, and 7 in Figure 1, are likely to be frequently active [1], [9], [20]. The reason is that, once a state is active, the chance that the next byte in the payload falls into the character-set of the repetition, increases as the size of the character-set increases. Also, we observe that, if there is a transition from states, say n_1 to n_2, on a large character-set, and n_1 is frequently active, then n_2 is also frequently active. Based on the above two reasons, we propose the following scoring rules:

1) States with exactly one incoming transition, which is also a self-transition, e.g. states 0 and 4 in Figure 1, are assigned score 0.
2) Other states with a self-transition on a character-set, say C', are assigned score $|C'|$.
3) Based on the score of the previous steps, if there is a transition from states n_1 to n_2 ($n_1 \neq n_2$) on a character-set, say C'', and state n_1 has a score $|C'|$, then the score of n_2 is increased by $\min\{|C'|, |C''|\}$.

As an example, the scores of the states in the NFAs in Figure 1 are shown in Table II. In the same table, we also show the number of cases that each NFA state duplicates the sets of other active states (see Section III). The NFA states that have a high score also have a high number of such cases. Also, when using 10 as a threshold, we can identify all frequently active NFA states, i.e. states 2, 6, and 7. In our experiment in Section VI, we use a heuristic to select 32 NFA states into N_2 that have the highest scores and satisfy the two constraints to be introduced below.

C. Determine the extended character-set

In order to encode the complementary state d_2, we need to put two constraints on the selection of the complementary states. As a result, not all NFA states with the highest scores are complementary states. The purpose is to reduce the range of function H^e, which is also the size of the extended character-set C^e. Otherwise, a large extended character-set would undermine the advantage of reducing the number of states D_1 in the main DFA.

We define a function $C_o : N_2 \rightarrow C$, which returns the set of total characters on all the transitions from a complementary state $n_2 \in N_2$ to some main states in N_1. In Figure 1, for the complementary states 2, 6, and 7, $C_o(2) = K$ and $C_o(6) = C_o(7) = \phi$. We define a *non-conflicting complementary set* as a complementary set N_2 such that $C_o(n_i) \cap C_o(n_j) = \phi$, for any $n_i, n_j \in N_2$. If N_2 is non-conflicting, then we can define a reverse function $C'_o : C \rightarrow N_2$ as:

$$C'_o(c) = \begin{cases} n, & c \in C_o(n), n \in N_2 \\ \phi, & c \notin \cup_{n \in N_2} C_o(n) \end{cases}$$

Here, C_o is reversible since any character-set c can be in at most one $C_o(n)$ for some $n \in N_2$, when N_2 is non-conflicting.

With N_2 being non-conflicting, we define the extended character-set as $C^e = C \times \mathbb{B}$ and function H^e as $H^e : D_2 \times C \rightarrow C \times \mathbb{B}$, where $\mathbb{B} = \{T, F\}$ is the boolean set. Specifically, $H^e(d_2, c) = \{c, T\{C'_o(c) \neq \phi\}\}$, where $d_2 \in D_2$ ($d_2 \subseteq N_2$) is the current state of the complementary program, and $T\{C'_o(c) \neq \phi\}$ is a true function which returns either true or false (T/F), depending on whether the enclosed condition is satisfied. Here, $C'_o(c) \neq \phi$ means, on character c, there is a transition from a state $n_2 \in N_2$ to some states in N_1.

With the non-conflicting constraint, for each main state $d_1 \in D_1$, each character $c \in C$, and each value of H^e (T/F), there is exactly one transition in the main DFA. Therefore, the state of the complementary program can be encoded by a single bit for a given byte, and the size of the extended character-set can be reduced to $|C^e| = 2|C|$. The following theorem states that, with our definition of H^e, the first equivalence condition (Equation 3) in Theorem 2 is satisfied.

Theorem 3 (The correctness of H^e): If N_2 is non-conflicting, there exists a function T^e such that $T^e(d_1, H^e(d_2, c)) = \{T_{11}(d_1, c) \cup T_{21}(d_2, c)\} \times T_{12}(d_1, c)$.

