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

Cong Liu, Yan Pan, Ai Chen, and Jie Wu, *Fellow*, *IEEE*

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 doubling the size of the character-set. Unlike existing state reduction algorithms, our solution requires only a single main memory access for each byte in the traffic payload, which is the minimum. We perform experiments with several Snort rule-sets. Results show that, compared to DFAs, DFA/ECs are very compact and are over four orders of magnitude smaller in the best cases; DFA/ECs also have smaller memory bandwidth and run faster. We believe that DFA/EC will lay a groundwork for a new type of state compression technique in fast packet inspection.

Index Terms—Deep packet inspection (DPI), regular expression, deterministic finite automata (DFA), extended character-set (EC)

1 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]–[5] on high-speed DPI algorithms, 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 the general-purpose processor approaches is limited by the memory bandwidth of the processors. Therefore, to improve inspection speed, it is critical to minimize the number of main memory (off-chip memory) accesses per byte in the traffic payload. Some implementations of the regular expressions, such as the nondeterministic finite automata (NFAs), have a nondeterministic number of main memory accesses per byte. Another critical issue is reducing the size of the automata stored in memory in order to reduce the cost of memory, improving the scalability for a larger number of rules, and increasing the inspection speed (with the use of cache memory). While *deterministic finite automata* (DFAs) implementations of regular expressions take only one main memory accesses per byte, they often require very large memory space to store their transition tables, which undermines their scalability in real applications. Therefore, conventional DFA and NFA are not ideal in real systems.

Recent research efforts have been focused on reducing the memory storage 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 main memory accesses. The amount of compression in transition reduction and character-set reduction is bounded by the size of the character-set (e.g., the maximum reduction is $\frac{1}{256}$ if ASCII is used) due to the fact that there is at least one transition in each state. We focus on state reduction, which is not limited by the maximum reduction ratio of 256, and we manage to reduce the storage size of DFAs by up to 4 orders of magnitude in our experiment. Moreover, our approach can be incorporated into the other approaches to achieve further memory reduction.

This paper proposes 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. This general model removes part of each DFA state and incorporates it with the next input character. This results in an extended the set of input characters. However, simply

- *C. Liu and Y. Pan are with Sun Yat-sen University, Guangzhou 510006, China. E-mail: panyan5@mail.sysu.edu.cn. (corresponding author: Y. Pan.)*
- *A. Chen Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences.*
- *J. Wu is with the Temple University, Philadelphia, PA 19122.*

Manuscript received 14 Mar. 2012; revised 27 Oct. 2012; accepted 16 Apr. 2013. Date of publication 18 Apr. 2013; date of current version 15 July 2014. Recommended for acceptance by H. Shen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.93

TABLE 1
Notations Used in This Paper

Notation	Meaning	Note
N	The set of NFA states N	
D	The set of 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	The original character-set	
C^e	The extended character-set	$C^e = C \times \mathbb{B}$
$H^e\&M$	Functions used in the complementary program	

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 the removed DFA state into a single bit. As a result, the size of the extended character set merely doubles, while 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 a part of the DFA state into the set of input characters.
2. We provide an efficient implementation of the 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 an extensive evaluation to compare DFA/EC with related algorithms by using several Snort rulesets.

Compared with existing state reduction algorithms, DFA/EC significantly increases the inspection speed by keeping the number of per-byte main memory accesses 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 several Snort rule-sets demonstrate that DFA/ECs are very compact and achieve high inspection speed. Specifically, DFA/ECs are over four orders of magnitude smaller than DFAs in the best cases; DFA/ECs can even have smaller memory bandwidth than DFAs, which is not seen in previous compression algorithms. The advantages of a DFA/EC are summarized in the following:

1. A DFA/EC requires only one main memory access for each byte in the packet payload, while significantly reducing storage in terms of table size.
2. A DFA/EC is conceptually simple, easy to implement, and easy to update due to fast construction speed.
3. A DFA/EC can be combined with other compression 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 2. Section 3 introduces the concept of DFA/EC with an example. Section 4 presents the formal model of DFA/EC and its DFA-equivalence condition. Section 5 describes an efficient implementation of DFA/EC and proves its DFA equivalence. Section 6 evaluates DFA/EC by using the Snort rule-sets and synthetic traffic. Section 7 concludes the paper. The notations used in this paper are summarized in Table 1.

2 Related Work

Prior work on regular expression matching at line rate can be categorized by their implementation platforms into FPGA-based implementations [13]–[17] and general-purpose processors and ASIC hardware implementations [1], [2], [9], [10], [18], [19]. FPGA implementations exploit high degree of parallelism, and the achieved high throughput is difficult for the memory-based approaches. However, FPGAs are not available in many applications including those already deployed. On the other hand, the general-purpose processor approaches are often desirable because they provide a higher degree of flexibility and they allow for frequent update of rule-sets.

Existing transition table compression techniques based on general-purpose processors include: (1) DFA state compression techniques and (2) transition compression techniques. DFA state compression techniques reduce the number of DFA states like MDFA [1], HFA [9], XFA [20]. Transition compression techniques reduce the number of transitions in each state such as $\mathrm{D^2FA}$ [2], $\mathrm{CD^2FA}$ [18]. Both kinds of techniques effectively reduce the memory storage but introduce additional main memory accesses per byte. Note that the two kinds of compression techniques are perpendicular and can be combined. Our work in this paper builds upon the area of DFA state compression.

Delayed Input Deterministic Finite Automata ($\mathrm{D^2FA}$) [2] uses default transitions to reduce the memory storage requirement. 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, the main memory must be accessed once more to retrieve the transitions of the referred state [2], [18].

Using auxiliary variables and devising a compact and efficient inspection program is challenging and is related to our work. Two seminal papers [9], [20] use auxiliary variables to represent the "factored out" auxiliary states in order to reduce the DFA size. However, the auxiliary variables are manipulated by auxiliary programs associated with each state or transition, resulting in extra main 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 [20] 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, DFA/EC uses a compact program to generate its extended characters, and it requires a single main memory access for each byte in the payload.

