Fast Deep Packet Inspection with a Dual Finite Automata

Cong Liu[†]
[†]Sun Yat-sen University
gzcong@gmail.com

Jie Wu[‡]
[‡]Temple University
jiewu@temple.edu

Abstract—Deep packet inspection, in which packet payloads are matched against a large set of patterns, is an important algorithm in many networking applications. Non-deterministic Finite Automaton (NFA) and Deterministic Finite Automaton (DFA) are the basis of existing algorithms. However, both NFA and DFA are not ideal for real-world rule-sets: NFA has the minimum storage, but the maximum memory bandwidth; while DFA has the minimum memory bandwidth, but the maximum storage. Specifically, NFA and DFA cannot handle the presence of character sets, wildcards, and repetitions of character sets or wildcards in real-world rule-sets. In this paper, we propose and evaluate a dual Finite Automaton (dual FA) to address these shortcomings. The dual FA consists of a linear finite automaton (LFA) and an extended deterministic finite automaton (EDFA). The LFA is simple to implement, and it provides an alternative approach to handle the repetition of character sets and wildcards (which could otherwise cause the state explosion problem in a DFA) without increasing memory bandwidth. We evaluate the automaton in real-world rule-sets using different synthetic payload streams. The results show that dual FA can reduce the number of states up to five orders of magnitude while their memory bandwidth is close to minimum.

Index Terms—Deep packet inspection, linear finite automaton (LFA), dual finite automaton (dual FA).

1 Introduction

Deep packet inspection, in which packet payloads are matched against a large set of patterns, is an important algorithm in many networking applications. Some applications, such as exact string matching, have been well-studied. However, in many others, the design of high-performance pattern matching algorithms is still challenging. Networking applications that require high performance inspection include: network intrusion detection and prevention systems [1], [2], [3], [4], [5], content-based routing [6], e-mail scanning systems [7], etc. In these applications, packet payloads must be inspected against a large set of patterns (e.g., a rule-set with thousands of regular expressions) at a very high speed (e.g., several gigabits per second). Due to their wide application, there is substantial research [8], [9], [10], [11], [12] on high-speed deep packet inspection algorithms in the literature.

In this paper, we focus on deep packet inspection algorithms in network intrusion detection and prevention systems. Because regular expression is expressive and is able to describe a wide variety of patterns, the focus of the research community has recently moved from exact string matching to regular expression matching. A substantial amount of work has been devoted to the architectures, data structures and algorithms to support fast deep packet inspection against a rule-set of a large number of regular expressions. The main challenge with fast deep packet inspection is to minimize the memory bandwidth and reduce the storage to an acceptable level at the same time.

Inspection programs based on regular expressions [13] are typically implemented by two classic *finite automata* (FA): *non-deterministic finite automata* (NFA) and *deterministic finite automata* (DFA). Each of them has its strengths and weaknesses, but neither one is ideal to implement in a general-purpose processor for real-world rule-sets. NFA has the benefit of limited storage, which is proportional to the total number of characters in the rule-set. However, it has a non-deterministic number of concurrently active states, which results in $O(N)$ main memory accesses to process each byte in the payload, where N is the number of states in the NFA. This high memory bandwidth makes NFA unacceptable in high speed applications. On the other hand, DFA requires a single (the minimum) main memory access for each byte in the payload, which is independent of the number of regular expressions in the rule-set. Unfortunately, DFA is not ideal for real-world rule-sets [8], [14] either, due to its maximum $O(2^N)$ storage. The number of DFA states can be exponential to the number of characters N in the rule-set.

Several characteristics in many real-world rule-sets [1], [2], [3], [4], [5], [6], [7], such as the repetition of sub-patterns, make the implementation of fast inspection programs a challenging task. Much work has focused on different compression techniques [8], [9], [14], [15], [16], [17], [18], [19], [20], which reduce the number of DFA states or reduce the number of transitions on the cost of increased memory bandwidth. In this paper, we propose a novel algorithm, called *dual finite automata* (dual FA). Dual FA has a much smaller storage demand compared to DFA, and it requires the minimum number of mem-

Digital Object Indentifier 10.1109/TC.2011.231

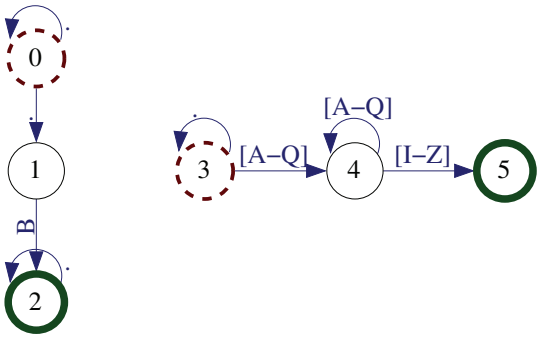


Fig. 1. The NFAs for ".+B.*" and ".*[A-Q]+[I-Z]", respectively.

ory accesses among existing compressed FAs: one, or sometimes, two main memory accesses for each byte in the payload. This is due to the fact that dual FA can efficiently handle unbounded repetitions of wildcards and character sets using a novel *linear finite automata* (LFA).

Dual FA is a state compression technique [8], [18], and state compression is the most fundamental and challenging compression technique in reducing DFA size. Dual FA is a simple and novel state compression technique, which can combine with other compression techniques, such as transition compression [9], [17] and alphabet compression [17], [19], [20]. Specifically, our contributions are in both theory and practice and are summarized as follows:

1) We develop a *linear finite automata* (LFA) that handles unbounded repetitions without a transition table (see Section 3) and therefore requires no main memory access.
2) We propose a dual FA, which contains an LFA and an extended DFA (EDFA). Dual FA requires only one (or sometimes two) memory accesses per byte in the payload.
3) We evaluate the performance of dual FA with real-world rule-sets and synthetic payload streams. The results show that dual FA can reduce the number of states and transitions up to five orders of magnitude.

The remainder of this paper is organized as follows: Section 2 provides necessary background. Section 3 describes our contributions with motivating examples. We discuss implementation details in Sections 4. In Section 5, we present evaluations in real-world rule-sets. Section 6 reviews related work. We conclude in Section 7.

2 Background & Motivation

2.1 Regular Expression

A regular expression, often called a *pattern*, is an expression that describes a set of strings. Features found in regular expressions derived from commonly used anti-viruses and network intrusion detection systems include exact-match substring, character sets and simple wildcards, repetitions of simple characters, character sets, wildcards, etc. We will use two regular expressions ".+B.*" and ".*[A-Q]+[I-Z]" in our automaton construction examples throughout this paper.

An exact match substring is a fixed size pattern, which occurs in the input text exactly as it is. In our example, "B" is an exact match substring.

Character sets are found in regular expressions as [c_i-c_j] expressions. The set matches any character between c_i and c_j. A wildcard, which matches any character, is represented by a dot. In our example, "[A-Q]" matches any character between A and Q, and "." matches any character.

Simple character repetitions appear in the form $c+$ (c repeats at least once) or $c*$ (c repeats 0 or any times), where c is a character of the alphabet. Repetitions of character sets and wildcards may cause the *state explosion problem*, which will be discussed later. Examples are ".*" and "[A-Q]+".

2.2 Non-deterministic Finite Automaton (NFA)

NFA and DFA are pattern matching programs that implement a set of one or more regular expressions. In Figure 1, we show the NFAs accepting the regular expressions in our example. To evaluate an FA, either an NFA, DFA, or others, two important metrics are (1) the storage, i.e. the amount of main memory needed to store the transition table, and (2) the memory bandwidth, i.e. the number of main memory accesses per byte in the payload, which is critical to inspection speed.