Proof: Let $T^e = T^e_1 \times T^e_2$. First, we define $T^e_2(d_1, H^e(d_2, c)) = T_{12}(d_1, c)$. Then, we only need to prove that $T^e_1(d_1, H^e(d_2, c)) = T_{11}(d_1, c) \cup T_{21}(d_2, c)$. We define T^e_1 as $T^e_1(d_1, H^e(d_2, c)) =$

$$\begin{cases} T_{11}(d_1, c), & \text{if } C'_o(c) = \phi \\ T_{11}(d_1, c) \cup T_{21}(\{C'_o(c)\}, c), & \text{if } C'_o(c) \neq \phi \end{cases} \quad (5)$$

Since N_2 is non-conflicting, for a given c, there is at most one $n \in d_2$ ($n = C'_o(c)$) transition to a set of one or more main states. In the case that n does not exist, $C'_o(c) = \phi$ and $T_{21}(d_2, c) = \phi$. In the case that $C'_o(c) \neq \phi$, $T_{21}(\{C'_o(c)\}, c) = T_{21}(d_2, c)$ because $n = C'_o(c)$ is the only active state in d_2 that has transitions to some states in N_1 on character c. ■

To summarize, with the non-conflicting constraint on N_2 and our definition of H^e, the extended character-set $C^e = C \times \mathbb{B}$, whose size is twice that of the original character-set C. At first glance, the non-conflicting constraint may exclude many states from N_2. Fortunately, this constraint excludes few

Algorithm 1 The DFA/EC simulator

1: $e \leftarrow d_2 \& A_{21}[c]$
2: $\{d_1, d_2'\} \leftarrow Tx[d_1][c \cdot e]$
3: $d_2 \leftarrow (d_2 \& A_2[c]) | ((d_2 \& A_{22}[c]) \gg 1) | d_2'$

states from N_2 in practical rule-sets with a large number of regular expressions. This is because, most states transit to only one other state, and for a state n_1 whose score is high and who has a transition to another state n_2 on a large character-set, the score of n_2 is likely to be high too, and in this case, $C_o(n_1) = \phi$. Therefore, it is likely that, for a $n \in N_2$, $C_o(n)$ is either of small size or equal to ϕ, and non-conflicting is not a stringent constraint.

D. The efficient complementary program

Recall that, in our DFA/EC defined in Equation 1, function T^e is implemented by a transition table and a lookup function, and functions H^e and A are implemented by the complementary program. For the efficient implementation of H^e and A, we have one more constraint on the selection of the complementary states N_2: we call a set of complementary states *binary*, if each $n \in N_2$ can transit to at most one other state in N_2. Note that the binary constraint is in terms of the transitions within N_2, while the non-conflicting constraint, defined previously, concerns transitions from states in N_2 to states in N_1.

We show the implementation of function A first, followed by H^e. From Theorem 2, it is required that $A = T_{22}$ for the equivalence of DFA/EC and DFA. First, if the binary constraint is satisfied, the states in N_2 can be arranged such that, if there is a transition from n_i to n_j, then $j = i + 1$. Second, we represent the states in N_2 with an array of bits, and we use the i^{th} bit to represent state n_i. Third, we can represent the transitions within N_2 with two sets of bit masks, A_2 and A_{22}. For each character $c \in C$, $A_2[c]$ and $A_{22}[c]$ are the bit masks for c. The i^{th} bit in $A_2[c]$ being one means that state n_i has a transition to itself on character c, and the i^{th} bit in $A_{22}[c]$ being one means that state n_i has a transition to state n_{i+1} on character c. Let $d_2 \subset N_2$ be represented by a bit array with the i^{th} bit being one or zero representing whether state n_i is active, then the next state of the complementary program can be calculated by $d_2 = A(d_2, c) = (d_2 \& A_2[c]) | ((d_2 \& A_{22}[c]) \gg 1)$, where &, |, $\gg$ are the bitwise AND, OR, SHIFT operations, respectively. Clearly, $A = T_{22}$ implements the transitions within N_2.

