Computer Networks 55 (2011) 1648–1661

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Deep packet pre-filtering and finite state encoding for adaptive intrusion detection system

Ning Weng [a,*], Luke Vespa [a], Benfano Soewito [b]

[a] Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, United States
[b] Department of Information Technology, Bakrie University, Jakarta, Indonesia

ARTICLE INFO

Article history:
Available online 19 December 2010

Keywords:
Adaptive systems
Intrusion detection
Security system

ABSTRACT

An intrusion detection system (IDS) is a promising technique for detecting and thwarting attacks on computer systems and networks. In the context of ever-changing threats, new attacks are constantly created, and new rules for identifying them are dramatically increasing. To adapt to these new rules, IDSs must be easily reconfigurable, they must keep up with line rates of network traffic, and they must have high detection accuracy. In this paper, we propose a high-performance memory-based IDS that can be easily reconfigured for new rules. Our IDS achieves high performance and memory efficiency by utilizing deep packet pre-filtering and novel finite state encoding. We present simulation and experimental results that show the novelty and feasibility of our system.

Published by Elsevier B.V.

1. Introduction

The Internet suffers from lack of security; therefore, it is of utmost importance to secure Internet operations. For this purpose, intrusion detection systems (IDSs) are valuable security tools. IDSs detect patterns of potentially hazardous code, and they subsequently allow the elimination of the threat. As computer networks become more complex, their speeds increase, and more and more attack attempts are made. As a result, intrusion detection tasks become more complex and more crucial.

From the early Denning's classic statistical analysis of host intrusion, various IDSs have been proposed to detect and thwart attacks in computer systems and networks. IDSs can be classified as *signature*-based detection and *anomaly*-based detection. Signature-based detection compares the packet stream against signatures of known attacks, but it cannot detect a new zero-day attack. Anomaly detection can identify a new attack based on deviation from "normal" operation, but defining the normal conditions is difficult. A poor definition could result in high false positive and false negative rates. This paper will focus on signature-based detection.

The key requirements of an IDS are worst-case guaranteed performance, scalability with increasing line-rates of network traffic and growing number of attack patterns, high detection accuracy, and easy reconfiguration for newly acquired attacks once their signatures are identified by a third party. The capabilities of current IDSs are widely accepted as inadequate, particularly in the context of growing threats and changing network environments. Therefore, IDSs are deployed mostly in end systems with a much slower data rate and limited protection capability. However, it is beneficial to deploy IDSs inside computer networks (e.g., routers) to provide better protection. This method is especially important in light of the expansion of networks to incorporate sensors, mobile wireless devices, and other end-systems that have resource constraints and cannot execute complex security functions.

In this paper, we propose a high-performance and memory-efficient IDS with the ability to adapt to newly found attacks once their signatures are identified. Because of its simple memory-based architecture, our system can be easily reconfigured for new attack patterns. However,

* Corresponding author.
E-mail addresses: nweng@siu.edu (N. Weng), lvespa@siu.edu (L. Vespa), benfano.soewito@bakrie.ac.id (B. Soewito).

1389-1286/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.comnet.2010.12.007

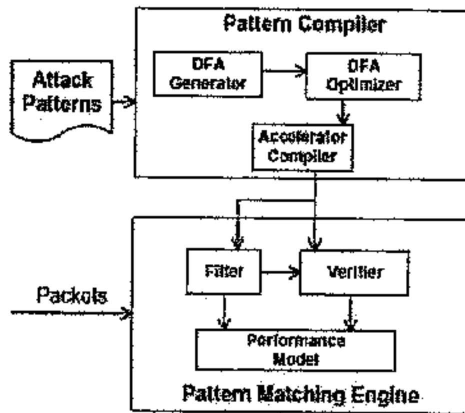


Fig. 1. Adaptive pattern matching system architecture.

this paper does not address the generation of new attack signatures. There is an ongoing effort to automate the generation of new attack signatures (e.g., Autograph [1], EarlyBird [2], Hancock [3] and Elcano [4]). Regardless of whether new signatures are acquired from slow-paced manual generation or an automated approach, our system can still easily adapt to these new signatures through excellent programmability. Fig. 1 illustrates the internal structure of our system's architecture. The pattern matching engine compares the incoming packet with known attack signatures. The main components are the following

- **Pattern Matching Engine.** The basic function of the pattern matching engine is to compare the incoming packet with known attacks. The key requirements are worst-case guaranteed line rate performance and ease-of-update. Our two-stage pattern matching engine (Fig. 2) consists of a deep packet pre-filter (Fig. 3) and a memory-based verifier (Fig. 4). The pre-filter can quickly filter out benign packets with zero false negatives. The verifier is based on an extremely resource-efficient and high-performance memory-based DFA. The detailed pattern matching engine is presented in Section 3.
- **Pattern Compiler.** The goal of the pattern compiler is to configure the pattern matching engine using minimum memory with the highest possible performance. The basic roles of the pattern compiler are DFA optimization, state encoding and pre-filter representative selection. To optimize the performance of the pattern matching engine, our pattern compiler is based on a group of novel ideas: state collapsing, DFA splitting and character-aware state coding. They ideas are presented in detail in Section 4.

Our system, as shown in Fig. 1, utilizes various novel DFA optimizations and a packet pre-filter that makes intrusion detection operations more efficient. The architecture of each component and methodology presents significant differences from existing work. These differences will be discussed in detail in Section 2:

1. We introduce a deep packet pre-filter, which leverages that the majority of packets are benign to improve performance. This pre-filter uses a small number of shorter pattern representatives to cover the entire attack signature database. These pattern representatives will have a zero false negative rate and will significantly reduce the workload of the verifier.
2. Our verifier engine uses a novel encoding technique called "character-aware". This unique character-aware encoding requires one entry in memory for each DFA state and only one memory lookup to identify the next state. This approach differs from other hardware-aided approaches, such as TCAM [5], parallel bloom filters [6] and our previous customized decoder-based approach [7,8]. These approaches require parallel query operations performed by hardware. Our character-aware coding requires only one memory lookup to determine the next state, and therefore, no specific hardware is required.
3. We introduce the split-block DFA (SDFA) optimization technique to reduce state dependencies and generate efficient character-aware encoding. SDFA achieves this by partitioning the DFA into multiple blocks, thus reducing DFA transitions. Our approach does not require hardware or hashing, and it achieves deterministic performance.
4. We apply a DFA optimization called "state collapsing", which combines many sequential states in a linear path into a single state. State collapsing improves memory efficiency by reducing the number of states, and it enhances performance by processing more than one character per matching transition.

The main contribution of this work is to present and implement a novel intrusion detection architecture to

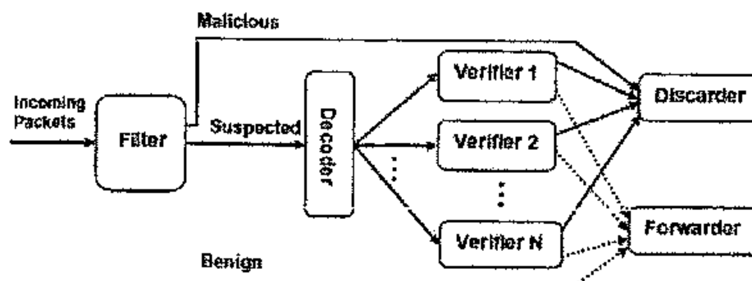


Fig. 2. Two stage pattern matching engine architecture.