The storage of an FA is proportional to its number of states and its number of transitions per state. In an NFA, the number of states is not greater than the number of characters in the regular expressions in the rule-set. This is true even when the regular expressions contain repetitions and character sets.

To calculate the memory bandwidth, we need to know how the NFA works. The pattern matching operation starts from the entry states that are initially active. In Figure 1, the entry states are 0 and 3. A match is reported every time an accepting state (with a bold rim) is traversed. All outgoing transitions of the active states, labeled with the next byte in the payload, are taken. Notice that, in NFA, each state can have more than one transition on any given character, and many states can be active in parallel. We will call the set of simultaneously active NFA states the *active set*. Since every state traversal involves a main memory access, the size of the active set gives a measure of the memory bandwidth.

As an example, we show how the NFA for ".+B.*" (Figure 1) processes the input text "DCBA". This NFA traversal results in the following sequence of active sets:

$$(0) \xrightarrow{D} (0,1) \xrightarrow{C} (0,1) \xrightarrow{B} (0,1,2) \xrightarrow{A} (0,1,2)$$

In this example, the largest active set size is 3 (the maximum active set includes all of the states in the

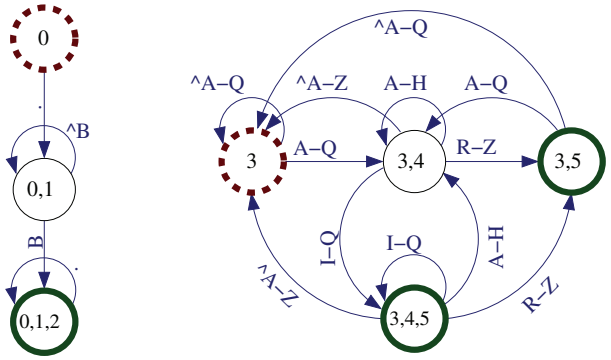


Fig. 2. The DFAs for ".+B.*" and ".*[A-Q]+[I-Z]", respectively. We abbreviate the bracket expression, e.g., we use "A-H" as an alias for the character set "[A-H]".

NFA). Theoretically, NFA has the maximum memory bandwidth.

2.3 Deterministic Finite Automaton (DFA)

Different from an NFA, in a DFA, each state has exactly one outgoing transition for each character of the alphabet. Consequently, its active set always consists of a single state. The DFAs for regular expressions ".+B.*" and ".*[A-Q]+[I-Z]", respectively, are shown in Figure 2. In this figure, "[^A-Q]" is the complementary character set of "[A-Q]". These DFAs can be constructed from the NFAs in Figure 1, respectively, using the standard subset construction routine [13], in which a DFA state is created for each possible active set of NFA states. In Figure 2, the label inside each DFA state shows the active set of NFA states, for which the DFA state is created.

2.4 The State Explosion Problem

The *state explosion problem* is: when several regular expressions containing repetitions of character sets are compiled together into a single DFA, the number of DFA states may increase exponentially [8]. In this case, a single DFA is not ideal due to the huge amount of storage needed to store its transition table. In the worst case, for a regular expression whose NFA has N states, the number of active sets (or DFA states) can be $O(2^N)$. Note that in reality this may not happen because not all of the 2^N combinations are possible active sets.

Here, we provide a slightly different explanation: the state explosion problem [8], [9], [10], [11], [12] in a DFA is caused by the fact that a large number of NFA states can be active independently.

Definition 1 (Independent NFA states): 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 $u \in A$ and $v \notin A$, $u \notin B$ and $v \in B$, and $u \in C$ and $v \in C$.

As an example, in the NFA of ".*[A-Q]+[I-Z]" (as shown in Figure 1), states 4 and 5 are independent

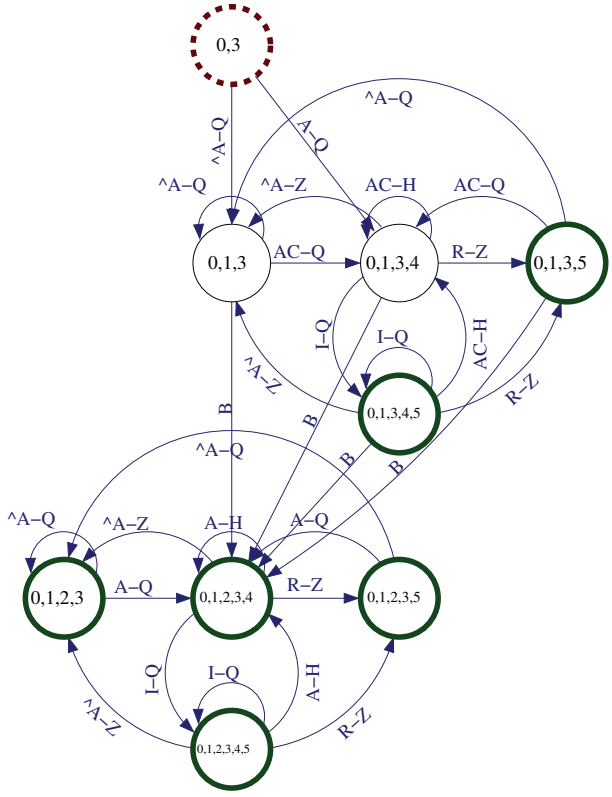


Fig. 3. A single DFA for ".+B.*" and ".*[A-Q]+[I-Z]".

because, in the corresponding DFA (Figure 2), there exist active sets (3,4), (3,5) and (3,4,5). On the other hand, in the NFA of ".+B.*", each pair of states is not independent, since in all of its active sets (i.e., (0), (0,1) and (0,1,2)), all sets contain state 0, and all sets containing state 2 also contain state 1.

Independent NFA states cause an increase in the number of states in the corresponding DFA, as we have seen in Figure 2 (right). Moreover, when combining both of the NFAs (Figure 1) into a single DFA (Figure 3), the increase becomes more significant. This increase occurs because state 2 in the first NFA is independent of all states in the second NFA. The consequence is that all states in the second DFA (on the right of Figure 2) show twice in the combined DFA (Figure 3).

In general, NFA states that are independent of many other NFA states may cause a massive increase in the number of DFA states. This is because: (1) in the sub-set construction algorithm, which constructs a DFA from an NFA, a DFA state is created to represent each possible set of NFA states that can be concurrently active, and (2) the independent NFA states, which can be active or inactive independent of other NFA states, increase the number of possible combinations of concurrently active NFA states, i.e., the number of DFA states. Therefore, as the number of independent NFA states increases, the state explosion problem becomes more significant. We will discuss how to identify these independent NFA states later.

State explosion can be mitigated by dividing the rules

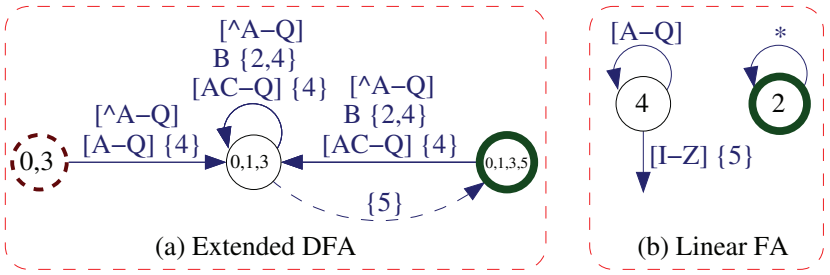


Fig. 4. The dual FA for ".+B.*" and ".*[A-Q]+[I-Z]" with the size of the LFA being 2.