Similarly, we define another set of bit masks A_{21} for different $c \in C$, and the i^{th} bit in $A_{21}[c]$ being one means that the state n_i has a transition to some main states in N_1 on character c. Then, $H^e(d_2, c) = \{c, T\{d_2 \& A_{21}[c] \neq 0\}\}$. The masks A_2, A_{22}, and A_{21} of the DFA/EC in Figure 2 are shown in binary digits in Table III(b-d), respectively.

The main DFA is implemented by a lookup program and a transition table Tx with its two dimensions being the state indexes of the main DFA and the extended character-set. The pseudo code for the execution of a DFA/EC is listed in Algorithm 1, where e is a boolean value representing the result of function H^e. The concatenation $c \cdot e$ is the extended character created from c and e.

Algorithm 2 The construction of the main DFA table

1: $D \leftarrow$ the set of conventional DFA states
2: **for each** (d **in** D)
3: $\quad d_1 \leftarrow d \cap N_1$
4: $\quad$ **for each** (c **in** C)
5: $\quad\quad d_1' = T(d_1, c) \cap N_1$
6: $\quad\quad d_1'' = T(d_1 \cup N_2, c) \cap N_1$
7: $\quad\quad d_2' = T(d_1, c) \cap N_2$
8: $\quad\quad Tx[d_1][c \cdot F] = \{d_1', d_2'\}$
9: $\quad\quad Tx[d_1][c \cdot T] = \{d_1'', d_2'\}$

Now, let us discuss the memory requirement and the memory bandwidth of DFA/EC. The size of the main DFA table depends on the number of states in the main DFA, the size of the extended character-set $2|C|$, and the encoded size of each transition entry, i.e. $\{d_1, d_2'\}$. Let the number of states in the main DFA be $|D_1|$, the bits required to encode the index for d_1 is $\lceil \log_2 |D_1| \rceil$. Note that the value of $d_2' = T_2^e(d_1, H^e(d_2, c)) = T_{12}(d_1, c)$ is irrelevant to the value of H^e and it can ideally be stored once for each c. In practice, we do not have to represent d_2' explicitly as a bit-array of length $|N_2|$ since the set of all possible values of d_2', which can be represented by a set of bit-arrays, denoted by A_{12}, are very limited in number, and we can use the index of d_2' in A_{12} to represent d_2'. Therefore, the total size of the transition table is $|D_1| \times |C| \times (2 \times \lceil \log_2 |D_1| \rceil + \lceil \log_2 |A_{12}| \rceil)$ bits, and the memory bandwidth is $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |A_{12}| \rceil$ bits. A DFA/EC needs to maintain its current state, i.e. $\{d_1, d_2\}$, which takes $\lceil \log_2 |D_1| \rceil + |N_2|$ bits.

E. The construction of DFA/EC

The data needed to be constructed for a DFA/EC are: the main DFA table Tx, the sets of bit-masks A_2, A_{22}, and A_{21}. The construction of the main DFA table, which implements function T_1^e, as defined in Equation 5, is shown in Algorithm 2, where we regard each DFA state as a set of NFA states and assume T to be a function that returns the new set of active NFA states, given the old set of active NFA states and the next byte. We use all states in a constructed conventional DFA to determine the possible main DFA states, because not all the combinations of main states in N_1 can be simultaneously active. We will study more efficient DFA/EC construction without using a constructed DFA in our future work.

The transition table Tx of the main DFA and the masks A_2, A_{22}, and A_{21} of our example DFA/EC in Figure 2 are shown in Table III. In Table III(a), the first column shows the indexes of the states in the main DFA, the second column shows the sets of simultaneously active main states represented by the main DFA states, and all the remaining columns are transitions. Each cell in the transition table consists of three values, which are the results of the functions $T_1^e(d_1, \{c, F\})$, $T_1^e(d_1, \{c, T\})$,

TABLE III
TABLES IN THE DFA/EC IN FIGURE 2.