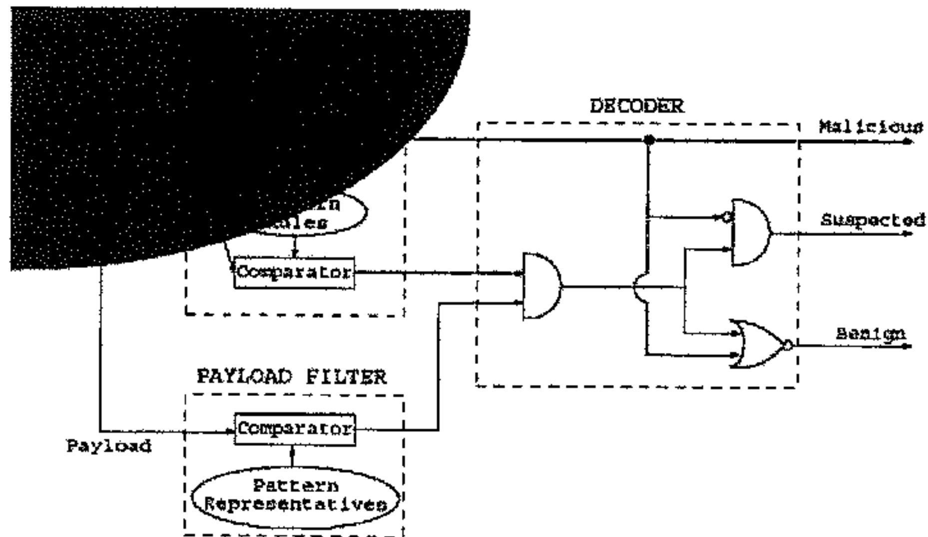


Fig. 3. Deep packet pre-filter.

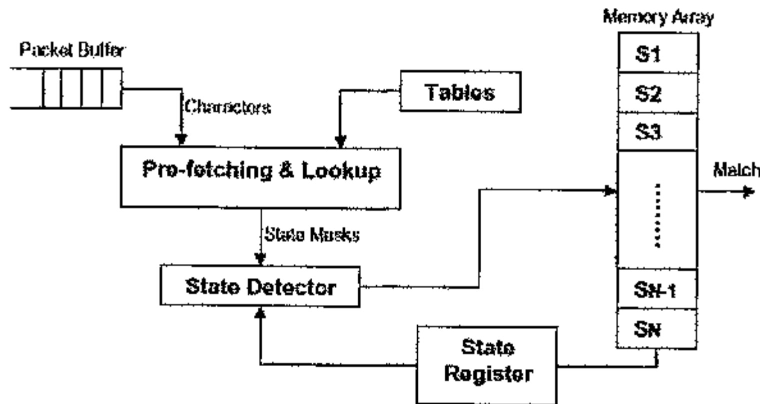


Fig. 4. Memory-based verifier architecture.

demonstrate its feasibility. The key features of our system are the following:

- An exceptionally memory-efficient pattern matching architecture that can reduce memory requirements (e.g., 80 times for 500 patterns) (resource scalability).
- The ability to match patterns at high speeds independently of the number and size of the patterns (worst-case performance).
- Easy updating of the memory-based engine with new known attack patterns (easy-to-update).

To show the novelty and feasibility of our system, simulation and experimental results are presented in Section 5. In particular, we present the following results:

- performance speedup due to the pre-filter of the pattern matching engine.
- memory efficiency and throughput of the verifier of the pattern matching engine.

The remainder of this paper is organized as follows. We discuss related work in Section 2. Section 3 introduces the architecture of our pattern matching engine. The pattern compiler, which reconfigures the pattern matching engine, is discussed in Section 4. The experimental and simulation results are presented and discussed in Sections 5 and 6 concludes the paper.

2. Related work

Pattern matching plays an increasingly important role for DNA sequence matching in bioengineering [10,11] and payload scanning in network applications (network intrusion detection [12–14], virus scanner and spam filters). Although a substantial amount of work has been performed in the area of pattern matching, most of the existing work does not meet the challenges of intrusion detection system because such systems must scan thousands of patterns at a rate of Gb/s.

Furthermore, new attacks are being created all the time. Accordingly, new rules for detecting these attacks are dramatically increasing in the Snort database [15], and this trend is expected to continue. Thus, it is important to design a pattern matching engine that is fully reconfigurable and worst-case performance guaranteed. One goal of this paper is to pursue an adaptive pattern matching engine that is not only performance scalable and memory efficient but also fully reconfigurable. With this in mind, we review work that addresses the partial requirements of an adaptive intrusion detection system.

2.1. Intrusion detection systems

Intrusion detection systems detect network attacks using either signature-based methods or anomaly-based detection. Signature-based systems search for a set of rules or signatures in incoming packets. These rules consist of specific header information and payload strings. Signature-based IDSs require a pattern matching engine to quickly identify payload patterns. Anomaly-based systems look for aberrant behavior using methods such as rate-based and statistical analyses. Anomaly detection can potentially be used to detect zero-day attacks, but it tends to have a high false-positive rate. Our work falls into the signature-based category, and we will discuss signature-based intrusion detection, pattern matching performance and memory in Sections 2.2 and 2.3.

There are several commercially available intrusion detection systems [16–21]. *Snort* [16] is a popular open source IDS. It is rule-based, includes both header and payload criteria and uses DFA-based deep packet inspection for payload rules. The patterns for deep packet inspection are divided into subsets based on their relation to the header rules. Bro [17] is an open source IDS similar to Snort. Bro is rule-based and compatible with Snort rules.

The Cisco Intrusion Prevention System [18] is another signature-based commercially available IDS. CIPS connects to the Cisco Security Intelligence Operation to correlate locally found threats to knowledge of global threats. Symantec Endpoint Protection [21] is a signature/anomaly IDS that is meant for endpoint deployment. Other commercially available IDSs include Sax2 [19] and the NIKSUN NetDetector [20]. The NIKSUN NetDetector is both a signature and anomaly detection IDS; however, the threshold values in the anomaly detection system must be user defined.

Optimizations to rule-based processing have also been proposed. A hierarchical firewall rule processing method called OPTWALL [22] partitions a list of N rules into optimal K subsets. The goal is to optimize list rule processing to minimize the number of rules examined for each packet.

There is an extensive body of work on anomaly detection algorithms. These algorithms use different techniques and information from the packet stream. Holt-Winter uses a volume-based forecasting model [23]; Barford et al. [24] use frequency information to determine anomalies from a signal processing point of view; Gu et al. [25] have proposed a maximum-entropy-based anomaly detection algorithm; Lakhina et al. [26] use a subspace method to identify anomalies in byte counts, packet counts, and IP-flow counts; Siris and Papagalou [27] present a statistical anomaly detection algorithm that identifies SYN flooding attacks using adaptive threshold and cumulative sum. These anomaly detection algorithms are limited in what anomalies they can detect. Therefore, single anomaly detection has a high false-negative rate. To achieve better detection accuracy, Shanbhag and Wolf [28] implement multiple anomaly detection algorithms and monitor multiple traffic subsets in parallel on high-performance network processors [29].

Our work differs from other intrusion detection systems in several ways. Our pattern matching engine uses a two-stage model with a deep packet pre-filter and verifier. Our pre-filter performs deep packet inspection using pattern representatives that represent all attack signatures. Our pattern representatives have a zero false negative rate, and they significantly reduce the verifier's workload. Our verifier is fully optimized for memory and performance by our DFA state encoding methods. Our state encoding allows our pattern matching engines to store only one entry in memory for each DFA state, while requiring only one memory lookup to find the next state.

There are numerous concerns when deploying IDS in a network. These concerns include privacy, network topology, cost of accessing packet payloads and additional necessary protocols. Currently, common practice in large institutions and corporations is to rely on privacy policies to handle data privacy. Also, encrypted payloads may not allow deep inspection. As a standalone device added to a network, topology changes may be required. However, if the IDS functions as a component of the host appliance, there are no network topology changes required to deploy IDS. There is also an extra cost for the router associated with accessing and inspecting a packet payload. The complete TCP/IP stack might be required for a stateful IDS, but most likely it is not required for a stateless IDS. Efficient solutions that address these concerns will pave the way for deploying our scalable pattern matching engine.