TABLE 1
Transition tables of the dual FA in Figure 4.

(a) EDFA.

	1^{st} Tx			2^{nd} Tx
	[^A-Q]	B	[AC-Q]	{5}
(0,3) START	(0,1,3)	(0,1,3) {4}	(0,1,3) {4}	
(0,1,3)	(0,1,3)	(0,1,3) {2,4}	(0,1,3) {4}	(0,1,3,5)
(0,1,3,5) ACC	(0,1,3)	(0,1,3) {2,4}	(0,1,3) {4}	

(b) LFA.

		[^A-Z]	[A-H]	[I-Q]	[R-Z]
2 ACC	self mask	1	1	1	1
	next mask	0	0	0	0
4	self mask	0	1	1	0
	next	0	0	0	0
	extern mask {5}	0	0	1	1

into different groups and compiling them into respective DFAs [8]. However, there are two problems with this approach: first, it increases the memory bandwidth since each of these DFAs must access the main memory once for each byte in the payload. Second, the state explosion problem might occur even when compiling a single rule, such as ".+B.*|.*E[A-Q]+G".

This paper focuses on state compression. For other compression techniques, please refer to [9], [17] (transition compression) and [17], [19], [20] (alphabet compression).

3 Dual Finite Automata

We use a different method, called *dual finite automata* (dual FA), to mitigate the state explosion problem. First, we identify those NFA states that cause the state explosion problem in DFA, i.e., those that are independent to a large number of other states. Then, we use a *linear finite automaton* (LFA, Definition 2) to implement these NFA states. Then, we compile the rest of the NFA states into a single *extended DFA* (EDFA), whose size is significantly reduced after getting rid of those "problematic" NFA states. Finally, since the LFA and the EDFA cannot work separately, we implement an interaction mechanism. The fact that the LFA and the EDFA cannot work separately is because a pair of NFA states, implemented by the LFA and the EDFA, respectively, may have transitions between them. The difference between EDFA and DFA is that EDFA has extra features to support the interaction mechanism. Let us start with the LFA.

Definition 2 (Linear finite automaton): a linear finite automaton (LFA) is an NFA in which each state can have transitions from (to) at most one other state, excepts self-transitions.

An LFA implementing an NFA with k states (with k being small) can be very memory efficient. It does not require a huge DFA transition table containing entries for all of the $O(2^k)$ states. We use k bits to represent an active set, and we use $2C$ of bit-masks of k bits to calculate all of the state transitions, where C is the alphabet size. Specifically, (1) each of the k NFA states is represented by a bit, and the k bits are represented by a bit-array l; (2) the states are ordered such that if state u can transit to state v and state u is represented by the i^{th} bit, then state v must be represented by the $(i-1)^{th}$ bits (the $(i-1)^{th}$ bit is to the right of the i^{th} bit); (3) a k-bits mask, $self[c]$, is associated with each character c in the alphabet whose i^{th} bit is 1 only if the i^{th} state has a transition to itself on character c; (4) a k-bits mask, $next[c]$, is associated with each character c in the alphabet whose i^{th} bit is 1 only if the i^{th} state has a transition to the $(i-1)^{th}$ state on character c.

Given the active set states l (represented by bit-array), the next byte c in the payload, and the masks $self[c]$ and $next[c]$; the next active set can be calculated using the following bitwise operations: $(l\&self[c])|((l\&next[c]) \gg 1)$, where $\&$, $|$, and $\gg$ are bitwise operators *and*, *or*, and *right-shift*, respectively. In a general-purpose processor (or in ASIC hardware), the $2C$ of bit-marks used can be stored in the cache (or on-chip memory) instead of in main memory (or external memory). LFA is fast for eliminating main memory accesses. Also, since we use LFA to implement those NFA states that cause state explosion in DFA, the EDFA, which implements the remaining DFA states, becomes very compact.

In Section 4.1, we will analyze how to identify the NFA states that cause state explosion. We can see that they include the NFA states corresponding to the unbounded repetitions in regular expressions. This leads to one of our contributions: LFA handles unbounded repetitions without a transition table.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

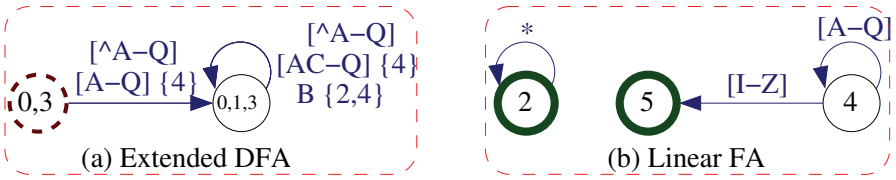


Fig. 5. The dual FA for ".+B.*" and ".*[A-Q]+[I-Z]" with the size of the LFA being 3.

TABLE 2
Transition tables of the dual FA in Figure 5.

(a) EDFA.

	1^{st} Tx		
	[^A-Q]	B	[AC-Q]
(0,3) START	(0,1,3)	(0,1,3) {4}	(0,1,3) {4}
(0,1,3)	(0,1,3)	(0,1,3) {2,4}	(0,1,3) {4}

(b) LFA.

		[^A-Z]	[A-H]	[I-Q]	[R-Z]
2 ACC	self mask	1	1	1	1
	next mask	0	0	0	0
5 ACC	self mask	0	0	0	0
	next mask	0	0	0	0
4	self mask	0	1	1	0
	next mask	0	0	0	0

3.1 The Differences to Existing Methods

Here, we provide a brief discussion on the differences between the proposed dual FA and existing methods. In [8], NFAs are divided into M groups, which are then compiled into M DFAs, respectively. While in the dual FA, the problematic NFA states are separated and are implemented by the LFA to reduce the storage of the DFA.

Using auxiliary variables, such as the LFA, to devise a compact and efficient inspection program is challenging. Two seminal papers, H-FA [18] and XFA [21], use auxiliary variables to represent the "factored out" auxiliary states and to 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 [18] 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, LFA uses a single piece of program to generate its next state, and dual FA requires at most two memory accesses for each byte in the payload.

3.2 Examples of Dual Finite Automata

Dual FA can be better illustrated directly by examples. In Figure 4, we show a dual FA with an LFA, whose size is two (which means the LFA implements two NFA states). A dual FA needs to maintain two states for its LFA (with a bit-array) and its EDFA (with an integer state index), respectively. In the figure, each EDFA state represents a subset (which does not contain any LFA state) of a possible active set of the NFA. For example, since (0,1,2,3) is an active set for the NFAs and 2 is not in any EDFA state because it belongs to the LFA, (0,1,3) is a state in EDFA. Each EDFA transition is labeled with a character set and, optionally, a set of LFA states. In Figure 4, the transition labeled with "[A-Q] {4}" means that state (0,3) transits to state (0,1,3) on any character $c \in$[A-Q] and, while this transition occurs, the LFA state 4 will be set to active.

According to Definition 2, each LFA state can have at most three transitions. The first is a self transition, e.g., the transition from state 4 to itself, which is labeled with "[A-Q]" in Figure 4(b). The second is a transition to another LFA state. This transition does not exist in the example shown in Figure 4(b) but exists in another example shown in Figure 5(b) (the transition labeled with "[I-Z]"). The third transition does not transit to any LFA state, but it sets active an NFA state that is implemented by the EDFA, e.g., the transition labeled with "[I-Z]{5}" in Figure 4(b). Corresponding to this type of transition, there is an additional type of transitions in the EDFA that a normal DFA does not have, which are called *secondary transitions*. An example of a secondary transition is shown with a dashed-line in Figure 4. The label "{5}" on this transition suggests that, when NFA state 5, implemented by the EDFA, is set to active (by the LFA), the EDFA transits from state (0,1,3) to (0,1,3,5). To provide more details on the implementation, we show the transition tables for the dual FA in Table 1.