(a) The main DFA table T^e

state#	$d_1 \subseteq N_1$	[^AC-N]	A	[CD]	[E-GIJL]	H	K	[MN]
0 accept	(0,3,4)	2, 2, 000_B	3, 3, 000_B	2, 2, 000_B	2, 2, 000_B	1, 1, 000_B	2, 2, 000_B	2, 2, 000_B
1	(0,4,5)	2, 2, 010_B	3, 3, 010_B	2, 2, 010_B	2, 2, 000_B	1, 1, 000_B	2, 2, 000_B	2, 2, 000_B
2 start	(0,4)	2, 2, 000_B	3, 3, 000_B	2, 2, 000_B	2, 2, 000_B	1, 1, 000_B	2, 0, 000_B	2, 2, 000_B
3	(0,1,4)	2, 2, 100_B	3, 3, 100_B	2, 2, 000_B	2, 2, 000_B	1, 1, 000_B	2, 0, 000_B	2, 2, 100_B

(b) The masks A_{21}

	K
A_{21}	100_B

(c) The masks A_{22}

	[^I-R]
A_{22}	010_B

(d) The masks A_2

	[MN]	[O-R]	[^C-R]	[CD]	[E-H]
A_2	100_B	110_B	111_B	011_B	001_B

and $T_2^e(d_1, \{c, T/F\})$, respectively. As we can see in Table III(a), values of $T_1^e(d_1, \{c, F\})$ and $T_1^e(d_1, \{c, T\})$ are represented by indexes, and they are equal in most cases. Values of $T_2^e(d_1, \{c, T/F\})$ are shown in binary digits, and there are only three different values (i.e. 000_B, 010_B, and 100_B). This shows that there is room for further compression in DFA/EC with transition compression techniques [12].

VI. EVALUATION

In our experiment, we endeavored the following efforts: First, we developed several compilers, which read files of rules and create the corresponding inspection programs and the transition tables for DFA, MDFA [1], H-FA [9], and DFA/EC. Second, we extracted rule-sets from the Snort [6], [7] rules. Third, we developed a synthetic payload generator. We generate the inspection programs for the rule-sets, measure their storages, and feed them with the synthetic payloads to measure their performances.

We compare with DFA and MDFA [1]. MDFA divides the rule-set into M groups and compiles each group into a distinct DFA. Although our algorithm can be combined with MDFA, i.e. we can replace the individual DFAs in a MDFA with DFA/ECs, we compare our algorithm with this widely adopted algorithm to show the efficiency of our method in terms of storage, memory bandwidth, and speed. We compare with 2DFA, 4DFA, and 8DFA, which are MDFAs with 2, 4, and 8 paralleled DFAs, respectively.

Since our algorithm is for state compression, we do not compare our algorithm with other types of algorithms that are orthogonal and complementary to our algorithm, such as transition compression [2] and alphabet compression [12]. We will examine how well DFA/EC can be combined with them in the future. We do not show the results of H-FA [9] because, with our rule-sets, it has very large numbers of conditional transitions per character, which results in significant memory requirements and memory bandwidth. We did not implement XFA [20] because the XFA compiler, which employs complicated compiler optimization technologies, is not available.

Our compilers are based on the Java regular expression package "java.util.regex.*". Our compilers generate NFA data structures instead of parser trees, as in the original implementation. All Perl-compatible features, except back-reference and counters, are supported. Our compilers output C++ and Java files for NFAs, DFAs, H-FA, and DFA/ECs. The construction of the DFA/ECs is as efficient as the construction of DFAs.

We extracted rule-sets from Snort [6], [7] rules released Dec. 2009. Rules in Snort have been classified into different categories. We adopt a subset of the rule-set in each category, such that each rule-set can be implemented by a single DFA using less than 2GB of memory. Almost all patterns in our rule-sets contain repetitions on large character-sets.

Each payload file consists of payload streams of 1KB, and the total size of each payload file is 64MB. To generate a payload stream for a rule-set, we travel the DFA of the whole rule-set. We count the visiting times of each state, and give priority to the less visited states and non-acceptance states. This traffic generator can simulate malicious traffics [22], which prevent the DFA from traveling only low-depth states, as it does in normal traffics. We do not use normal traffic since it would result in similar performance across all inspection programs, as only a small number of shallow states are travelled in normal traffic.