2.2. Performance enhancement of pattern matching

The key bottleneck of signature-based IDSs is the pattern matching engine because it compares every byte of a packet with a set of thousands of possible attack patterns in line speed. Algorithms [30–32] based on the commodity processor approach do not scale well with increasing line-rates of network traffic. There are several methods for speeding up pattern matching, including hashing and the use of specialized hardware. Our approach does not require special hardware or hashing, and it achieves deterministic performance.

To deliver the required high performance, specific techniques such as hashing [33], bloom-filtering [6], and pre-filtering [34] have been employed to optimize the average case. However, this average-case optimization can be easily attacked by sending numerous rare-case packets or packets containing known attack patterns.

Hardware-based approaches such as the customized hardware accelerator [35] and the rule-based comparator [36] can deliver deterministic performance, but they require special hardware. Another approach is to exploit hardware parallelism to process multiple packets, multiple pattern segments [37], or bit-splits [38] simultaneously. Our approach differs from other hardware-aided ap-

proaches, such as TCAM [5], because no parallel operations or comparisons are required.

The multiple strides DFA [39,40] is another method for speeding up pattern matching by inspecting multiple packet characters simultaneously. VS-DFA [39] increases DFA stride by using a fingerprinting scheme to hash the incoming packet data. VS-DFA requires expensive hardware, such as TCAM. In [40], DFA stride is increased by allowing multiple transitions to be traversed simultaneously. This system also uses a specially designed hardware approach.

2.3. Memory efficient pattern matching

Memory-based DFAs are easily reconfigurable for new pattern updates. However, the huge memory requirement of memory-based DFAs is an issue. For example, Snort [15] Dec. 05 has 2,733 patterns that need 27,000 states to represent them, requiring about 13 MB, which is too large for on-chip memory.

There are several approaches to improving memory efficiency. Bitmap and compression [12] have been proposed to optimize the AC algorithm data structure to improve memory efficiency. The problems with bitmap compression are the requirement of 2 memory references per character in the worst case and 256 bits per bitmap. A potential problem with path compression is that failure pointers may point to the middle of other path compressed nodes. Lunteren [41] introduced a pattern matching engine called BFPM (Balance Randomized Tree FSM Pattern Matching). BFPM uses a prioritized rule list, which reduces the number of transitions in a DFA. The potential problem with this approach is that it sometime requires multiple memory accesses per transition, which degrades performance. We will compare the memory efficiency of these approaches with our own in Section 5.

Our previous work proposed a self-addressable state coding [7], where state S_i's code consists of a group of address tags that serve as pointers to all the next states of state S_i. To determine the next state, a special memory decoder is required to simultaneously check all address tags and incoming packet characters. Two custom decoders [8] were implemented to perform parallel lookup. This paper differs from our previous work [7,8] in several ways. First, we introduce a deep packet pre-filter, which leverages that the majority of packets are benign to improve performance. This pre-filter uses pattern representatives to represent the entire attack signature database with no false negative rate. Second, our verifier engine uses a new encoding technique called "character-aware", where only one address tag in any state code needs to be examined to find the next state. Thus, character-aware encoding does not require a customized decoder to perform parallel lookup. Third, we introduce the split-block DFA optimization technique to reduce state dependencies to generate efficient character-aware encoding.

3. Pattern matching engine architecture

A high-performance and memory-efficient two-stage pattern-matching engine is proposed in Fig. 2. A packet enters the pattern matching engine and is first processed by the high-speed deep packet pre-filter shown in Fig. 3. If necessary, the packet is further processed by the verifier shown in Fig. 4. The packet is then either discarded or forwarded. The pre-filter classifies the incoming packet into three categories: malicious, suspected or benign. The verifier is the second stage of our pattern matching engine and is used to verify whether the suspected packet is malicious. If the packet turns out to be benign, then it is a false positive.

3.1. Pre-filter architecture

The filter architecture for the proposed NIDS is shown in Fig. 3. The key components of this filter are the header checker and the payload filter. The header checker examines the header information of the incoming packet, and it compares the header fields against a set of known network attacks.

If no match is found, the packet is forwarded to the network. If the header information matches the rule set, then it is sent to the payload filter. The payload filter, or verifier, scans every byte of the payload to see if it contains a known attack pattern. In this process, the pre-filter only uses partial patterns, called pattern representatives, which speed up processing. Based on the result of the payload filter, the packet is either discarded or forwarded to the verifier for further investigation.

The pre-filter's high speed is achieved by only inspecting whether the payload contains segments of the attack patterns. A short segment of each pattern is used to represent each full pattern. By only comparing the incoming packets with segments of the Snort rules, the operations to be performed by the pre-filter are reduced, resulting in high-speed performance. For a given set of patterns, the representative selection, the number of representatives R and the representative length r all potentially affect the performance of the filter.

When selecting representatives for a set of patterns, we attempt to find single representatives that correspond to numerous patterns. For example, assuming the pattern is "*/bin*" and r is selected to be 2, the representative will be one of the following strings: */b*, *bi*, *in*. If another pattern in the pattern list is "into", then we can use the representative "in" to represent both "*/bin*" and "into". This approach allows two 4-byte patterns to be represented by one 2-byte pattern. If the pattern length is smaller than the selected r value, then the entire pattern is used as the representative.

The process for choosing pattern representatives is described in detail in Section 4.3. There are also many performance and memory tradeoffs that help decide what representative length, r, to use when selecting representatives. These tradeoffs guide us in selecting an optimal representative length. The tradeoffs are discussed in Sections 5.4, 5.5 and 5.6.

3.2. Verifier architecture

Our proposed memory-based verifier is based on the following observation. In a straightforward memory-based

DFA implementation, for a given state, 256 next-state pointers are stored in the memory. Many of these pointers are repeating data entries, which waste memory resources. If we can design a meta-pointer for each state that points to all the possible next states, we can store a single pointer for each state and, consequently, save a great portion of the memory. Because the meta-pointer indicates multiple states, the next state selection requires a special memory decoder [7,8]. In this paper, instead of relying on special decoders, our verifier uses character-aware encoding, which requires only one memory lookup, even without the aid of special hardware.

The verifier architecture is shown in Fig. 4. The architecture consists of a memory array, a state register, a state detector, and input pre-fetching and lookup. Similar to other memory-based DFAs, the memory array stores state codes (metapointers), and the state register stores the current state. The state detectors detect address tags contained in the code of the current state, and they identify the address of the next state. Assuming the incoming packet is buffered for processing, we can pre-fetch the next incoming character of a packet before the next state is ready.

4. Pattern compiler

The pattern compiler is used when new attack patterns are available and new pattern representatives and state codes need to be generated and programmed to the memory. The pattern compiler performs DFA generation, DFA optimization and pre-filter representative selection. The compiler then programs the engine by deriving state codes and operational lookup tables and writing them to memory. To optimize the performance of the pattern matching engine, our pattern compiler also uses a group of novel ideas during the engine programming process: state collapsing, DFA splitting, and character-aware state coding.

4.1. DFA generation

Deterministic finite automata (DFA) are able to match multiple patterns simultaneously in *worst-case* time linear to the size of a packet. Fig. 5(a) shows an example DFA used to match "SHE", "HERS" and "HIS". Starting at state *s*0, the state machine is traversed to state *s*1 or *s*2 depending on whether the input character is "H" or "S". When an end state is reached, a string is said to be matched. In the example in Fig. 5(a), if state *s*9 is reached, string "HERS" has been matched.