Another example of dual FA with an LFA of size three is shown in Figure 5, which is functionally equivalent to the NFAs in Figure 1 and the dual FA in Figure 4. Its transition table is shown in Table 2. Note that, in this example, no secondary transition table is required for the EDFA, which means that this dual FA has exactly one main memory access per byte in the payload.

3.3 The Execution of Dual FA

In the following, we will show how the LFA and the EDFA interact with each other when consuming a byte in the payload, so that the dual FA is functionally equivalent to other finite automaton. For each bye in the payload, a dual FA executes the following actions: (1) according to the next byte in the payload, the LFA and the EDFA take their transitions and determine their new active states, respectively; (2) additional NFA states in the LFA might be set to active if they are indicated by the transition taken by the EDFA; (3) the EDFA experiences a secondary transition if it is indicated by the transition taken by the LFA.

As an example, we show an execution of the dual FA in Figure 4 when consuming string "AZ". The initial state of the dual FA is (0,3)(), where (0,3) is the initial EDFA state[1], and () is the initial LFA state in which no state is active. When consuming character "A", the EDFA transits to state (0,1,3), and no second transition is taken because there are no active states in the LFA that can trigger a second transition of the EDFA. In this EDFA transition, state 4 in the LFA is set to active. Therefore, the new state of the dual FA is (0,1,3)(4). Then, the dual FA proceeds to consume the next character "Z". In the EDFA, state (0,1,3) transits to itself via the transition labeled "[^A-Q]", which does not set any states in the LFA to active. In the LFA, the active state 4 takes the transition labeled "[I-Z]{5}". The result is that (a) there is no new active state in the LFA since no transition to any LFA state is taken, and (b) the EDFA experiences a second transition from states (0,1,3) to (0,1,3,5). Thus, the new state of the dual FA is (0,1,3,5)().

To show that dual FA is equivalent to other finite automata, we compare this execution of the dual FA (Figure 4) to that of its corresponding DFA (Figure 3) as follows:

Dual FA: $(0,3)() \xrightarrow{A} (0,1,3)(4) \xrightarrow{Z} (0,1,3,5)()$

DFA: $(0,3) \xrightarrow{A} (0,1,3,4) \xrightarrow{Z} (0,1,3,5)$

Theorem 1: for any DFA, there is an equivalent dual FA.

Proof: if is sufficient to prove that: (1) each DFA state can be mapped into a dual FA state, and (2) each DFA transition can be mapped to a dual FA transition.

Firstly, each DFA state d_1 represents a set of concurrently active NFA states. Let N be the set of total NFA states, L be the set of NFA states implemented by the LFA, and $E = N - L$ is the set of NFA states implemented by the EDFA. The DFA state can be mapped into a dual FA state (e_1, l_1), where $e_1 = d_1 \cap E$, $l_1 = d_1 \cap L$, and $d_1 = e_1 \cup l_1$.

Secondly, we will prove that, for a byte c, if the DFA transits from states d_1 to d_2, d_1 maps into the dual FA state (e_1, l_1), and d_2 maps into the dual FA state (e_2, l_2); then, the dual FA transits from (e_1, l_1) to (e_2, l_2) on c. Let $T^n : N \times C \to N$ be the transition function of a NFA, and let $T^d : D \times C \to D$ be the transition function of a DFA, where $D \subset 2^N$ is the set of DFA states. We have $T^d(d, c) = \cup_{n \in d} T^n(n, c)$. For dual FA, we define:

$T^1 : E \times C \to E$ as $T^1(e, c) = E \cap (\cup_{n \in e} T^n(n, c))$,
$T^2 : E \times C \to L$ as $T^2(e, c) = L \cap (\cup_{n \in e} T^n(n, c))$,
$T^3 : L \times C \to E$ as $T^3(l, c) = E \cap (\cup_{n \in l} T^n(n, c))$,
$T^4 : L \times C \to L$ as $T^4(l, c) = L \cap (\cup_{n \in l} T^n(n, c))$.

T^1 to T^4 are implemented in dual FA as follows: (T^1, T^2) is given directly by the entries in the main EDFA transition table, while T^3 and T^4 are given by the LFA. In our implementation, for example, $T^3(l, c) = l\&extern[c]$ (see Section 3.4), and $T^4(l, c) = (l\&self[c]) | ((l\&next[c]) \gg 1)$. $e_2 = T^1(e_1, c) \cup T^3(l_1, c)$ and the union is implemented by the secondary transition, where the result of $T^3(l_1, c)$ is mapped to the virtual input byte. $l_2 = T^2(e_1, c) \cup T^4(l_1, c)$ and the union is implement by bit-wise or.

Finally, from the definition of T^1 to T^4, we have $e_2 \cup l_2 = T^1(e_1, c) \cup T^2(e_1, c) \cup T^3(l_1, c) \cup T^4(l_1, c) = \cup_{n \in e_1 \cup l_1} T^n(n, c) = \cup_{n \in d_1} T^n(n, c) = d_2$.

□

3.4 The Dual FA Transition Tables

Each EDFA transition needs to be encoded by two bit-arrays. The first bit-array indicates the index of the next EDFA state. The second bit-array (with k bits) indicates the states in the LFA (of size k) that are set to active by this EDFA transition, where k is the LFA size. For example, as shown in Table 2, given that state (0,1,3) in the EDFA table is indexed by 0 and that, in the LFA, the NFA states 2, 4 and 5 are given indexes 0, 1 and 2 in the bit-array, respectively, the EDFA transition table item "(0,1,3){4}" is encoded in binary as "0,010", where "010" is a bit-array which shows that the transition sets the NFA state 4, whose index is 1, to active.

Recall that each LFA state can only have three types of transitions, and the first two types are implemented with a number $|C|$ of k-bit $self$ masks and k-bit $next$ bit-masks, where $|C|$ is the size of the alphabet. Additionally, we use a number $|C|$ of k-bit $extern$ masks to implement a third type of transitions. According to Definition 2, each LFA state can only transit to at most one *external* NFA state (non-LFA state). The i^{th} bit in $extern[c]$ is 1 only if the i^{th} LFA state transits to an external NFA state on c.

Let l be the bit-array representing the old LFA states, and let c be the next byte in the payload. It is not difficult to prove that each $l\&extern[c]$ uniquely represents the set of external NFA states that are set to active by the LFA after consuming c. Consequently, the alphabet of the secondary EDFA transition table contains all bit-arrays for the possible results of $l\&extern[c]$. We simply use a non-colliding hashing ($hash(e) = e\%D$, modulo a constant D) to compress this alphabet size. The transition tables in dual FA can be further compressed by different table encoding techniques [22].

1. In implementation, a EDFA state is encoded as an integer state index. In this example, we use (0,3) to illustrate that the EDFA state corresponds to the configuration where only the NFA states 0 and 3 implemented by the EDFA are concurrently active.