Hybrid-FA [9], [10] prevents state explosion by performing partial NFA-to-DFA conversions. The outcome is a hybrid automaton consisting of a head-DFA and several tail-automata. The tail-automata can be NFAs or DFAs. However, maintaining multiple DFA/NFA may introduce a large per-flow state and scarify the 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, we propose an encoding method to limit the extended character set to twice the size of the original character-set, which is the key to making 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 are inefficient to be directly implemeted by 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 by using the techniques in [9] and [20]. We omit these advanced features in this work for simplicity.

3 The Conceptual DFA/EC

In this section, we will first review the preliminaries on automata that is used in packet inspection, i.e., the *nondeterministic finite automata* (NFA) and *deterministic finite automata* (DFA). We then discuss an example of DFA/EC that describes our motivation for this paper.

3.1 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: "$.*A[\hat{}C\text{-}L]+K$" and "$.*H[\hat{}E\text{-}N]+[\hat{}I\text{-}R]+$". 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]", match any character between "E" and "N", and "$[\hat{}E\text{-}N]$" is the complement of "[E-N]" that matches any character not in this range. A wildcard "." is equal to "$[\hat{}]$" and matches any character. Repetition "$.*$" matches any strings with a length from zero to infinity, and repetition "$[\hat{}E\text{-}N]+$" matches nonempty strings containing characters in "$[\hat{}E\text{-}N]$". For instance, the pattern "$H[\hat{}E\text{-}N]+[\hat{}I\text{-}R]+$" matches the strings "HAT" and "HADST".

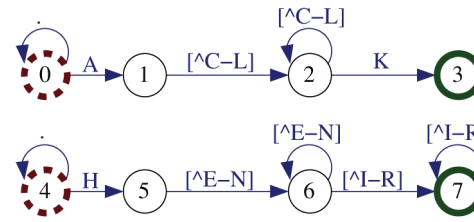


Fig. 1. The NFAs for "$.*A[\hat{}C\text{-}L]+K$" and "$.*H[\hat{}E\text{-}N]+[\hat{}I\text{-}R]+$", respectively.

NFA and DFA are popular pattern matching programs for a set of one or more regular expressions. Fig. 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 main memory accesses are required to obtain the next transitions for all active states. The sequence of the sets of active states experienced by the example NFA while matching 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 a set of NFAs by using the subset construction routine, in which a DFA state is created to represent each set of NFA states that can be simultaneously active in some matching process. Therefore, the number of DFA states is the number of possible combinations of active NFA states that can be simultaneously active, which can be exponential to the number of NFA states. 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, and let D be the set of DFA states. We have: (1) for any DFA state $d \in D$, $d \subseteq N$, (2) $D \subset 2^N$ (2^N is 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 minimal DFA (which is not shown in this paper) constructed for the example NFA contains 18 states. On the other hand, $|D| \ll 2^{|N|}$ because not all combinations of NFA states can be simultaneously active.

3.2 Motivation and Overview

Different methods to resolve the state explosion problem have been proposed in [1], [9], and [20]. The NFA states that correspond to the repetitions of large character-sets, such as states 2, 6, and 7 in our example NFA in Fig. 1, 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 a set of concurrently active NFA states, and it is a DFA state; the frequently active NFA

TABLE 2
The Cases of Each NFA State "Duplicating" the DFA States and the Number of Independent NFA States (Section 5.3) for Each NFA States in Fig. 1

NFA states	0	1	2	3	4	5	6	7
Duplicating cases	0	7	7	1	0	2	7	8
Independent numbers	0	3	3	0	0	1	3	4

state 2, which can be concurrently active with NFA states 0, 1, and 4, creates another DFA state (0,1,2,4). In Table 2, for each state n in our example NFA (Fig. 1), we show the number of cases where there exists a DFA state $d \in D$ such that $d \cap \{n\} \in D$ too. In other words, this number shows that the number of DFA state can be reduced if the NFA state n is removed.

To reduce the 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* (denoted by D_1) in a 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* (denoted by N_2); we call the remaining NFA states the *main states* (denoted by N_1). As we will see in Section 5, we have additional constraints, which exclude some of the frequently active NFA states from the set of complementary states in order to enable a single-bit encoding method of the complementary states in the extended character set, and to facilitate an efficient DFA/EC implementation.

While the main DFA implements 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 program is very fast while the main states, N_1, can be implemented by a compact main DFA whose size, $|D_1|$, ideally, is equal to $|N_1|$.

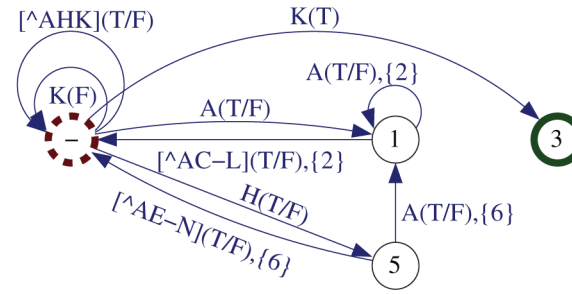
In our evaluation (see Section 6), the main DFA of the DFA/EC is shown to be smaller than its corresponding conventional DFA by an order of four magnitudes, while the extended character-set only doubles the size of the original character-set. When a DFA/EC consumes a byte, the complementary program generates the extended character-set efficiently by only a few instructions without any main memory accesses (Section 5).

3.3 A Detailed Illustration of DFA/EC

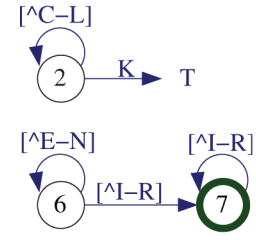
From the NFAs in Fig. 1, we construct a conceptual DFA/EC, which is shown in Fig. 2. Since we have not presented how to select complementary states, we simply assume that the complementary states are the NFA states 2, 6, and 7. The main DFA constructed from the main states (i.e., the NFA states 0, 1, 3, 4, and 5) is shown in Fig. 2(a).

3.3.1 The Structure of a DFA/EC

For clarity, we make the following simplifications in Fig. 2(a): (1) in each DFA state, we remove the NFA states 0 and 4 from the labels of the states, which always exist in any state labels



(a) Main DFA



(b) complementary states

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

since states 0 and 4 are always active: state "1" should actually be labeled with "0,4,1", state "-" 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 in the payload 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 when the extra bit is either true or false, makes the complementary state "6" active.