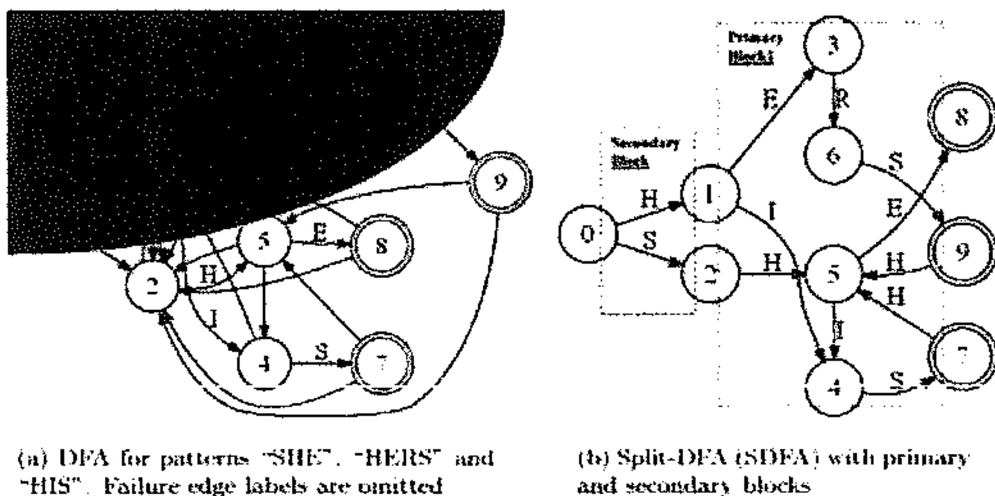
Each state in the DFA has pointers to other states. If an input character is the next character in a string that is currently being matched, it moves to the next state in that string; otherwise, it follows a failure pointer to the first state of another string that begins with that character or to the initial state of the machine if no other strings begin with that character. An example of this process can be seen in Fig. 5(a). If the current state of the machine is *s*3, the last input characters would have been "HE". If the next input character were to be "R", then the next state would be *s*6. If the next input character were not "R", but rather "S", then the next state would follow a failure pointer to state *s*2, which is the starting point for the string "SHE".

4.2. DFA optimization

We present two DFA optimizations: state collapsing, which reduces the number of states in a DFA, and split-DFA (SDFA), which reduces the number of transitions.

4.2.1. State collapsing

State collapsing finds commonly occurring sub-strings (referred to as "super-characters") and collapses the states corresponding to the occurrence of such sub-strings into a single state. A sub-string is qualified for a state collapsing operation only if its corresponding states have only one



(a) DFA for patterns "SHE", "HERS" and "HIS". Failure edge labels are omitted

(b) Split-DFA (SDFA) with primary and secondary blocks

Fig. 5. Example of DFA and Split DFA.

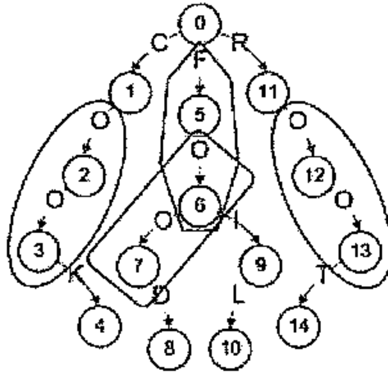


Fig. 6. DFA with circled super-characters.

fan-out branch, except the state corresponding to the last character. For example, Fig. 6 shows a DFA for patterns *COOK*, *FOOD*, *FOIL*, *ROOT*. The valid super-characters are marked by ovals in the figure, while the sub-string surrounded by a rectangle is not a valid super-character. Note that the proposed technique not only reduces memory size but also makes it possible for the DFA to process multiple characters within one clock cycle, which increases the system throughput.

One challenge of implementing state collapsing is how to follow a super-character transition. In contrast to a standard DFA implementation, here we need to recognize the selected super-characters. Therefore, instead of only inspecting one character at a time, this system needs pre-fetching, and it looks ahead up to N characters to check for occurrences of super-characters.

The complexity of the data structure will be affected by the length and number of super-characters. Therefore, it is crucial to identify the proper set of super-characters. The proposed heuristic (not necessarily the best) is as follows. First, locate all possible valid sub-strings within the pattern set up to length N. Second, calculate the occurrence of each sub-string and sort sub-strings in descending order by length/occurrence. Third, replace all states for each valid sub-string occurrence with one state, referred to as a "super-state".

The performance increase of state-collapsing will be determined by the occurrence of particular super-characters in packet traffic. The processing cost savings is calculated by Eq. (1), where (C_s) is the number of cycles saved by matching (n) super-characters. For each super-character matched, we save as many cycles as the length of super-character (S_j) minus 1. This speedup gives the system an added speed boost if numerous super-characters are present in the packet payload.

$$C_s = \sum_{j=1}^{n}(len(S_j) - 1). \quad (1)$$

4.2.2. Split-DFA (SDFA)

SDFA partitions the DFA into separate blocks, called primary and secondary blocks, to reduce DFA transitions. All transitions to first-level states other than those from state zero are removed from the DFA. The outgoing transitions from state zero are partitioned into the secondary block, as seen in Fig. 5(b). The remainder of the transitions are placed in the primary block.

SDFA operates by passing the incoming character to the primary and secondary blocks. If the primary block has a valid transition for a given character, this transition is taken, and the next character is processed. However, if the primary block fails for a given character, the transition from the secondary block is used, and operation continues with the next character.

4.3. Pre-filter representation selection

Given a pattern set P, the first step in selecting pattern representatives is to extract all possible r-byte substrings from the patterns, where r is the chosen length of the representatives. This step yields all of the possible pattern representatives for pattern set P. The next step is to find the minimum number of representatives from the set of all possible representatives. There will be minimum one and maximum $|P|$ representatives necessary to represent all patterns. For example, consider the following 4 patterns:

(1) */bin*
(2) *invalid*
(3) */booking*
(4) *uname*

If the r value is selected to be 2, the representative candidates are:

(1) */b, bi, in*
(2) *in, nv, va, al, li, id*
(3) */b, bo, oo, ok, ki, in, ng*
(4) *un, na, am, me*

From the above representatives, we select *in* to cover patterns *a*, *b*, and *c*. We select one of the representative candidates from pattern *d*: *un*, *na*, *am* or *me* as a representation of pattern *d*. This process leads to a solution that uses two representatives to cover all four patterns. In our work, a method based on a greedy algorithm is used to search for the optimal set of representatives.

4.4. Pattern matching engine programming

This section demonstrates the construction of the look-up tables used in the operation of the pattern matching engine. The two tables needed for DFA operations are the code table and the character/cluster table. The code table contains a special code for each state in the DFA, which is derived such that each state's code forecasts all of that state's possible next states. The state codes are "character aware" in that the next state can be easily isolated using the incoming character from a packet. The character/cluster table contains information about each character in the DFA alphabet.

Algorithm 1: Algorithm for searching mutually independent groups

```
    Input: State Trans. Table: T
    Result: Indep. Group Set R
1   Search_Indep_Group(T)
2   begin
3       R = ∅
4       G = StateGrouping(T);
5       𝒢=Create_Graph(G);
6       𝒢_c=Character_Aware(𝒢);
7       while 𝒢_c ≠ ∅ do
8           c = MaxClique (𝒢_c);
9           add c to set R;
10          remove c from 𝒢_c;
11      end
12  end
13  Create_Graph(G)
14  begin
15      while G ≠ ∅ do
16          remove a group g_i from G;
17          add a vertex v_i for group g_i on 𝒢;
18          for vertex Vv_j on 𝒢 do
19              if g_i ∩ g_j = ∅ then
20                  add an edge between v_i and v_j
21              end
22          end
23      end
24  end
25  Character_Aware(𝒢)
26  begin
27          for each char c_i in 𝒢_c do
28              for other chars c_j in 𝒢 do
29                  NC = non-complete sub-graphs
                    for nodes (c_i, c_j);
30                  remove all edges in NC from 𝒢_c;
31              end
32          end
33  end
```

4.4.1. State code construction

Each state S_i has a single code word. The code word for S_i consists of a series of address tags, where each address tag represents one next state of S_i. Each state S_i has an address tag that is to be included in the state code of all states that have S_i as a next state. We refer to all states that share a next state S_i as a group G_i. This group G_i should also store the character of the shared next state, which will be used when constructing state codes. We use this concept of group to derive the address tags for each state and then use those address tags to form the final state codes for each state.