A. Results on memory requirements

We measure the memory requirement of each inspection program in terms of (1) the number of states, (2) the number of transitions, and (3) the bits needed to store the transitions. Ideally, the number of states determines the number of bits required to encode a state index. As shown in Table IV(a), the number of states in a DFA/EC can be four orders of magnitude smaller than that of a DFA, two orders of magnitude smaller than a 2DFA, 5 times smaller than a 4DFA, and comparable to that of an 8DFA. The significant reduction is because of the removal of the frequently active complementary states in DFA/EC, which otherwise causes the exponential expansion in the number of DFA states.

The number of transitions is the sum of the numbers of transitions of each state. The number of transitions of each state is measured by the number of distinguished states it can transit to. In other words, we measure the minimum possible number of transitions in any state and transition encoding technique, which is not our focus. As shown in Table IV(c), the number of transitions of DFA/EC can be four orders of magnitude smaller than that of a DFA, two orders of magnitude smaller than a 2DFA, 2 times smaller than a 4DFA, and comparable to that of an 8DFA.

TABLE IV
THE MEMORY REQUIREMENTS WITH DIFFERENT RULE-SETS.

(a) The total number of states (percentage to DFA).

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	343k	0.06%	2%	0.3%	0.1%
smtp-36	141k	0.4%	6%	0.6%	0.3%
specific-threats-21	53k	2%	7%	2%	1%
spyware-put-93	269k	1%	18%	5%	1%
web-client-35	106k	3%	14%	1%	0.7%
web-misc-28	453k	0.08%	3%	0.3%	0.1%

(b) The size of the per-flow state (bits / words).

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	19 / 1	23 / 2	25 / 2	32 / 4	48 / 8
smtp-36	18 / 1	38 / 2	22 / 2	31 / 4	46 / 8
specific-threats-21	16 / 1	31 / 2	21 / 2	31 / 4	47 / 8
spyware-put-93	19 / 1	39 / 2	27 / 2	43 / 4	70 / 8
web-client-35	17 / 1	33 / 2	24 / 2	37 / 4	56 / 8
web-misc-28	19 / 1	30 / 2	26 / 2	36 / 4	56 / 8

(c) The total number of transitions (percentage to DFA).

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	7m	0.04%	1%	0.1%	0.02%
smtp-36	2m	0.4%	6%	0.2%	0.07%
specific-threats-21	702k	3%	6%	1%	0.3%
spyware-put-93	7m	1%	7%	2%	0.3%
web-client-35	2m	4%	7%	0.6%	0.2%
web-misc-28	8m	0.08%	2%	0.1%	0.04%

(d) The transition storage (bits/percentage to DFA).

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	124m	0.03%	1%	0.06%	0.008%
smtp-36	37m	0.3%	4%	0.1%	0.02%
specific-threats-21	11m	2%	4%	0.6%	0.1%
spyware-put-93	141m	1%	6%	1%	0.1%
web-client-35	40m	3%	6%	0.3%	0.1%
web-misc-28	155m	0.05%	1%	0.08%	0.01%

We measure the total minimum memory (storeage) requirement of the transition tables in terms of bits, and the number of bits is the product of the number of transitions and the number of bits needed to encode each transition. For DFA, MDFA, and DFA/EC, the number of bits needed to encode each transition are $\lceil \log_2 |D| \rceil$, $\sum_{i=1}^{M} \lceil \log_2 |D_i| \rceil$, and $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |A_{12}| \rceil$, respectively. Here, D is the set of DFA states, D_i is the set of DFA states in the i^{th} DFA of a MDFA, D_1 is the set main DFA states of a DFA/EC, and A_{12} is the set of masks of a DFA/EC required to implement the transition function T_{12}. As shown in Table IV(d), the transition storage of a DFA/EC can be four orders of magnitude smaller than that of a DFA, two orders of magnitude smaller than a 2DFA, and 2 times smaller than a 4DFA.