In Fig. 2(b), the complementary states are concepturally shown as an NFA, which is to be replaced by an efficient implementation in Section 5. The transition from complementary state 2, labelled with "K", does not make any state active, but it sets the extra bit in the extended character-set to true.

3.3.2 How Does a DFA/EC Work?

A DFA/EC maintains two states in runtime: one state d_1 for the main DFA, and an additional state d_2 for the complementary program. In the following discussion, the current runtime state d_1 of the main DFA is represented by a DFA state label, and state d_2 of the complementary program is represented by a set that contains currently active complementary states. In Fig. 2, the initial DFA/EC states is $(-)\{\}$, where $d_1 = (-)$ is the initial state of the main DFA, and $d_2 = \{\}$ is the initial state of the complementary program, which contains no active complementary states.

For each byte in the payload, the DFA/EC functions as follows. (1) The complementary program calculates the extra bit e for the extended character by using the next byte c and the current state d_2 of the complementary program. (2) The next state d_1^* of the main DFA and a label d_2' is looked-up by using the current state d_1 of the main DFA and the extended character $c \cdot e$, which is composed of the next byte c and the extra bit e. (3) The complementary program calculates its next state d_2^* by using the its current state d_2, the next byte c, and the label on the main DFA transition d_2'.

In Fig. 2(b), the example complementary program is conceptually represented by an NFA. We now illustrate how to determine the extra bit e in the extended character and the next state d_2^* of the complementary program. (a) The extra bit e in the extended character is set by the transition in the complementary program. In Fig. 2(b), the transition labelled by "K" set the extra bit e to true. The extra-bit is false if it is not set to true by any transition in the complementary program. (b) The next state d_2^* of the complementary program is the union of the set of complementary states that are activated on the main DFA transition, and those (represented by label d_2') that are activated by the transitions in the complementary program.

Essentially, the execution of DFA/EC is an interactive process between the main DFA and the complementary program: the next state d_2^* of the complementary program is partially determined by the label d_2' on the main DFA transition determined by d_1 and c; while the next state d_1^* of the main DFA is partially determined by e, which reflects whether d_2 affects d_1^*. As we will see in Algorithm 5.4, these interactions in DFA/EC can be implemented efficiently by using a single main memory access and several bit-wise instructions.

3.3.3 A Step-by-Step Example

We will explain how the state of the example DFA/EC changes when matching a payload of string "ABK". The resulting sequence of the DFA/EC states are:

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

Initially, the state of the main DFA is (-), and the state of the complementary program is {}. For the first input character 'A', (1) the extra bit is F since no complementary state is active, (2) the main DFA transition, which is labeled by "A(T/F)" and are from states "-" to "1", is token, and (3) no transition in the complementary program is token (since no complementary state is active). Thus, the second DFA/EC state is "(1){}".

For the second input character 'B', (1) the extra bit is F since no complementary state is active, (2) the main DFA transition, which is labeled by "[^AC-L](T/F), {2}" and is from states "1" to "-", is token, (3) no transition in the complementary program is token since the current complementary state is "{}", and the next state of the complementary program is "{2}", where the complementary state "2" is activated by the main DFA transition token above, whose label ends with a "{2}". Alas, the third DFA/EC state is "(-){2}".

For the third input character 'K', (1) the extra bit is set to T since, in the complementary program (Fig. 2), the transition labeled by "K" is token, and (2) the main DFA transition, which is labeled by "K(T)" and is from states "-" to "3", in the main DFA is token, and (3) the next state of the complementary program is "{}" since no complementary state is activated either on the main DFA transition, or by the transition taken by the complementary program. Thus, the fourth DFA/EC state is "$(\underline{3})\{\}$".

We have illustrated how the main DFA and the complementary program interact with each other. We will define a formal model for DFA/EC and will show the equivalence between DFA/EC and DFA in Section 4. An efficient implementation of DFA/EC will be presented in Section 5.

4 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 states of the main DFA, and $D_2 = \{d_2 | d_2 = d \cap N_2, d \in D\}$ is the set of states of the complete program. A DFA/EC can be defined by $(D_1, D_2, C, C^e, H^e, T^e, M)$, with the following functions:

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

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 $c^e \in C^e$ by using d_2 and c. (2) With d_1 and c^e, function T^e generates a pair of partial states, $\{d_1, d_2'\}$, where d_1 is the next state of the main DFA. (3) With d_2 and the original character c, function M generates another partial state d_2'', and $d_2 = d_2' \cup d_2''$ is the next state of the complementary program.

In our implementation, the transition function of the main DFA, T^e, is implemented by a transition table; H^e and M are implemented by the complementary program, which only contains several efficient instructions.

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

Proof. If we let $T: D \times C \rightarrow D$ 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. Firstly, 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:

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

The equivalence holds as long as, for any $d \in D$, there exist $d_1 \in D_1$, $d_2 \in D_2$ 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 newly active NFA states in D_1 that is activated 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 newly active NFA states in D_2, D_1, D_2 that are activated 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 the DFA (D, C, T).

In the following, we are going to construct a DFA/EC in terms of the functions T_{11}, T_{12}, T_{21}, and T_{22}. Since we only need to prove the existence of such a DFA/EC, we simply assume that $C^e = D_2 \times C$, and we 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:

$$\begin{aligned} T_1^e(d_1, c^e) &= T_1^e(d_1, H^e(d_2, c)) = T_1^e(d_1, \{d_2, c\}) \\ &\equiv T_{11}(d_1, c) \cup T_{21}(d_2, c), \\ T_2^e(d_1, c^e) &= T_2^e(d_1, \{d_2, c\}) \equiv T_{12}(d_1, c), \\ M(d_2, c) &\equiv T_{22}(d_2, c). \end{aligned}$$

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

$$c^e = H^e(d_2, c), d_1 = T_1^e(d_1, c^e), d_2 = T_2^e(d_1, c^e) \cup M(d_2, c).$$

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

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

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:

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

Lemma 1 summarizes the conditions when a DFA/EC is equivalent to a DFA. It will be used in Section 5 to prove the correctness of the efficient DFA/EC implementation.