The process starts by applying Algorithm 1, which finds all mutually independent groups. These are groups that do not share any of the same members. This qualification is important because the address tags of all members of a mutually independent group can be placed in the same location in any state code. The locations where address tags are placed are referred to as clusters. With more mutually independent groups, less clusters are required to form state codes. This condition results in shorter state codes and thus requires less memory. We apply binary encoding to generate address tags for the groups within a cluster. Once address tags are generated for each state, the code for each state can be generated by concatenating all the address tags that make up a state's possible next states.

Deriving state codes starts with locating all independent groups beginning with line 4 of Algorithm 1. For each state S_i, place all fan-in states of S_i into one group. After this procedure, N groups are generated, where N is the number of DFA states. In line 5, a group graph is generated following the procedure described by lines 13–24. An example group graph is shown in Fig. 7. In lines 25–33, a character-aware function removes all edges in the group graph between groups of different characters if all the groups of those characters do not form a complete subgraph. In line 8, a well-known maximum clique search algorithm is used to recursively search the maximum cliques of the graph. The vertices of the obtained cliques represent the mutually independent groups. The process of deriving state codes is further illustrated by the following example:

This example derives state codes for the DFA in Fig. 5(b). The first step in deriving the state codes is to group all the states in the DFA that have the same next state into one group. The result is one group for each state (i.e., G1 = the fan-in of S1). Also, the character needed to transition to a state is associated with that state's group (i.e., S1.character = 'H'). The resulting groups are: G1[S0][H], G2[S0][S], G3[S1][E], G4[S1, S5][I], G5[S2, S7, S9][H], G6[S3][R], G7[S4][S], G8[S5][E], G9[S6][S].

The second step is to combine these groups to form clusters. Groups with the same character are combined

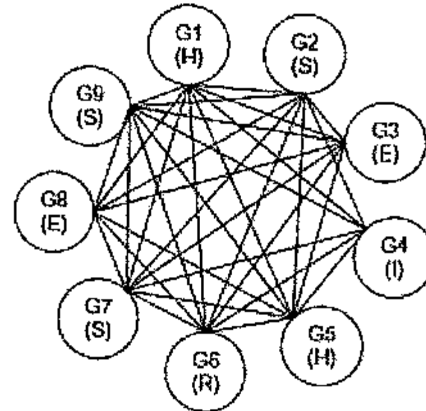


Fig. 7. Group graph.

Table 1
Clustering.

Cluster	Character	Group	Bit length
C1	H S E R	G1 G5 G2 G7 G9 G3 G8 G6	4
C2	I	G4	1

into a cluster. Next, the number of clusters is reduced by merging all clusters that do not have common states to form one cluster. The clustering result is given in Table 1. Characters 'H', 'S', 'E', and 'R' do not have common states, and they can be placed in one cluster.

The third step is encoding the groups. Group codes have two parts, the character tag and the state tag, which together form an address tag. The character tag determines whether the incoming character has a valid state transition. The state tag represents the state to which the group corresponds. The resulting group codes are shown in Table 2. For example, from Table 2, we see that groups G1 and G5 have the same character tag, 0 0, because both groups have character 'H'; however, their state tags are 0 1 and 1 0, respectively.

Finally, the state code for each state is obtained by concatenating the group codes for the groups each state is a member of. Table 3 shows the state codes. As an example, S1 is a member of G3 and G4; therefore, the group codes for G3 and G4 make up the code of S1. The group code of G3 is 1001 and the group code of G4 is 0 1, as seen in Table 2. The group code for G3 is placed before the group code for G4 because G3 lies in Cluster 1, whereas G4 is in Cluster 2.

4.4.2. Lookup table construction

This section shows how the lookup tables are formed. The tables include the character/cluster table and the code table.

$$S_{index} = Ch_{index} + S_{tag}. \tag{2}$$

Character/Cluster Table. The character/cluster table has four fields. The first field is the character tag. The second field is the cluster number that contains all the states associated with that character. The third field is an index number assigned to each character. The index starts at 0 for the first character and is incremented for each character by the number of states with the previous character. The fourth field is a failure index assigned to each character. If a valid transition is not produced during DFA operation, this failure index automatically becomes the next state index. Table 4 shows the final character/cluster table for this example.

Code table The code table consists of the state codes placed in the correct order. The index position for each state code in the code table can be calculated by Eq. (2) by adding the state-tag to the character index for any given state. Table 5 shows the final code table for this example.

4.4.3. Reconfiguration for new patterns

When new attack patterns need to be included in the pattern search, it is simple to update the memory-based engine to include them. The new patterns are simply added to the existing pattern lists, new pattern representatives are generated for the prefilter if necessary, and new DFA state codes are generated. These new code words then replace the old ones in the memory.

5. Results

Experimental results are discussed in this section, beginning with the verifier and moving on to the pre-filter. For the verifier, we first discuss the memory requirements, followed by the throughput and then the memory/performance tradeoffs of our encoding methods. For the pre-filter, we discuss false positive rates, the tradeoffs when choosing different pattern representative lengths and the speedup due to the pre-filter. We summarize the results in Section 5.7.

Table 2
Group coding.

Group	Char-tag	State-tag
G1	0 0	0 1
G5	0 0	1 0
G2	0 1	0 1
G7	0 1	1 0
G9	0 1	1 1
G3	1 0	0 1
G8	1 0	1 0
G6	1 1	0 1
G4	0	1

Table 3
State codes.

State	Group	Code
S1	G3 G4	1 0 0 1 0 1
S2	G5	0 0 1 0 0 0
S3	G6	1 1 0 1 0 0
S4	G7	0 1 1 0 0 0
S5	G8 G4	1 0 1 0 0 1
S6	G9	0 1 1 1 0 0
S7	G5	0 0 1 0 0 0
S8	G6	1 1 0 1 0 0
S9	G5	0 0 1 0 0 0

Table 4
Character/cluster table.

Character	Signature	Cluster	Index	Failure index
H	0 0	1	0	1
S	0 1	1	2	3
E	1 0	1	5	0
R	1 1	1	7	0
I	0	2	8	0

Table 5
Code table.

Index	State code	State
1	1 0 0 1 0 1	S1
2	1 0 1 0 0 1	S5
3	0 0 1 0 0 0	S2
4	0 0 1 0 0 0	S7
5	0 0 1 0 0 0	S9
6	1 1 0 1 0 0	S3
7	0 0 0 0 0 0	S8
8	0 1 1 1 0 0	S6
9	0 1 1 0 0 0	S4

Table 6
Memory efficiency compared with traditional DFA (TDFA). Our approach uses character-aware encoding, split-DFA (SDFA) and state collapsing (memory size in kByte).