Finally, we measure the sizes of the per-flow state of the inspection programs in terms of bits and words. In terms of bits, the per-flow state for DFA, MDFA, and DFA/EC are $\lceil \log_2 |D| \rceil$, $\sum_{i=1}^{M} \lceil \log_2 |D_i| \rceil$, and $\lceil \log_2 |D_1| \rceil + N_2$, respectively. Here, N_2 is the number of complementary states in a DFA/EC. As shown in Figure IV(b), DFA/EC has small sizes of per-flow state in terms of both bits and words.

B. Results on memory bandwidth and speed

Memory bandwidth is the amount of memory access per byte in the payload, which we measure it in terms of bits. The memory bandwidth of DFA, MDFA, and DFA/EC are $\lceil \log_2 |D| \rceil$, $\sum_{i=1}^{M} \lceil \log_2 |D_i| \rceil$, and $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |A_{12}| \rceil$, respectively. In Figure 3(a), it is exciting to see that the memory bandwidth of DFA/EC can be smaller than DFA and is much smaller than MDFAs. To our best knowledge, we are the first to report a inspection program for rule-sets of regular expressions whose bandwidth is smaller than that of DFA.

In Figure 3(b), we show the number of memory accesses per KB of payload. DFA/EC and DFA have the minimum number of memory accesses, while those of MDFAs increase proportional to M.

We measure the speed of the inspection programs using Java and C++ implementations in a Unix machine with 4GB of memory and a 3GHz Intel Core 2 Duo CPU. Note that the speeds of the inspection programs depend on the hardware and software on which they are implemented. For example, with general-purpose processors and ASIC hardware, they vary in different amounts of cache or on-chip memory.

Results are shown in Figures 3(c-d). In several cases, DFA/EC is the fastest in both implementations, and DFA/EC can be over 10 times faster than DFA and two times faster than MDFA in Java. MDFA is fast because of its compact transition table size and the large amount of cache memory in our platform. We believe that DFA/EC will be more favorable for the implementations in ASIC hardware or GPUs that have less cache memory and more computation resources.

The experiment results suggest that our Java programs are faster than our C++ programs in terms of instruction execution, but are slower in terms of memory access. We believe this is due to the difference between the Java and the g++ compilers: the Java just-in-time (JIT) compilation is better optimized to the targeted CPU and OS. On the other hand, memory access in Java is slower because that Java programs perform additional data security checks, such as null pointer checks and array index out of bound checks.

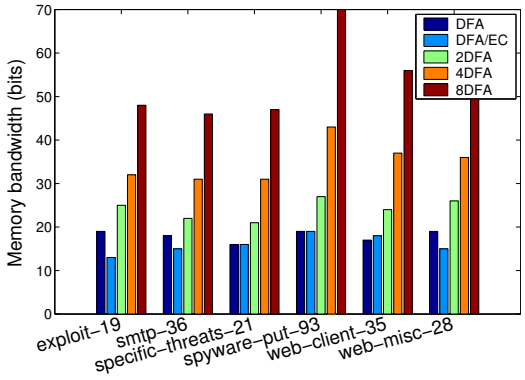
C. Summary

Our experiment results show that DFA/EC can be over four orders of magnitude smaller than DFA in terms of number of states and transitions. DFA/EC also has a smaller memory bandwidth and runs faster than DFA.

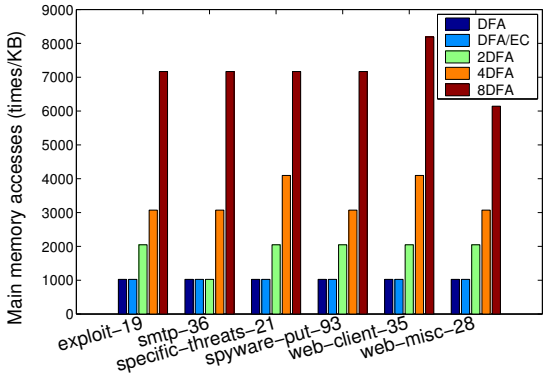
VII. CONCLUSION

In this paper, we investigated a general-purpose processor and regular expressions based deep packet inspection algorithm, called deterministic finite automata with extended character-set (DFA/EC). Different from existing state reduction algorithms, our solution requires only a single memory access for each byte in the traffic payload, which is the minimum. We performed experiments with Snort rule-sets and synthetic payloads. Experiment results show that a DFA/EC