Algorithm 1 EDFA construction

```
1: D ← the set of DFA states
2: L ← the set of LFA states
3: for each (d in S_D)
4:     e_1 ← d − L
5:     l_1 ← d ∩ L
6:     for each (c in CHAR_SET)
7:         e_2 ← active_set(e_1, c) −L
8:         tx_1[e_1][c] ← e_2
9:         e_+ ← active_set(l_1, c) −L
10:        e_3 ← e_2 ∪ e_+
11:        if (e_3 ≠ e_2) tx_2[e_2][hash(e_+)] ← e_3
```

Algorithm 2 DualFA simulator

```
1: (e_2, l_+) ← tx_1[e_1][c]
2: if (l_1 = 0)
3:     e_3 ← e_2
4:     l_3 ← l_+
5: else
6:     e_+ ← l_1 & extern[c]
7:     if (e_+ = 0) e_3 ← e_2
8:     else e_3 ← tx_2[e_2][hash(e_+)] // secondary transition
9:     l_3 ← l_+| (l_1 & self[c]) | ((l_1 & next[c]) ≫ 1)
```

In the execution of a dual FA, there is exactly one active EDFA state anytime: for each byte c consumed, the EDFA will experience a normal transition first and probably a secondary transition. If there is a secondary transition, it is determined by the virtual input byte calculated by $(l\&extern[c])\%D$, where l is the bit-array representing the currently active LFA states.

4 Implementation Details

4.1 Linear Finite Automaton

Number of LFA states: there is a trade-off between the per-flow dual FA state size (the length of the EDFA state index plus the length of the LFA active-set bit-array) and the storage of the EDFA transition tables, which will be examined in Section 5.

LFA states selection: since the size of a DFA cannot be determined before enumerating all of the states in the DFA, selecting the optimal k NFA states into the LFA to minimize the storage of the EDFA (which implements the rest of the $N-k$ NFA states) is an NP-hard problem: it requires the enumeration of all possible combinations of the sets of k NFA states and the construction of the corresponding EDFA. Therefore, we need to use a greedy heuristic algorithm to select the k LFA states one by one.

First, we observe that an NFA state that is continuously active for a long sequence of input bytes (except those NFA states that are constantly active) has more chances to be active both when other states are active and inactive, and therefore, are likely to be independent of other NFA states. On the other hand, NFA states that seldomly keep active for even a few consecutive input bytes are less likely to be independent. The former type of NFA states include: (1) the states with self transitions on a large character set, e.g., state 4 in Figure 1; (2) states with incoming transitions on a large character set from the states in case 1, e.g. state 5 in Figure 1.

In our LFA selection heuristics, each NFA state is given a score which equals the sum of the size of the character set mentioned in cases 1 and 2. In each round, an eligible NFA state with the highest score is selected and added to the LFA. An NFA state is eligible if the added state and the LFA states, together, still form an LFA.

4.2 Extended Deterministic Finite Automaton

After the k LFA states are selected, we can build an EDFA to implement the rest of the NFA states. We divide the EDFA states into two categories: (1) expandable states and (2) non-expandable states. An expandable state is a state that has one or more secondary transitions to other states. For example, in Figure 4, state (0,1,3) is an expandable state, and other states are non-expandable states. When consuming a byte, the EDFA experiences a secondary transition only when all of the following conditions hold: (a) the EDFA transits to an expandable state, (b) the set of active LFA states are not empty before consuming the byte, and (c) one of the LFA transitions indicates a secondary EDFA transition.

EDFA construction: the EDFA construction algorithm is listed in Algorithm 1. In this algorithm, we think of each DFA state d, each EDFA states e_1, e_2 and e_3, and each LFA configuration $l_1 \subseteq L$, as a set of active NFA states. In the algorithm, e_1 is an EDFA state, which is a subset of a particular DFA state d; l_1 is a LFA configuration concurrent to e_1 (i.e., $e_1 \cup l_1 = d$); the function *active_set(s, c)* returns the active set transit from active set s on byte c; $e_2 = active\_set(e_1, c) - L$ is an EDFA state (an active set) transit from e_1 on c; $e_+ = active\_set(L_1, c) - L$ is the active set transit from l_1 on c.

Next, we complete the secondary EDFA transition, with (1) e_2 being set active by e_1, (2) e_+ being set active by the LFA configuration l_1, and (3) $e_3 = e_2 \cup e_+$ being the final EDFA state. The secondary transition is from e_2 to e_3 and e_+ is the character in the transition. To reduce the alphabet, we use a function $hash(e_+) = e_+\%D$. In this construction, e_2 is an expandable EDFA state (that has secondary transitions) if it is not equal to e_3.

4.3 Dual Finite Automaton

The dual FA simulator is listed in Algorithm 2. In this algorithm, e_1, l_1 is the current EDFA state and LFA configuration, respectively. e_3, l_3 is the new EDFA state and LFA configuration after consuming character c, respectively. l_+ represents those LFA states that are set to active by the EDFA transition when consuming character c. $e+$ represents those NFA states implemented by the EDFA that are set to active by the EDFA when consuming character c.

Algorithm 3 Hardware dual FA implementation

1: $l_2 \leftarrow (l_1$ & self[c]) | ((l_1 & next[c]) $\gg 1$)
   $e_+ \leftarrow l_1$ & extern[c]
   $(e_3, l_+) \leftarrow$ tx_1[e_1][c]
2: $l_3 \leftarrow l_+ | l_2$
3: **if** ($e_+ > 0$) $e_3 \leftarrow$ tx_2[e_3][$hash(e_+)$]

4.4 A Discussion on Hardware Implementations

Although dual FAs have small storage sizes, they are still too large for pure hardware implementations. We consider a hardware architecture consisting of a small processing engine, e.g., implemented on an FPGA, coupled with a memory bank. The simple logic needed to implement in a dual FA reduces the need for LUTs compared to an NFA, which can lead to higher operating frequencies.

In the previous Algorithm 2, we have an additional if-else statement in order to save instructions whenever possible. If we implement the same logic in hardware, the algorithm can be simplified as listed in Algorithm 3. Also, the parallelism available in hardware, such as an FPGA, allows the processing of the three tasks in step 1 of Algorithm 3 to be completed in one memory cycle. Also, step 2 can be postponed to the beginning of the next memory cycle.

Step 3 of Algorithm 3 is invoked only when a secondary transition is needed. There can be two implementation options for this step: (1) implement secondary transitions in hardware, or (2) put the secondary transition table in the memory bank. As we will see Table 6 (b), the number of secondary transitions can be very small, which makes hardware implementation possible. On the other hand, if the secondary transition table is put in the memory, there should be a signal to delay the input of the next bytes, until the access to the secondary transition table completes.

5 Evaluation

To evaluate the proposed algorithm, we endeavored the following efforts: first, we developed several compilers, which read files of rule-sets and create the corresponding inspection programs and the transition tables for DFA, MDFA [8], H-FA [18] and dual FA. Second, we extracted rule-sets from the Snort [1], [2] 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.

5.1 Evaluation Method

We compare dual FA with DFA and MDFA [8]. 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 each DFA in an MDFA with a dual FA), we compare our algorithm with this widely adopted algorithm to show the efficiency of our method in terms of storage and memory bandwidth. We compare dual FAs whose LFAs are implemented by one computer word of 32 bits, as well as two words and three words with MDFAs with two, three and four paralleled DFAs, respectively. We called the compared FAs *dual FA 1*, *dual FA 2*, *dual FA 3*, *MDFA 2*, *MDFA 3*, *MDFA 4*, respectively. Since we assume that each computer word is 32 bits, dual FA 1 and MDFA 2 have the same per-flow state size, along with dual FA 2 and MDFA 3, and dual FA 3 and MDFA 4. We summarize the algorithms in comparison in Table 3.

TABLE 3
Algorithms in comparison and their per-flow state sizes.