Snort rules		Binary encoding TDFA	Character-aware encoding			Memory size per char (byte)
Number of patterns	Number of char.		TDFA	SDFA	SDFA & state collapsing	
5	98	21.25	0.69	0.10	0.21	2.16
20	334	94.38	5.16	0.77	0.51	1.55
50	661	195.25	11.31	2.50	1.15	1.77
100	1291	397.50	34.16	8.15	3.04	2.41
200	2129	600.38	60.39	16.03	5.85	2.82
300	4313	1258.56	218.21	42.73	14.26	3.38
400	6722	2116.19	268.66	72.63	23.79	3.62
500	7637	2304.11	298.33	81.65	27.04	3.63

5.1. Verifier memory

Table 6 shows that character-aware encoding of a traditional DFA reduces the memory requirement from binary encoding by more than 7 times, as seen in columns 3 and 4. Character-aware encoding achieves this memory decrease by reducing the number of required memory entries from $|E|$ to $|S|$, where E is the set of edges in a DFA and S is the set of states. Character-aware encoding of a split-DFA reduces the memory required for a binary encoded traditional DFA by more than 20 times, as seen in columns 3 and 5.

Character-aware encoding of a split-DFA with state collapsing reduces the memory required by over 80 times, as seen in columns 3 and 6. State collapsing, character-aware encoding and split-DFA are orthogonal techniques to reduce memory. State collapsing reduces the number of DFA states. Split-DFA reduces the length of a state.

Fig. 8 compares storage efficiency in terms of bytes per character with AC [30], Bitmap AC [12], Path compr. AC [12] and B-FSM [41]. Our result is taken from the seventh column of Table 6 for 500 patterns. Even though the B-FSM approach results in a similar memory per character to our approach, it might require multiple memory accesses to determine the next state, which potentially degrades system throughput. In contrast, our approach requires only two memory accesses per state.

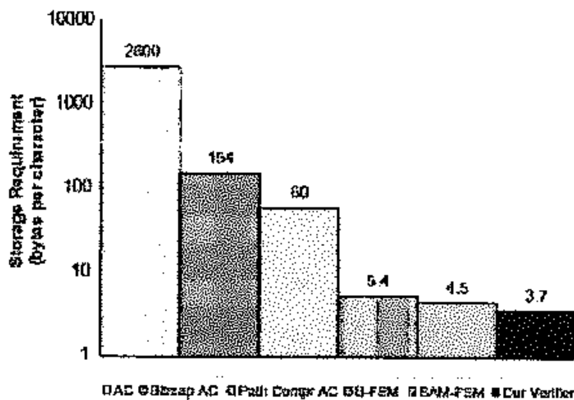
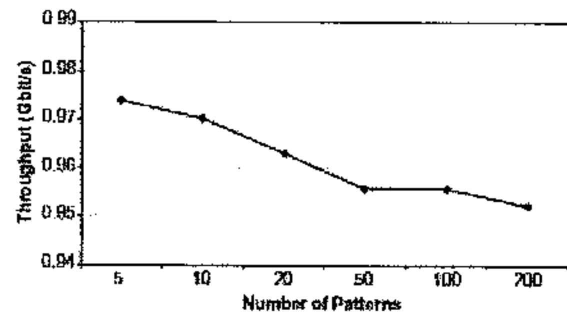


Fig. 8. Verifier memory requirement comparison with related work.

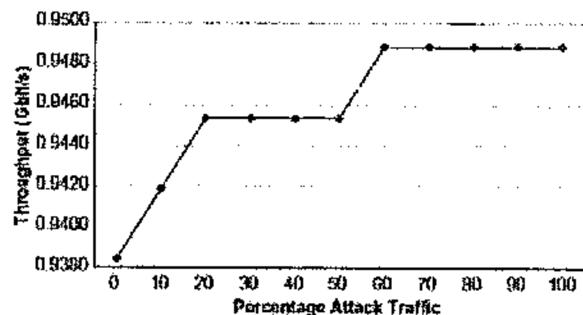
5.2. Verifier throughput

To test the performance of the verifier component of the pattern matching engine, the DETERlab Test-Bed [43] was used to create a real-time network scenario. Fig. 9(a) shows the throughput of the verifier as the number of patterns being searched increases. The throughput remains constant in the range of 0.952 Gb/s to 0.975 Gb/s, regardless of the number of patterns.

Fig. 9(b) shows the throughput of the verifier as the percentage of traffic containing attacks increases. The throughput remains constant in the range of 0.938 Gb/s to 0.949 Gb/s regardless of the percentage of attacks in the packet payloads. Fig. 9 shows that the performance of the verifier is deterministic regardless of the number of patterns or traffic mix.



(a) throughput with # of patterns



(b) throughput with attack traffic mix

Fig. 9. Verifier throughput evaluated in the DETER Testbed [43].

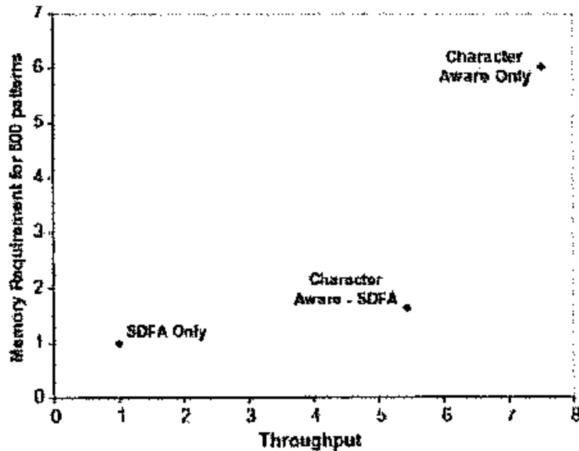


Fig. 10. Verifier memory and throughput tradeoffs. SDFA-only serves as a baseline, and 500 patterns are used.

5.3. Verifier memory and throughput tradeoffs

The encoding techniques provide a tradeoff in terms of memory and performance. Understanding this tradeoff will help us design a suitable verifier. Fig. 10 shows the three possible verifier designs. The "Character Aware only" approach achieves the highest throughput. This approach stores each character in a separate cluster, which increases the speed of lookup because all the possible next states for a given character are now in one cluster. However, the "Character Aware only" approach also increases the memory requirement because the clustering method used in this approach increases the number of clusters.

At the same time, the "SDFA-only" approach has the lowest memory requirement because it has large cluster sizes, thus yielding small state codes. However, because this approach does not use character-aware encoding to minimize lookup complexity, the throughput is reduced. "Character Aware-SDFA" utilizes the properties of both "SDFA" and "Character Aware" approaches to produce good memory requirements and throughput. The result in Fig. 10 is obtained by evaluating the three designs in the DETER Testbed [43] using a 500-pattern DFA.

5.4. Pre-Filter false positive rate evaluation

Our pre-filter uses a set of short pattern representatives to filter out benign traffic. Even though this pre-filter is free of false negatives, it will introduce false positive matches. The false positive rate not only depends on the traffic mix and pattern rule set, it also depends on the pattern representatives. This section first evaluates the false positive rate using randomly generated traffic. Then a set of real traffic traces is employed to further analyze the false positive rate. Both experiments use the Snort rule set as the source of attack patterns.

Table 7
False positive rate evaluation.

Length of rep.	Number of rep.	Number of match	False positives (%)
2	12	4,454,886	99.77
3	18	261,504	96.08
4	20	24,638	54.42
5	21	11,000	6.87
6	23	10,245	0.01
8	23	10,244	0.00
19	23	10,244	0.00

Table 7 compares false positive rates with representatives of different lengths. The second column shows the minimal number of representatives required to represent all patterns for different representative lengths. The third column shows the number of matches for representatives of different lengths using the same randomly generated traffic. Using the number of matches, we can easily derive the false positive rate shown in the fourth column. The results show that the false positive rate will dramatically drop to a satisfactory value when the representative length equals 5. This experiment uses attack patterns from the ddos rule, which consists of 23 patterns, and the pattern length is 3 to 19, and with 41 unique characters.