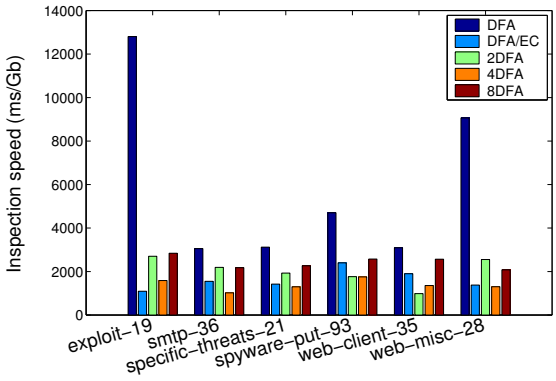
Fig. 3. Memory bandwidth, memory accesses, and inspection speed with different rule-sets (milliseconds per 64MB).



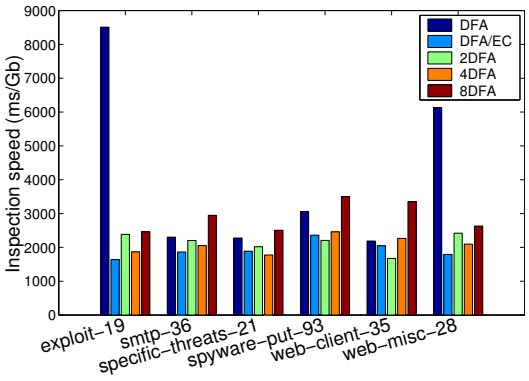
(a) Memory bandwidth (bits)



(b) Memory accesses (times/KB)



(c) Inspection speed (Java)



(d) Inspection speed (C++)

can be over four orders of magnitude smaller than a DFA, has a smaller memory bandwidth, and runs faster than a DFA. In the future, we will study efficient DFA/EC construction algorithms without using DFA, combine DFA/EC with the existing transition compression and character-set compression techniques, and perform experiments with more rule-sets.

REFERENCES

[1] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection. In *Proc. of ANCS*, 2006.

[2] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *Proc. of ACM SIGCOMM*, September 2006.

[3] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. In *Proc. of IEEE INFOCOM*, 2010.

[4] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In *Proc. of IEEE INFOCOM*, 2004.

[5] K. Namjoshi and G. Narlikar. Robust and Fast Pattern Matching for Intrusion Detection. In *Proc. of IEEE INFOCOM*, 2010.

[6] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of 13th System Administration Conf.*, November 1999.

[7] Snort: http://www.Snort.org/.

[8] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, December 1999.

[9] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms From Insomnia. In *Proc. of ANCS*, 2007.

[10] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proc. of CoNEXT*, 2007.

[11] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. In *Proc. of IEEE INFOCOM*, 2009.

[12] S. Kong, R. Smith, and C. Estan. Efficient Signature Matching With Multiple Alphabet Compression Tables. In *Proc. of Securecomm*, 2008.

[13] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. of FCCM*, 2001.

[14] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection With Reconfigurable Hardware. In *Proc. of FCCM*, 2002.

[15] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuit for Matching Complex Network Intrusion Detection Patterns. In *Proc. of FLP*, 2003.

[16] B. Brodie, R. Cytron, and D. Taylor. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proc. of ISCA*, 2006.

[17] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proc. of ANCS*, 2007.

[18] S. Kumar, J. Turner, and J. Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proc. of ANCS*, 2006.

[19] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proc. of ISCA*, 2005.

[20] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata. In *Proc. of ACM SIGCOMM*, 2008.

[21] M. Becchi and P. Crowley. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proc. of CoNEXT*, 2008.

[22] M. Becchi, M. Franklin, and P. Crowley. A Workload for Evaluating Deep Packet Inspection Architectures. In *Proc. of IISWC*, 2008.