Lemma 1 (The DFA/EC–DFA Equivalence Conditions). *For a DFA defined by (D, C, T) and its equivalent form $(D_1, D_2, C, T_{11}, T_{12}, T_{21}, T_{22})$ (see Equation 2), and a DFA/EC defined by $(D_1, D_2, C, C^e, H^e, T^e, M)$, the equivalence conditions are:*

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

$$M(d_2, c) = T_{22}(d_2, c). \quad (4)$$

Proof. It follows from the proof of Theorem 1. □

5 An Efficient Implementation

5.1 Overview

We have presented the formal model of DFA/EC, which removes part of a DFA state and incorporates this part along with the set of input characters into the extended character set. However, this model does not ensure a reduction in the size of the transition table. For instance, if we define $C^e = D_2 \times C$ and $H^e : D_2 \times C \to C \times D_2$, 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., it is always true that $|C| \times |D_2| \times |D_1| \geq |C| \times |D|$.

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 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.

The efficient implementation of DFA/EC consists of (1) a compact main DFA of size $|D_1| \times 2|C|$, which requires only one main memory access in its transition table for each byte in the payload, and (2) a complementary program that is efficient and runs without table lookup in the main memory, and, as a result, no main memory access is required. 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 challenges in our implementation lie in 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 1 holds. This section provides the solutions to the above challenges.

5.2 Two Constraints on the Complementary States

In order to encode the complementary state d_2 into the extra bit, we put two constraints on the selection of the complementary states, which are named the *conflicting constraint* and the *binary constraint*. The purpose of these constraints 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 $C_o : N_2 \to C$, which returns the set of total characters on all of the transitions from a complementary state $n_2 \in N_2$ to the main states in N_1. In Fig. 1, for all of the complementary states 2, 6, and 7, $C_o(2) = K$, and $C_o(6) = C_o(7) = \phi$.

Definition 1 (Non-Conflicting Complementary Set). *A complementary set N_2 is non-conflicting if $C_o(n_i) \cap C_o(n_j) = \phi$, for any $n_i, n_j \in N_2$.*

Since all pairs of $C_o(n_i)$ and $C_o(n_j)$ are disjoint for $n_i, n_j \in N_2$ when N_2 is non-conflicting, for any $c \in C$, there is at most one n_k such that $C_o(n_k) = c$. As a result, we can define a reverse function $C'_o : C \rightarrow N_2$ of C_o as:

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

With N_2 being non-conflicting, we define the extended character-set as $C^e = C \times \mathbb{B}$ and function $H^e : D_2 \times C \rightarrow C \times \mathbb{B}$ as:

$$H^e(d_2, c) = \{c, T\{C'_o(c) \neq \phi\}\}, \quad (5)$$

where $\mathbb{B} = \{T, F\}$ is the boolean set, $d_2 \in D_2$ ($d_2 \subseteq N_2$) is the current state of the complementary program, and $T\{C'_o(c) \neq \phi\}$ is a true function that returns either true or false (T/F), depending on whether the enclosed condition is satisfied. Here, $C'_o(c) \neq \phi$ means that, on character c, there is a transition from a state $n_2 \in N_2$ to some states in N_1.

The underlying idea of the non-conflicting constraint is as follows. When the current state of the complementary program $d_2 = \{\}$, the current state of the main DFA d_1 has a single next state d'_1 for each character c. When $d_2 \neq \{\}, d_1$ can have multiple next states $d^*_1 = d'_1 \cup d''_1$, where d'_1 contains the next main states that are activated by the current main states in d_1, and d''_1 contains the next main states that are activated by the current complementary states in d_2. For a given d_1 and c, d'_1 is unique. However, with an arbitrarily selected N_2, for a given c, d''_1 can have different elements depending on the current d_2. Fortunately, under the non-conflicting constraint, for a given c, there can be at most one complementary state n_2 in d_2 that has a transition to one or several main states, $\{n_1\}$. This means that, regardless of d_2, for a given $c, d''_1 = \{n_1\}$ if n is active, and $d''_1 = \{\}$ otherwise. As a result, for each d_1 and c, there is no more than two next states of d_1 (i.e., d'_1 and optionally $d'_1 \cup \{n_1\}$), and we can logically regard that the size of the extended character set is $|C^e| = 2|C|$.

Theorem 2 (The Correctness of H^e). *With the set of complementary states N_2 being non-conflicting and H^e as defined in Equation 5, there exists a function $T^e(d_1, H^e(d_2, c))$ such that the DFA/EC has an equivalent DFA.*

Proof. Following the result of Lemma 1, let $M(d_2, c) = T_{22}(d_2, c)$, it is sufficient to prove that $T^e(d_1, H^e(d_2, c)) \equiv \{T_{11}(d_1, c) \cup T_{21}(d_2, c)\} \times T_{12}(d_1, c)$.

Let $T^e \equiv T^e_1 \times T^e_2$ and $T^e_2(d_1, H^e(d_2, c)) \equiv T_{12}(d_1, c)$. 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 $T^e_1(d_1, H^e(d_2, c)) \equiv$

$$\begin{cases} T_{11}(d_1, c), & H^e(d_2, c) = \{c, F\} \\ T_{11}(d_1, c) \cup T_{21}(\{C'_o(c)\}, c), & H^e(d_2, c) = \{c, T\} \end{cases}. \quad (6)$$

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, $H^e(d_2, c) = \{c, F\}, C'_o(c) = \phi$, and $T_{21}(d_2, c) = \phi$. In the case that $C'_o(c) \neq \phi, H^e(d_2, c) = \{c, T\}, T_{21}(\{C'_o(c)\}, c) = T_{21}(d_2, c)$ because $n = C'_o(c)$ is the only active complementary state in d_2 that has transitions to some states in N_1 on character c. To sum up, $T^e_1(d_1, H^e(d_2, c)) = T_{11}(d_1, c) \cup T_{21}(d_2, c)$ regardless of the value of $H^e(d_2, c)$. □

For the efficient implementation of DFA/EC, we have one more constraint on the selection of the complementary states N_2.

Definition 2 (Binary Complementary Set). *A complementary set N_2 is binary if each $n_2 \in N_2$ 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.