Table 8 shows false positive rates for representatives with different lengths and packet traces. We use the web-coldfusion ruleset as the pattern source in which the longest pattern is 47 characters. Therefore, all matches from representative length 47 are true positive, which means the false positive rate is 0. Three traces from the orange file in defcon10 [44] are employed to evaluate the packet filter. For different representative lengths, a set of representatives is generated that will produce a different number of matches. For example, the number of patterns matched for length 2 using trace file Orange1.5 is 746 matches, and for length 47, it is 49 matches. In this case, we can see that the number of false positives for length 2 is 746 − 49 = 697 matches, or 0.934.

An interesting observation is that the number of matches for representative length 6 is greater than that of length 4 for the Orange1.5 and Orange2.6 traces but smaller for the Orange3.3 trace. We achieve this result because the group of representatives for a given length is

Table 8
False positive rates for different length representatives and packet traces.

Trace	Number of packets	Number of matches/false positive rate Length of representatives					
		2	4	6	8	16	47
Org1.5	21986	746/0.94	69/0.29	97/0.49	78/0.37	70/0.30	49/0.0
Org2.6	119155	1280/0.98	158/0.80	180/0.82	69/0.54	38/0.16	32/0.0
Org3.3	121012	3196/0.99	240/0.84	192/0.80	145/0.73	57/0.32	39/0.0

selected by analyzing the pattern rules only. If prior knowledge of the trace is available, then a better group of representatives can be selected.

5.5. Pre-filter representative selection tradeoffs

The overall performance of our two-stage intrusion detection system can be optimized by evaluating the tradeoff between the number of representatives and false positives with respect to the length of the representative. Fig. 11 shows that the longer the length of representative r, the greater the number of representatives R; this will increase the memory requirement. However, if we select the length of the representatives to be extremely short, the false positive rate will be higher, as we can see in Fig. 12. A high false positive rate will place an unnecessary load on the Verifier.

5.6. Speedup due to pre-filter

Assuming that the full pattern length verifier is the bottleneck of our system, the overall pattern matching engine performance will increase if the pre-filter reduces traffic to the verifier. The throughput speedup is defined as the ratio of the total number of packets to the number of packets scanned by the verifier. The total number of packets forwarded to the verifier includes truly malicious packets and false positive packets identified by the pre-filter. Fig. 13 evaluates how the representative length impacts the system throughput speedup. This figure shows that the speedup can be up to 1000 times.

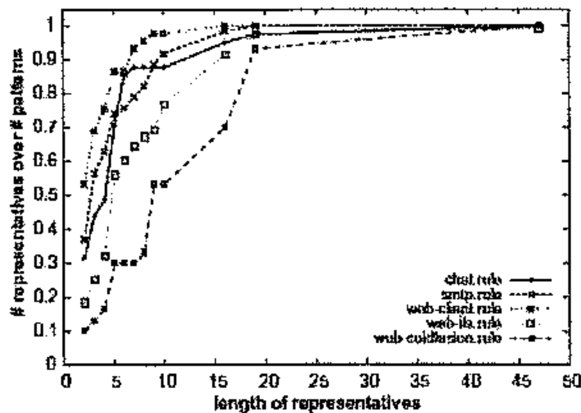


Fig. 11. Percentage of representatives to cover all patterns.

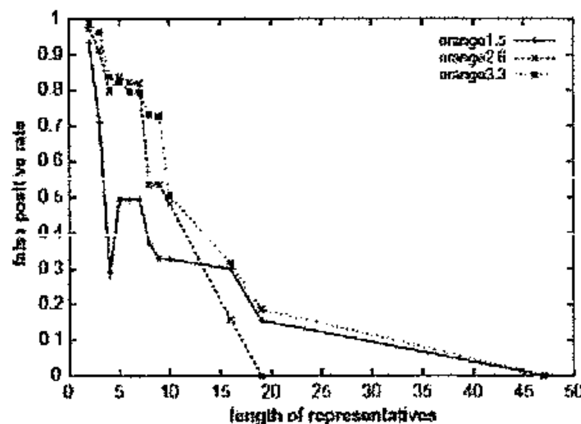


Fig. 12. False positive rate decreasing with representative length increasing.

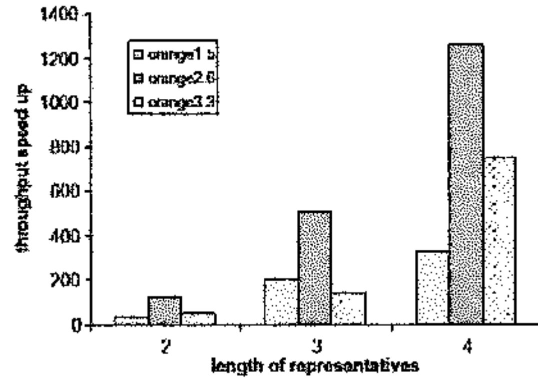


Fig. 13. Throughput speedup vs. length of representatives (normalized to non filter approach).

The data in Fig. 13 is generated using the values from Table 8. For example, trace orange1.5 has 21,986 packets with representative length 4. The number of packets that have to be verified is 69, as shown in Table 8. Therefore, the system speedup is 318 (21,986/69) times. The false positive rate and throughput speedup are both dependent on the trace file; however, they are independent of each other. For example, for $r = 4$, the false positive rate is 0.29 with trace orange1.5 and 0.80 with trace orange2.6. Even though the false positive rate of trace orange2.6 is higher than trace orange1.5, the throughput speedup for trace orange2.6 is 754 (119,155/158) times, which is higher than the throughput speedup for trace orange1.5 (318 times), as shown in Fig. 13.

5.7. Results summary

The results lead to three conclusions regarding our adaptive pattern matching engine. First, our "character-aware" encoding techniques, split DFA and state-collapsing optimization reduce the memory requirement dramatically, as shown in Table 6. Built from these techniques, our verifier engine is the most memory efficient in the reported literature, as shown in Fig. 8. Second, our pre-filter uses pattern representatives to filter out the majority of benign traffic without false negatives, yet it improves the overall system throughput, as shown in Fig. 13. Finally, our pattern matching engine does not require any special hardware to achieve high performance and memory efficiency.

6. Conclusion

In this paper, we introduced an adaptive and scalable network intrusion detection system to address ever-changing network attacks. The adaptive capability is based on memory efficiency and ease-of-reconfiguration for our no-

vel pattern matching engine. Our two-stage pattern matching engine can achieve a worst-case guaranteed line rate performance independent of the number of patterns and network traffic.

We believe our scalable IDS presents a solution that addresses the major problems of current IDSs: lack of scalability and poor performance. Our results substantiate this claim by showing the speedup and memory efficiency of our pattern matching engine. Specifically, the verifier, which is the major memory contributor of any intrusion detection system, requires less than 4 bytes of memory per pattern character (i.e., Table 6). This result is better than the results in related work, as shown in Fig. 8, and it allows our system to be scalable with thousands of attack patterns. We also show that our engine achieves a speedup that allows for scalability with current and future network line rates. Specifically, our engine can offload a majority of traffic from the verifier and achieve an overall speedup of hundreds of times (Fig. 13). This advancement, coupled with our platform independent design, means that our engine has the potential to permit line-rate deep-packet inspection on a variety of systems.

It is conceivable that our adaptable IDS will be a basic function of future network systems (e.g., routers), particularly in light of the expansion of networks to incorporate sensors, mobile wireless devices, and other end-systems that have resource constraints when executing complex security functions.

References

[1] B.K. Hyang-ah Kim, Autograph: toward automated, distributed worm signature detection, in: Proceedings of the 13th Usenix Security Symposium, 2004, pp. 271–286.