Algorithm	Per-flow state size (words of 32 bits)
DFA	1
dual FA 1	2
dual FA 2	3
dual FA 3	4
MDFA 2	2
MDFA 3	3
MDFA 4	4

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 [9], [17] and alphabet compression [17], [19], [20]. We will examine how well dual FA can be combined with these algorithms in our future work. We do not show the results of H-FA [18] 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 [21] because the XFA compiler, which employs complicated compiler optimization technologies, is not available.

We developed our compilers 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 constraint repetition, are supported. Our compilers output C++ files for NFAs, DFAs, H-FA and dual FAs, respectively. The construction of the dual FAs is as efficient as the construction of DFAs.

We extracted rule-sets from Snort [1], [2] rules released Dec. 2009. Rules in Snort have been classified into different categories. We adopt a subset of the rule-set in each categories, 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 of large character-sets. The information about the rule-sets we used is listed in Table 4. This table lists the information about the sizes of the NFAs and DFAs constructed from each rule-set.

Synthetic payload streams are generated to be inspected by different FAs to measure their average number of main memory accesses during the inspection. Each

TABLE 4
Information about the rule-sets.

	Rules	NFA states	NFA states with self-transitions
#1	exploit-19	200	21
#2	smtp-36	456	69
#3	specific-threats-21	535	45
#4	spyware-put-93	2815	213
#5	web-client-35	941	121
#6	web-misc-28	513	58

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 create a DFA for the whole rule-set and travel the DFA in the following way: 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 [22], which prevents the DFA from traveling only low-depth states, as it does in normal traffic.

5.2 Performance Measurements

Since our focus is not in transition table encoding techniques [22], we measure the storage of the compared FAs in terms of (1) the number of states, (2) the number of transitions and (3) the storage size of the transition tables storing direct indexes. These measurements are important since, ideally, the number of states determines the number of bits required to encode a state index, and the number of transitions is the minimum number of entries in a transition table for any table encoding technique. For example, a DFA requires a minimum of $t \log_2 s$ bits, given the number of states s and the number of transitions t.

According to whether they can have secondary transitions, the EDFA states can be divided into two categories: *expandable states* and *non-expandable states*. According to whether some of their transitions can set LFA states to active, they can be divided into another two categories: *bit-setting states* and *non-bit-setting states*. In some cases, especially for dual FA 1s, the number of states belonging to the first categories can be very small in number and, for the rest of the states, transitions can be encoded by single state indexes. If we divide the EDFA states into four subcategories and encode them separately, a dual FA, whose number of states/transitions are equal to a DFA, can have a comparable storage requirement to a DFA.

The inspection speed of a dual FA, as well as other finite automata, depends on the hardware on which it is implemented. Even on general-purpose processors or ASIC hardware, it differs in the amounts of cache or on-chip memory. We measure the inspection speed primarily on memory bandwidth, i.e. the number of main memory accesses per KB of payload. This is because the execution of an instruction is usually much faster than fetching a piece of data from the external memory. Furthermore, with multi-core CPUs, the speed of main memory will further lag behind that of computation. We also generate inspection programs on personal computers with large cache memory and measure the inspection speed in terms of the time they take to inspect 64MB of payload.

TABLE 5
Storage requirement, number of states, and number of transitions of MDFA and dual FA with respect to DFA.

(a) Number of DFA states and the percentage in dual FA.

Rules	DFA	dual FA 1	dual FA 2	dual FA 3
#1	343k	0.04%	0.03%	0.02%
#2	141k	0.31%	0.29%	0.2%
#3	53k	1.02%	0.95%	0.88%
#4	269k	1.01%	0.99%	0.98%
#5	106k	2.33%	2.19%	1.9%
#6	453k	0.07%	0.06%	0.06%

(b) Number of DFA states and the percentage in MDFA.

Rules	DFA	MDFA 2	MDFA 3	MDFA 4
#1	343k	2.68%	0.54%	0.34%
#2	141k	6.51%	0.98%	0.62%
#3	53k	7.45%	2.38%	2.27%
#4	269k	18.06%	5.94%	5.94%
#5	106k	14.96%	14.81%	1.7%
#6	453k	3.49%	3.02%	0.39%

(c) Number of transitions in DFA and the percentage in dual FA.

Rules	DFA	dual FA 1	dual FA 2	dual FA 3
#1	7m	0.03%	0.03%	0.02%
#2	2m	0.23%	0.22%	0.14%
#3	702k	0.72%	0.66%	0.61%
#4	7m	0.94%	0.92%	0.91%
#5	2m	2.08%	1.96%	1.69%
#6	8m	0.05%	0.05%	0.04%

(d) Number of transitions and the percentage in MDFA.

Rules	DFA	MDFA 2	MDFA 3	MDFA 4
#1	7m	1.83%	0.24%	0.14%
#2	2m	6.24%	0.54%	0.24%
#3	702k	6.17%	1.1%	1.01%
#4	7m	7.9%	2.26%	2.12%
#5	2m	7.66%	7.43%	0.6%
#6	8m	2.52%	2.19%	0.16%

(e) Storage size of DFA and the percentage in dual FA.

Rules	DFA	dual FA 1	dual FA 2	dual FA 3
#1	343m	0.09%	0.11%	0.1%
#2	141m	0.63%	0.88%	0.8%
#3	53m	2.06%	2.86%	3.53%
#4	269m	2.05%	3.06%	4.13%
#5	106m	4.69%	6.76%	7.79%
#6	453m	0.15%	0.2%	0.24%

(f) Storage size of DFA and the percentage in MDFA.

Rules	DFA	MDFA 2	MDFA 3	MDFA 4
#1	343m	2.68%	0.54%	0.34%
#2	141m	6.51%	0.98%	0.62%
#3	53m	7.45%	2.38%	2.27%
#4	269m	18.06%	5.94%	5.94%
#5	106m	14.96%	14.81%	1.7%
#6	453m	3.49%	3.02%	0.39%

5.3 Results on Storage

The number of states and transitions are shown in Tables 5(a) to 5(d). As shown in Table 5(a), the numbers

TABLE 6
The 2^{nd} states, transitions, and bit-setting states in the EDFA of dual FA 1.

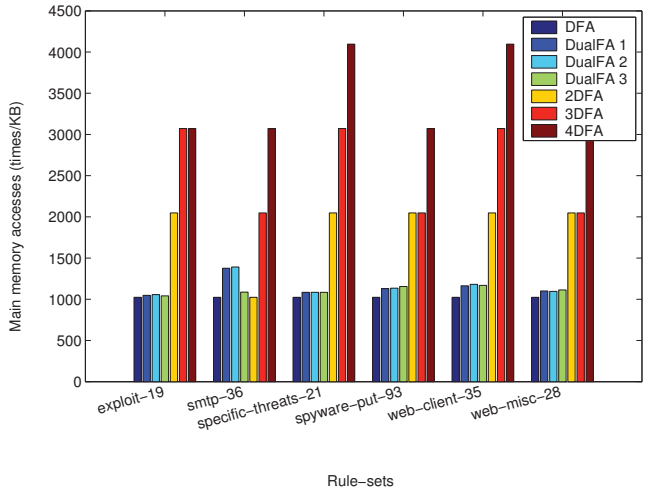
(a)

Rules	EDFA states	2^{nd} states	bit-setting states
#1	161	1.8%	100%
#2	452	23%	2.4%
#3	562	59%	5.6%
#4	2801	12%	5.8%
#5	2544	10%	1.1%
#6	358	7.8%	100%

(b)

Rules	primary EDFA transitions	2^{nd} transitions
#1	2575	0.2%
#2	5187	2.2%
#3	5193	1.8%
#4	72k	0.5%
#5	51k	0.6%
#6	5081	0.8%

Fig. 6. # of memory accesses per 1KB of payload stream.



of states in dual FA 1 ranges from 0.04% to 1 percent of those of DFA. The numbers of states of dual FA 2 and dual FA 3 are even smaller, but no further significant reduction is found as the size of LFA increases. The numbers of states in dual FAs are up to four orders of magnitude smaller than those of DFA. Compared with MDFA in Table 5(b), under the same per-flow state sizes, the dual FA are up to (e.g. in the "exploit-19" rule-set) two orders of magnitude smaller in the numbers of states. The dual FA also have smaller numbers of states than MDFA for all rule-sets, except for the "web-client-35" rule-set.