5.3 Determine the Complementary States

As discussed in Seciton 3, some NFA states are more likely to cause state explosion if they are included in a DFA implementation. 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. It is inefficient to find the optimal N_2 that minimizes $|D_1|$ since it requires the enumeration of all possible combinations of N_2 and the calculation of the corresponding D_1, which has a complexity of $O(|N|^{|N_2|} \times 2^{|N_1|})$.

In our previous work [22], we proposed a heuristic that uses scores to determine the candidates in the complementary states, where the score of each NFA state is based on the size of the character sets on its incoming transitions. In this paper, we propose a more precise method to determine the complementary states, which is named the *independent-state method*.

There are two steps in the selection of the complementary states. The first step is to estimate the extent to which each NFA state causes state explosion. The second step is to determine the complementary states based on the results in the first step and the two constraints that were introduced in the previous subsection. We start with the first step.

Definition 3 (Independent NFAStates). *Two NFA states are independent if they can be active independently. Specifically, two NFA states, u and v, are independent if there exist three active sets, A, B and C, such that $u \in A, v \in B, u \notin B, v \notin A$, and both $u \in C$ and $v \in C$.*

Algorithm 1 Determine the complementary states

```
1:  L ← the maximum size of N2
2:  N1 ← N
3:  N2 ← φ
4:  define priority(n) = (independent number of n ∈ N) / (# of n's transitions to n1 ∈ N1)
5:  Q ← N, sorted by decreasing priority(n), n ∈ N.
6:  for each n ∈ Q
7:      move n from N1 to N2
8:      if (N2 is the non-conflicting and binary)
9:          if (|N2| = L) return
10:     else
11:         move n from N2 back to N1
```

In other words, if two NFA states, u and v, are independent, u can be active or not, regardless of the status of v, and vice versa. On the other hand, if u and v are not independent, u and v either (1) cannot be active at the same time, or (2) one of them can only be active when the other is active.

Since the number of DFA states depends on the number of possible combinations of NFA states that can be active concurrently, the number of states of a DFA constructed from an NFA depends on the level of independence among the states in the NFA: (1) if every pair of states in the NFA is not independent, the size of the DFA equals that of the NFA, and (2) if every pair of states in the NFA is independent, the size of the DFA is $2^{|N|}$, where $|N|$ is the size of the NFA.

We measure the level of independence of an NFA state among other NFA states by using the number of times it appears in a pair of independent states, which we call the *independent number* of the state. It is not easy to enumerate all pairs of independent states directly. Because of this, we first list all pairs of states that can be concurrently active, and then remove from them the pairs of states in which one state is always active while the other state is active.

A pair of NFA states, u and v, can be concurrently active if one of the following conditions is true: (1) u and v are initially active, (2) u has a transition from state w on character c, v has a transition from state t also on character c, and either $w = t$ or w and t can be concurrently active.

A pair of NFA states, u and v, in which one is always active while the other is active, is a pair of states where one of the following conditions is satisfied: (1) u is always active, or (2) u has transitions to v, and for each character c on which there is a transition from u to v, there is also a transition on c from u to itself. An always active state is one that is initially active and has a transition to itself on every character. Note that the above two conditions are not inclusive, but they cover most of the cases.

In Table 2, we show the independent number for each state in the example NFA from Fig. 1. In this table, if we sort the NFA states in terms of their independent numbers and the number of cases that each NFA state duplicates the states in the corresponding DFA, respectively, we will find that the two resulting lists are the same. This example shows that we can use the independent number as a good suggestion for the priority in which each NFA state is selected into the set of complementary states.

The complementary states selection algorithm is listed in Algorithm 1. In a nutshell, this algorithm greedily adds NFA states with large independent numbers into the set of complementary states N_2, as long as the non-conflicting constraint and the binary constraint are satisfied. Considering the non-conflicting constraint, which requires that two states that have transitions to the states in N_1 on the same characters cannot co-exist in N_2, we divide the independent number of each state n by the number of n's transitions to the states in N_1 as a penalty for the states that might potentially exclude a large number of other states from N_2. At first glance, the non-conflicting constraint may make many NFA states ineligible to N_2. Fortunately, using the method above, the non-conflicting constraint excludes few states from N_2 in practical rule-sets with a large number of regular expressions.

The complexity of the complementary selection algorithm is $O(|N|^2)$, which is the complexity in the calculation of the independent numbers. The complementary selection algorithm is neglectably fast compared to that of the DFA construction algorithm, $O(2^N)$, which is a part of the DFA/EC construction algorithm.

5.4 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 M are implemented by the complementary program.

We show the implementation of function M first, which is followed by H^e. From Theorem 1, it is required that $M = T_{22}$ for the equivalence of DFA/EC and DFA. Firstly, 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 $j = i + 1$. Secondly, we represent the states in N_2 with an array of bits, and we use the ith bit to represent state n_i, which means that n_i cannot have transitions to any state in N_2 except for n_{i+1}. Thirdly, we can represent the transitions within N_2 with two sets of bit masks, M_{20} and M_{22}. For each character $c \in C$, $M_{20}[c]$ and $M_{22}[c]$ are the bit masks for c. The ith bit in $M_{20}[c]$ being one means that state n_i has a transition to itself on character c, and the ith bit in $M_{22}[c]$ 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 ith bit being one or zero representing whether state n_i is active; then, the next complementary states that are activated by the current complementary states can be calculated by $d_2' = M(d_2, c) = (d_2 \& M_{20}[c]) | ((d_2 \& M_{22}[c]) \gg 1)$, where $\&, |, \gg$ are the bitwise AND, OR, SHIFT operations, respectively. Clearly, $M \equiv T_{22}$ implements the transitions within N_2.

Similarly, we define another set of bit masks M_{21} for different $c \in C$, and the ith bit in $M_{21}[c]$ being one means that the state n_i has a transition to some main states in N_1 on character c. Then, $H^e(d_2, c) = \{c, T\{d_2 \neq \{\}, M_{21}[c] \neq 0\}\}$. The masks M_{20}, M_{22}, and M_{21} of the DFA/EC in Fig. 2 are shown in binary digits in Table 3(b–d), respectively.

Algorithm 2 The DFA/EC simulator

1: $e \leftarrow (d_2 \& M_{21}[c]) \neq 0$

2: $\{d_1^*, d_2'\} \leftarrow Tx[d_1][c \cdot e]$

3: $d_2^* \leftarrow (d_2 \& M_{20}[c]) | ((d_2 \& M_{22}[c]) \gg 1) | d_2'$

The main DFA, which implements function T_1^e, contains 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. Each entry in the transition table is d_1, d_2', where d_1 is the next main DFA state, and d_2' is part of the next complementary states activated by the current main states. d_2' can be represented by a bit-array, and b_2' can also be represented by an index to save space since the number of d_2' is very limited in practice. The pseudo code for the execution of a DFA/EC is listed in Algorithm 2, where d_1^* and d_2^* are the new DFA/EC state, and e is the extra bit that represents the value of function H^e. The concatenation $c \cdot e$ is the extended character created from c and e.

TABLE 3
Tables for the DFA/EC in Fig. 2; Numbers with Subscript B Are Binary Numbers

(a) Main DFA table T^e

state#	$d_1 \subseteq N_1$	[^AC-N]	A	[CD]	[E-GIJL]	H	K	[MN]
accept 0	(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
start 2	(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) Masks M_{21}

	K
A_{21}	100_B

(c) Masks M_{22}

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

(d) Masks M_2

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

Algorithm 3 The construction of the main DFA table

1: $D \leftarrow$ the set of conventional DFA states
2: **for each** (d **in** D)
3: $d_1 \leftarrow d \cap N_1$
4: **for each** (c **in** C)
5: $d'_1 = T(d_1, c) \cap N_1$
6: $d''_1 = T(d_1 \cup N_2, c) \cap N_1$
7: $d'_2 = T(d_1, c) \cap N_2$
8: $Tx[d_1][c \cdot F] = \{d'_1, d'_2\}$
9: $Tx[d_1][c \cdot T] = \{d''_1, d'_2\}$

5.5 The Construction of DFA/EC

The data structures needed to be constructed for a DFA/EC are: the main DFA table Tx, the sets of bit-masks M_{20}, M_{22}, and M_{21}. The construction of the main DFA table, which implements function T^e_1, as defined in Equation 6, is shown in Algorithm 3, where we regard each DFA state as a set of NFA states and assume T to be a function that returns the next set of active NFA states, given the current 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 of the combinations of main states in N_1 can be simultaneously active.

The transition table Tx of the main DFA and the masks M_{20}, M_{22}, and M_{21} of our example DFA/EC in Fig. 2 are shown in Table 3. In Table 3(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 of the remaining columns are transitions. Each cell in the transition table consists of three values, which are the results of the functions $T^e_1(d_1, \{c, F\}), T^e_1(d_1, \{c, T\})$, and $T^e_2(d_1, \{c, T/F\})$, respectively. As we can see in Table 3(a), values of $T^e_1(d_1, \{c, F\})$ and $T^e_1(d_1, \{c, T\})$ are represented by indexes, and they are equal in most cases. Values of $T^e_2(d_1, \{c, T/F\})$ 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].

5.6 Overhead in Storage and Computation

Let us first discuss the memory storage 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 $d'_2 = T^e_2(d_1, H^e(d_2, c)) = T_{12}(d_1, c)$ 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 M_{12}, are very limited in number, and we can use the index of d'_2 in M_{12} to represent d'_2. Therefore, the total size of the transition table is $|D_1| \times |C| \times (2 \times \lceil \log_2 |D_1| \rceil + \lceil \log_2 |M_{12}| \rceil)$ bits, and the memory bandwidth is $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |M_{12}| \rceil$ 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.

Secondly, we discuss the computation overhead of DFA/EC down to the level of individual instructions. From Algorithm 2, for each byte in the payload, DFA/EC performs the following instructions: a single access to the transition table Tx in the main memory, a multiple and an addition instruction to calculate the offset $c \cdot e$ in the transition table, a right-shift and a bitwise and instruction to obtain d^*_1 and d'_2, three instructions to load bit-masks $M_{21}[c], M_{20}[c]$, and $M_{22}[c]$ from the on-chip memory or cache, three bit-wise and instructions, four bit-wise or instructions, a zero-test instruction, and a right-shift instruction to obtain e and d^*_2. To sum up, there are one main memory access and 16 other instructions: three cache accesses, one integer multiple, one integer addition, four bit-wise and, four bit-wise or, two right-shift, and one zero-test.

In general-purpose processor architectures, the main memory access is often the bottleneck due to the ever-widening gap between the speed of processor and memory. Therefore, significant speed up of DFA/EC can be expected in multi-core single-memory platforms and in cheap platforms with limited cache. DFA/EC can also be implemented in hardware architecture, such as an FPGA coupled with a memory bank. The simple logic needed to implement in DFA/EC reduces the need for LUTs compared to an NFA, which can lead to higher operating frequencies. The parallelism available in hardware allows the processing of the instructions in Algorithm 5.4 to be completed in one memory cycle.

6 Evaluation

In our experiment, we endeavored the following efforts: Firstly, we developed several compilers, which read files of rules and created the corresponding inspection programs and the transition tables for DFA, MDFA [1], H-FA [9], and DFA/EC. Secondly, we extracted rule-sets from the Snort [6], [7] rules. Thirdly, we developed a synthetic payload generator. We generate the inspection programs for the rule-sets, measure their storages, and load 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 requirement and memory bandwidth. We did not implement XFA [20] because the XFA compiler, which employs complicated compiler optimization technologies, is not available.

6.1 Evaluation Settings

Our compilers contain a regular expression compiler. All Perl-compatible features, except back-references 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.[1] Rules in Snort have been classified into different categories. We adopt subsets of the rule-set in five categories, such that each rule-set can be implemented by a single DFA with less than 2GB of memory. Almost all patterns in our rule-sets contain repetitions on large character-sets. Since counter-constraints are not supported, we replace all counter-constraints with. $*$ enclosures.

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 traffic [23], which prevent the DFA from being traveled only its low-depth states, as it does in normal traffic. We do not show the results with normal traffic since they result in similar performances across all inspection programs, as only a small number of shallow states are traveled in normal traffic.

1. We use the rule-sets released Dec. 2009. We expect that similar experimental results will be observed using newer rule-sets.