For the numbers of transitions, as shown in Table 5(c), dual FA 1 is up to four orders of magnitude smaller than DFA and two orders of magnitude smaller than MDFA 2 in Table 5(d). The number of transitions also decreases as the number of computer words used by the LFA increases: dual FA 3 is up to four orders of magnitude smaller than DFA and up to one order of magnitude smaller than MDFA 4.

An important observation regarding the number of states and transitions in dual FA is that: they decrease significantly using the first computer word for the LFA, but less significantly when subsequent words are used. This shows that our LFA states selection algorithm is able to identify the states that contribute most to the state explosion problem. A method to construct dual FA in practice is to construct dual FA with an increasing number of words for their LFA until the number of states/transitions decreases to a desired value or the number of words increases to a certain limit.

The storage size of the transition tables storing direct indexes are compared in Table 5(e) and 5(f). Contrary to number of states and transitions, the storage size of dual FA increases as the number of computer words used by the LFA increases. This is because, for a dual FA whose LFA uses K words, each entry in its transition table contains $K + 1$ words, and the reduction in the number of states is not as fast as the increases in K. However, when compared with MDFA, dual FA 1 is still up to two orders of magnitude smaller than MDFA 2, and dual FA 2 is smaller than MDFA 3 in five out of six rule-sets. From these results, we can see that the compression of dual FA is more efficient for the rule-sets where the number of NFA states with self-transitions (which are likely problematic NFA states) is small, and the state explosion is significant, such as rule-sets #1 and #6 (Table 4).

In Table 6, we show the number of states that have secondary transitions (2^{nd} states), the number of states that set LFA states to active (bit-setting states) and the number of secondary transitions in dual FA 1. The results show that the number of 2^{nd} states and bit-setting states are small: when the the number of EDFA states is large, e.g., in rule-sets "spyware-put-93" and "web-client-35", the percentage of 2^{nd} states and bit-setting states are small. The number of 2^{nd} transitions ranges from 0.2% to 2.2% of those of the primary EDFA transitions, which are also very small. Therefore, the storage size of a dual FA is dominated by its EDFA table.

5.4 Results on Memory Bandwidth and Speed

The number of main memory accesses for DFA, MDFA and dual FA during their inspection are shown in Figure 6. The results show that the number of accesses of MDFA increases proportionally to the number of extra DFAs. On the other hand, dual FAs have an average of 1.02 accesses per byte in the payload. This is because a secondary transition occurs only when the following conditions are satisfied simultaneously: (1) the EDFA transits to an expandable state, which accounts for a small percentage, e.g., 10%, of the EDFA states, as shown in Table 6; (2) some LFA states are active before consuming the next byte; (3) some active LFA states set the external state (the NFA states implemented by the EDFA) to active when consuming the next byte in the payload. The results suggest that, by satisfying all these

Fig. 7. Inspection speed in ms per 64MB of payload stream in PCs with memory hierarchy using C++.

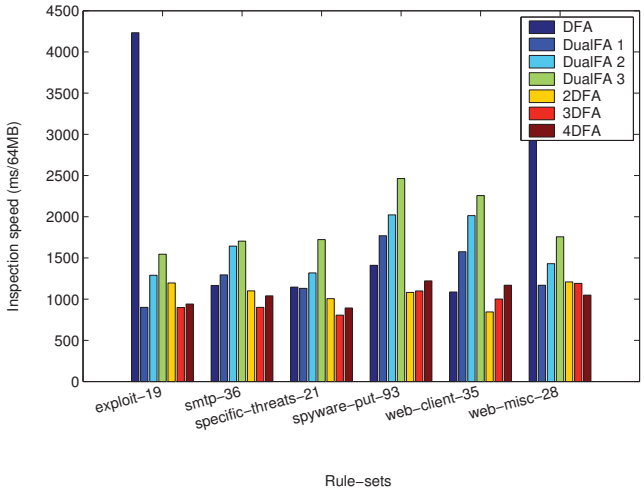
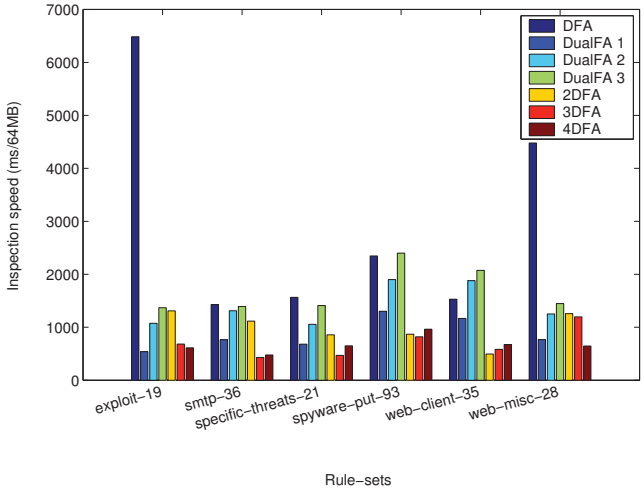


Fig. 8. Inspection speed in ms per 64MB of payload stream in PCs with memory hierarchy using Java.



three conditions, the chance of a secondary transition drops to 2%, even under the synthetic even payloads.

We also generate inspection programs in C++ and Java, respectively, on personal computers in order to show the computational efficiency of dual FA. For two of the rule-sets, the inspection speed of DFA is very slow because the DFA transition tables are particularly large in these cases, which cannot be effectively handled by the memory hierarchy (i.e., the 8MB L3 cache in our PC). Three observations can be made from the results in Figures 7 and 8 about dual FA: (1) dual FA 1 is more efficient than MDFA 2 in the challenging cases where DFAs are extraordinarily large and slow (i.e., rule-sets #1 and #6), while dual FAs have much smaller transition tables than MDFAs as can be seen in Tables 5(a) to 5(f); (2) dual FA 2 & 3, which require more computer instructions, are not suitable for sequential processors; (3) when memory hierarchy comes into play, as in other cases, MDFA is the fastest even though it has the largest number of memory accesses. Dual FA becomes comparatively slow because it requires several instructions. On the other hand, when memory hierarchy takes no effect, as in the cases of DFA in rule-sets #1 and #6, memory access is a major bottleneck, and dual FA excels.

5.5 Summary and Discussion

The results of the evaluation show that LFA is very efficient in reducing the number of states and transitions. It can reduce up to four orders of magnitude compared to DFA and two orders of magnitude compared to MDFA. Also, dual FA is much faster in terms of number of main memory accesses: the number of memory accesses hardly increase in the dual FAs, while the number of memory accesses in a MDFA increases as the number of extra DFAs increases.