TABLE 4
The Total Number of States (Percentage to DFA)

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	343k	0.04%	2%	0.3%	0.1%
smtp-36	141k	0.5%	6%	0.6%	0.3%
spyware-put-93	269k	1%	18%	5%	1%
web-client-35	106k	1%	14%	1%	0.7%
web-misc-28	453k	0.07%	3%	0.3%	0.1%

6.2 Results on Storage Size

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 4, 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, an order of magnitude smaller than a 4DFA, and comparable to that of an 8DFA. The significant reduction is due 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 with the optimal transition encoding technique, which is not our focus. As shown in Table 6, 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, 3 times smaller than a 4DFA, and comparable to that of an 8DFA.

We measure the total minimum memory (storage) 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 $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |M_{12}| \rceil$, respectively. Here, D is the set of DFA states, D_i is the set of DFA states in the ith DFA of a MDFA, D_1 is the set of main DFA states of a DFA/EC, and M_{12} is the set of masks of a DFA/EC required to implement the transition function T_1^e. As shown in Table 7, 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 states 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.

TABLE 5
The Size of the Per-Flow State (Bits)

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	19	40	25	32	48
smtp-36	18	42	22	31	46
spyware-put-93	19	45	27	43	70
web-client-35	17	43	24	37	56
web-misc-28	19	41	26	36	56

TABLE 6
The Total Number of Transitions (Percentage to DFA)

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	7m	0.03%	1%	0.1%	0.02%
smtp-36	2m	0.5%	6%	0.2%	0.07%
spyware-put-93	7m	1%	7%	2%	0.3%
web-client-35	2m	1%	7%	0.6%	0.2%
web-misc-28	8m	0.06%	2%	0.1%	0.04%

TABLE 7
The Transition Storage (Bits/Percentage to DFA)

	DFA	DFA/EC	2DFA	4DFA	8DFA
exploit-19	124m	0.02%	1%	0.06%	0.008%
smtp-36	37m	0.4%	4%	0.1%	0.02%
spyware-put-93	141m	1%	6%	1%	0.1%
web-client-35	40m	1%	6%	0.3%	0.1%
web-misc-28	155m	0.04%	1%	0.08%	0.01%

Here, N_2 is the number of complementary states in a DFA/EC. As shown in Table 5, DFA/EC has a small size per-flow state in terms of both bits and words.

6.3 Results on Memory Bandwidth and Speed

Memory bandwidth is the amount of memory accesses per byte in the payload, which we measure in terms of bits. The memory bandwidths of DFA, MDFA, and DFA/EC are $\lceil \log_2 |D| \rceil$, $\sum_{i=1}^{M} \lceil \log_2 |D_i| \rceil$, and $\lceil \log_2 |D_1| \rceil + \lceil \log_2 |M_{12}| \rceil$, respectively. Fig. 3 shows that the memory bandwidth of DFA/EC is very close to that of DFA and is much smaller than MDFAs. Moreover, It is exciting to see that the memory bandwidth of DFA/EC can even be smaller than DFA in rule-sets exploit-19 and web-misc-28. Memory bandwidth suggests the amount of information about a transition that the inspection program needs to obtain from the transition table. The reason that DFA/EC sometimes have a smaller memory bandwidth than DFA is that the complementary program of a DFA/EC contains some of the transition information that otherwise needs to be stored in the transition table.

In Fig. 4, we show the number of main memory accesses per KB of payload. DFA/EC and DFA have the minimum number of main memory accesses, while those of MDFAs increase in proportional to M.

We measure the speed of the inspection programs with both Java and C++ implementations in a Unix machine with 16GB of 1333 MHz DDR3 memory and a 2.66 GHz Intel Core

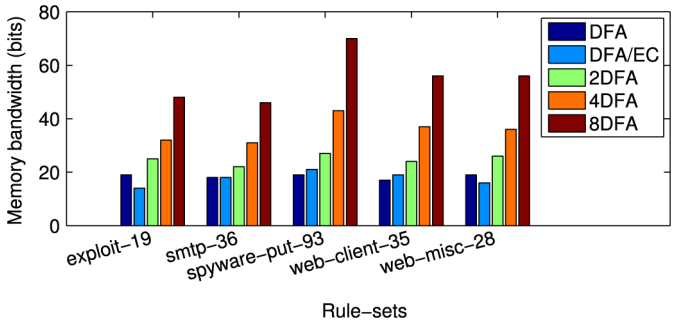


Fig. 3. Memory bandwidth (bits) with different rule-sets.

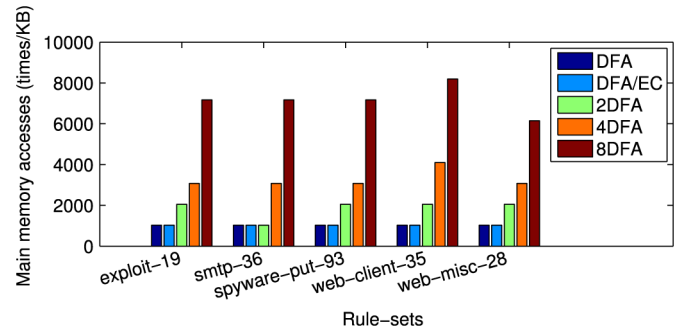


Fig. 4. Memory accesses (times/KB) with different rule-sets.

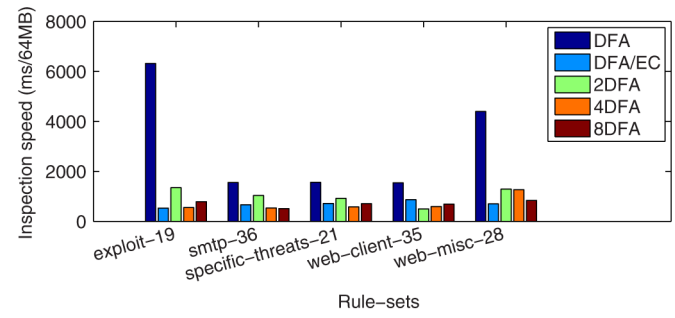


Fig. 5. Inspection speed (Java) with different rule-sets (milliseconds per 64MB).

i5 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 their amounts of cache or on-chip memory.