Comparing the results on compression (Tables 5(a) to 5(f)) and Figures 7 and 8 with the characteristics of the rule-sets (Table 4), we can see that dual FA performs the best when there is a small number of problematic NFA states, and they cause severe state explosion in DFA, e.g., in rule-sets #1 and #6. For software implementation, a small number of problematic NFA states means the number is not larger than the length of a computer word. In these cases, dual FA 1 can be two orders of magnitudes smaller than MDFA 2 and runs faster than MDFA 2 even when the latter is highly accelerated by memory hierarchy.

Finally, as a limitation of dual FA, the number of LFA states cannot be large. In personal computer implementation, a large number of LFA states results in significantly more computational overhead. Also, a large number of LFA states results in a larger per-flow state, a larger transition table entry size (memory bandwidth) and a larger transition table storage size.

6 Related Work

Prior work on regular expression matching at line rate can be categorized by their implementation platforms into FPGA-based implementations [23], [24], [25], [26], [27] and general-purpose processors and ASIC hardware implementations [8], [18], [17], [14], [16], [15], [9].

Existing DFA state compression techniques (e.g. MDFA [8], HFA [18], XFA [21]) and transition compression techniques (e.g. D^{2}FA [9], [17], CD^{2}FA [16]) can effectively reduce the memory storage and introduce additional memory accesses per byte. *Delayed Input Deterministic Finite Automata* (D^{2}FA) [9], [17] 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.

CompactDFA [10] and HEXA [28] compress the number of bits required to represent the states, and they are only applicable to exact string matching.

Hybrid-FA [18], [14] 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.

Alphabet compression [17], [19], [20] 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 through pure DFAs [14], [29]. The back-reference [12] is a previously matched substring that is to be matched again later. The bounded repetition, or counting constraint, is a pattern that repeats a specific number of times. It is widely seen in the Snort and other rule sets, and is also known to cause state explosion. Dual FA can be extended to support constraint repetitions using existing techniques, such as counters, in [18], [21]. We omit bounded repetitions in this work for simplicity.

One might think that a bounded repetition which causes state explosion may be expanded into a number of NFA states, and use an LFA state to implement each of these states. However, LFA might not be suitable for bounded repetitions because of the following reasons: (1) LFA cannot use too many machine words. The per-flow state size and the computational overhead of an LFA is linear to the number of machine words it used. Therefore, an excessively large LFA degrades the performance of its dual FA. (2) a limited number of LFA states may not include the large number of NFA states created by bounded repetitions, and state explosion might remain exist. The underlying reason is that a bounded repetition can simply result in an extremely large state space, which has to be implemented by larger external data structures.

7 Conclusion

In this paper, we proposed and evaluated a *dual finite automaton* (dual FA) designed to address the shortcomings in existing deep packet inspection algorithms. The dual FA consists of a *linear finite automaton* (LFA) that handles the states causing state explosion without increasing memory bandwidth, and an *extended deterministic finite automaton* (EDFA) that interacts with the LFA. The dual FA provides an effective solution to the conflict between storage and memory bandwidth, and it is simple to implement. Simulation results show that, compared with DFA and MDFA, dual FA has significantly smaller storage demand and its memory bandwidths are close to those of DFA. In the future, we will implement dual FA along with other existing table compression techniques and see how performance increases when they work together.

References

[1] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of 13th System Administration Conf.*, November 1999.
[2] Snort: http://www.Snort.org/.
[3] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, December 1999.
[4] 2007. Citrix Systems & Application Firewall. http://www.citrix.com.
[5] 2007. Cisco Systems. Cisco ASA 5505 Adaptive Security Appliance. http://www.cisco.com.
[6] M. Altinel and M.J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of VLDB Conference*, 2000.
[7] 2007. ClamAV: http://www.clamav.net/.
[8] 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.
[9] 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.
[10] A. Bremler-Barr, D. Hay, and Y. Koral. CompactDFA: Generic State Machine Compression for Scalable Pattern Matching. In *Proc. of IEEE INFOCOM*, 2010.
[11] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In *Proc. of IEEE INFOCOM*, 2004.
[12] K. Namjoshi and G. Narlikar. Robust and Fast Pattern Matching for Intrusion Detection. In *Proc. of IEEE INFOCOM*, 2010.
[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory*. Addison Wesley, 1979.
[14] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *Proc. of CoNEXT*, 2007.
[15] L. Tan and T. Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proc. of ISCA*, 2005.
[16] S. Kumar, J. Turner, and J. Williams. Advanced Algorithms for Fast and Scalable Deep Packet Inspection. In *Proc. of ANCS*, 2006.
[17] M. Becchi and P. Crowley. An Improved Algorithm to Accelerate Regular Expression Evaluation. In *Proc. of ANCS*, 2007.
[18] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. Curing Regular Expressions Matching Algorithms From Insomnia. In *Proc. of ANCS*, 2007.
[19] S. Kong, R. Smith, and C. Estan. Efficient Signature Matching With Multiple Alphabet Compression Tables. In *Proc. of Securecomm*, 2008.
[20] M. Becchi and P. Crowley. Efficient Regular Expression Evaluation: Theory to Practice. In *Proc. of ANCS*, 2008.
[21] 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.
[22] M. Becchi, M. Franklin, and P. Crowley. A Workload for Evaluating Deep Packet Inspection Architectures. In *Proc. of IISWC*, 2008.
[23] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. of FCCM*, 2001.
[24] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection With Reconfigurable Hardware. In *Proc. of FCCM*, 2002.
[25] C. R. Clark and D. E. Schimmel. Efficient Reconfigurable Logic Circuit for Matching Complex Network Intrusion Detection Patterns. In *Proc. of FLP*, 2003.
[26] B. Brodie, R. Cytron, and D. Taylor. A Scalable Architecture For High-Throughput Regular-Expression Pattern Matching. In *Proc. of ISCA*, 2006.
[27] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proc. of ANCS*, 2007.
[28] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. In *Proc. of IEEE INFOCOM*, 2009.
[29] M. Becchi and P. Crowley. Extending Finite Automata to Efficiently Match Perl-Compatible Regular Expressions. In *Proc. of CoNEXT*, 2008.

Cong Liu Cong Liu is an assistant professor at the Department of Computer Computer at Sun Yat-sen University. He received his Ph.D. degree in computer science from Florida Atlantic University. Before that, he received his M.S. degree in computer software and theory from Sun Yat-sen (Zhongshan) University, China, in 2005, and his B.S. degree in micro-electronics from South China University of Technology in 2002. He is currently working toward his PhD. degree in the department of computer science, Florida Atlantic University, under the supervision of Dr. Jie Wu. He has served as a reviewer for many conferences and journal papers. His main research interests include routing in Mobile Ad hoc Networks (MANETs) and Delay Tolerant Networks (DTNs), Deep Packet Inspection, and Transaction Processing.

Jie Wu Jie Wu is a chair and a professor in the Department of Computer and Information Sciences, Temple University. Prior to joining Temple University, he was a program director at National Science Foundation. His research interests include wireless networks and mobile computing, routing protocols, fault-tolerant computing, and interconnection networks. He has published more than 500 papers in various journals and conference proceedings. He serves in the editorial board of the IEEE Transactions on Computers, IEEE Transactions on Mobile Computing, and Journal of Parallel and Distributed Computing. Dr. Wu is program cochair for IEEE INFOCOM 2011. He was also general cochair for IEEE MASS 2006, IEEE IPDPS 2008, and DCOSS 2009. He has served as an IEEE Computer Society distinguished visitor and is the chairman of the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a Fellow of the IEEE.