Results are shown in Figs. 5 and 6. 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 relatively large amount of cache memory in our platform. We believe that DFA/EC will be more favorable for the implementations on ASIC hardware or GPUs that have less cache memory and more computation resources.

6.4 Summary

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

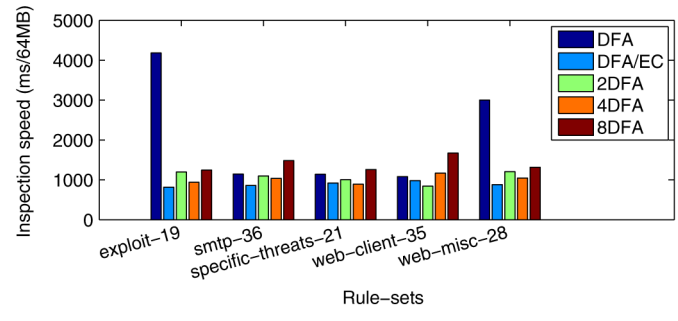


Fig. 6. Inspection speed (C++) with different rule-sets (milliseconds per 64MB).

7 Conclusion

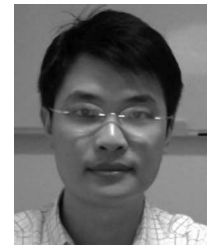
In this paper, we investigated a general-purpose processor and regular expressions-based deep packet inspection algorithm, called deterministic finite automata with extended character-set (DFA/EC). Unlike existing state reduction algorithms, our solution requires only a single main memory access for each byte in the traffic payload, which is the minimum. We performed experiments with several Snort rule-sets and synthetic payloads. Experiment results show that DFA/ECs are very compact, they are over four orders of magnitude smaller than a DFA in the best cases, 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.

Acknowledgment

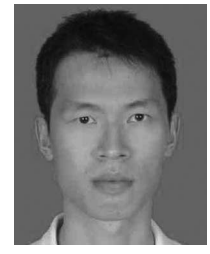
This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61003241, 6103045, 61370021, 61003296, in part by Natural Science Foundation of Guangdong Province, China under Grant S2013010011905, in part by Shenzhen Overseas High-level Talents Innovation and Entrepreneurship Funds under Grant No. KQC201109050097A, and in part by by the NSF grants ECCS 1231461, ECCS 1128209, CNS 1138963, CNS 1065444, and CCF 1028167.

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. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (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. ACM SIGCOMM*, Sep. 2006.
[3] A. Bremler-Barr, D. Hay, and Y. Koral, "Compact DFA: Generic state machine compression for scalable pattern matching," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2010.
[4] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2004.
[5] K. Namjoshi and G. Narlikar, "Robust, and fast pattern matching for intrusion detection," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2010.
[6] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. 13th Syst. Admin. Conf.*, Nov. 1999.
[7] Snort [Online]. Available: http://www. Snort.org/
[8] V. Paxson, "Bro: A system for detecting network intruders in real-time," in *Proc. Comput. Netw.*, Dec. 1999.
[9] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2007.
[10] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. Conf. Emerging Netw. Exp. Technol. (CoNEXT)*, 2007.
[11] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher, "HEXA: Compact data structures for faster packet processing," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, 2009.
[12] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proc. Int. Conf. Security Privacy Commun. Netw. (Securecomm)*, 2008.
[13] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2001.
[14] B. L. Hutchings, R. Franklin, and D. Carver, "Assisting network intrusion detection with reconfigurable hardware," in *Proc. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, 2002.
[15] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuit for matching complex network intrusion detection patterns," in *Proc. Int. Conf. Field-Program. Logic Appl. (FLP)*, 2003.
[16] B. Brodie, R. Cytron, and D. Taylor, "A scalable architecture for high-throughput regular-expression pattern matching," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2006.
[17] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2007.
[18] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast, and scalable deep packet inspection," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, 2006.
[19] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection, and prevention," in *Proc. Int. Symp. Comput. Archit. (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. ACM SIGCOMM*, 2008.
[21] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proc. Conf. Emerging Netw. Exp. Technol. (CoNEXT)*, 2008.
[22] C. Liu, A. Chen, D. Wu, and J. Wu, "A DFA with extended character-set for fast deep packet inspection," in *Proc. Int. Conf. Parallel Process. (ICPP)*, 2011.
[23] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2008.



Cong Liu received the BS degree in microelectronics from South China University of Technology, Guangzhou, China, in 2002, the MS degree in computer software and theory from Sun Yat-sen University, Guangzhou, China, in 2005, the PhD degree in computer science from Florida Atlantic University, Boca Raton, FL. He is an associate professor with the School of Supercomputing, Sun Yat-sen University. His research interests include delay tolerant networks (DTNs) and deep packet inspection.



Yan Pan received the BS degree in information science and the PhD degree in computer science from Sun Yat-sen University, Guangzhou, China, in 2002 and 2007, respectively. Currently, he is an associate professor with the School of Software, Sun Yat-Sen University. His research interests include machine learning algorithms, learning to rank, and computer vision.

Ai Chen received the BS and MS degrees both in electronic engineering from Tsinghua University, Beijing, China, in 2001 and 2004, respectively. He received the PhD degree in computer science and engineering from the Ohio State University, Columbus. Currently, he is an associate researcher with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Beijing, China. His research interests include wireless and sensor networks, mobile computing, and smartphone applications.

Jie Wu is the chair and a Laura H. Carnell professor with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA. Prior to joining Temple University, he was a program director with the National Science Foundation and a distinguished professor with Florida Atlantic University, Boca Raton, FL. He regularly publishes in scholarly journals, conference proceedings, and books. His research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He serves on several editorial boards, including IEEE Transactions on Computers, IEEE Transactions on Service Computing, and Journal of Parallel and Distributed Computing. He was general cochair/chair for IEEE MASS 2006, IEEE IPDPS 2008, and IEEE ICDCS 2013, as well as program cochair for IEEE INFOCOM 2011 and CCF CNCC 2013. Currently, he is serving as a general chair for ACM MobiHoc 2014. He was an IEEE Computer Society distinguished visitor, ACM distinguished speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). He is a CCF distinguished speaker and a fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.