[2] S. Singh, C. Estan, G. Varghese, S. Savage, Automated worm fingerprinting, in: OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, USENIX Association, Berkeley, CA, USA, 2004, pp. 45–60.

[3] K. Griffin, S. Schneider, X. Hu, T. Chineh, Automatic generation of string signatures for malware detection, in: RAID'09, 2009, pp. 101–120.

[4] J. Caballero, Z. Liang, P. Poosankam, D. Song, Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration, in: RAID'09, 2009, pp. 161–181.

[5] F. Yu, R.H. Katz, T.V. Lakshman, Gigabit rate packet pattern-matching using tcam, in: ICNP '04: Proceedings of the Network Protocols, 12th IEEE International Conference on (ICNP'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 174–183.

[6] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, J.W. Lockwood, Deep packet inspection using parallel Bloom filters, IEEE Micro 24 (1) (2004) 52–61.

[7] B. Soewito, L. Vespa, A. Mahajan, N. Weng, H. Wang, Self addressable memory based fsm (sam fsm): a scalable intrusion detection engine, IEEE Network Special Issue on Recent Developments in Network Intrusion Detection 23 (1) (2009) 14–21.

[8] L. Vespa, N. Weng, B. Soewito, Optimized memory based accelerator for scalable pattern matching, Microprocessors and Microsystems 33 (7–8) (2009) 469–482.

[10] B. Soewito, N. Weng, Methodology for evaluating dna pattern searching algorithms on multiprocessor, in: Proceedings of IEEE 7th International Symposium on Bioinformatics and BioEngineering (BIBE), IEEE, Boston, MA, USA, 2007, pp. 570–577.

[11] J. Hrisson, G. Payen, R. Gherbi, A 3d pattern matching algorithm for dna sequences, Bioinformatics 23 (6) (2007) 680–686.

[12] N. Tuck, T. Sherwood, B. Calder, G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection., in: Proceedings of the IEEE Infocom Conference, 2004, pp. 333–340.

[13] D. Denning, An intrusion-detection model, IEEE Transactions on Software Engineering 13 (2) (1987) 222–232.

[14] M. Roesch, Snort – lightweight intrusion detection for networks, in: Proceedings of the 13th Systems Administration Conference, 1999.

[15] Snort, Snort Rule Database, 2007. <http://www.snort.org/pub-bin/downloads.cgi>.

[16] Sourcefire, Inc., Sourcefire 3D System, 2010. <http://www.sourcefire.com/>.

[17] Bro Intrusion Detection System, 2009. <http://www.bro-ids.org/>.

[18] Cisco Systems, Inc., Cisco Itrusion Prevention System, 2010. <http://www.cisco.com/en/US/products/sw/secursw/ps2113/index.html>.

[19] Ax3soft Corporate, Sax2 Expert NIDS, 2010. <http://www.ids-sax2.com/>.

[20] NIKSUN Inc., NIKSUN NetDetector, 2010. <http://www.niksun.com/product.php?id=4>.

[21] Symantec Corporation, Symantec Endpoint Protection, 2010. <http://www.symantec.com/business/endpoint-protection>.

[22] S. Acharya, B.N. Mills, M. Abliz, T. Znati, J. Wang, Z. Ge, A.G. Greenberg, Optwall: a hierarchical traffic-aware firewall, in: NDSS, 2007, pp. 528–533.

[23] J.D. Brutlag, Aberrant behavior detection in time series for network monitoring, in: LISA '00: Proceedings of the 14th USENIX conference on System administration, USENIX Association, Berkeley, CA, USA, 2000, pp. 139–146.

[24] P. Barford, J. Kline, D. Plonka, A. Ron, A signal analysis of network traffic anomalies, in: IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, ACM, New York, NY, USA, 2002, pp. 71–82.

[25] Y. Gu, A. McCallum, D. Towsley, Detecting anomalies in network traffic using maximum entropy estimation, in: IMC '05: Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, USENIX Association, Berkeley, CA, USA, 2005, p. 32.

[26] A. Lakhina, M. Crovella, C. Diot, Characterization of network-wide anomalies in traffic flows, in: IMC '04: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, ACM, New York, NY, USA, 2004, pp. 201–206.

[27] V.A. Siris, F. Papagalou, Application of anomaly detection algorithms for detecting syn flooding attacks, in: Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM'04), 2004, pp. 2050–2054.

[28] S. Shanbhag, T. Wolf, Accurate anomaly detection through parallelism, IEEE Network Special Issue on Recent Developments in Network Intrusion Detection 23 (1) (2009) 22–29.

[29] Intel Corp., Intel Second Generation Network Processor, 2002. <http://www.in-tel.com/de-sign/net-work/pro-ducts/np-fami-ly/ixp2400.htm>.

[30] A. Aho, M. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–343.

[31] B. Commentz-Walter, A string matching algorithm fast on the average, in: Proceedings of the 6th International Colloquium on Automata, Languages and Programming, vol. 71, 1979, pp. 118–131.

[32] S. Wu, Manber, A fast algorithm for multi-pattern searching, Technical Report TR94-17, Department of Computer Science, University of Arizona, 1994.

[33] K.-K. Tseng, Y.-D. Lin, T.-H. Lee, Y.-C. Lai, A parallel automaton string matching with pre-hashing and root-indexing techniques for content filtering coprocesser, in: Proceedings of the 2005 IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05), 2005, pp. 113–118.

[34] I. Sourdis, V. Dimopoulos, D. Pnevmatikatos, S. Vassiliadis, Packet pre-filtering for network intrusion detection, in: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, 2006, pp. 183–192.

[35] M. Aldwairi, T. Conte, P. Franzon, Configurable string matching hardware for speeding up intrusion detection, SIGARCH Computer Architechture News 33 (1) (2005) 99–107.

[36] Y.H. Cho, S. Navab, W.H. Mangione Smith, Specialized hardware for deep network packet filtering, in: 12th International Conference on Field-Programmable Logic and Applications, Springer-Verlag, London, UK, 2002, pp. 452–461.

[37] S.-G.M. Hamid Sarbazi-Azad, Behrooz Parhami, S. Hessabi, A multi-gb/s parallel string matching engine for intrusion detection systems, Advances in Computer Science and Engineering 6 (2008) 847–851.

[38] L. Tan, B. Brotherton, T. Sherwood, Bit-split string-matching engines for intrusion detection and prevention, ACM Transactions on Architechture Code Optimization 3 (1) (2006) 3–34.

[39] N. Hua, H. Song, T. Lakshman, Variable-stride multi-pattern matching for scalable deep packet inspection, in: INFOCOM 2009, IEEE, 2009, pp. 415–423.

[40] B.C. Brodie, D.E. Taylor, R.K. Cytron, A scalable architecture for high-throughput regular-expression pattern matching, SIGARCH Computer Architecture News 34 (2) (2006) 191–202.
[41] J. van Lunteren, High-performance pattern-matching for intrusion detection, in: Proceedings of the 25th IEEE International Conference on Computer Communications, 2006, pp. 1–13.
[43] S. Schwab, B. Wilson, C. Ko, A. Hussain, Seer: a security experimentation environment for deter, in: Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test on DETER Community Workshop on Cyber Security Experimentation and Test 2007, USENIX Association, Berkeley, CA, USA, 2007.
[44] The Shmoo Group, Capture the Capture the Flag Data: Defcon10, <http://cctf.shmoo.com>, (accessed 01.03.07).

Ning Weng is an assistant professor in the Department of Electrical and Computer Engineering at Southern Illinois University Carbondale. He received a Ph.D. degree in Electrical and Computer Engineering from University of Massachusetts Amherst in 2005. His research interests are in the areas of computer architecture, computer networks, and embedded systems.

Lucas Vespa is a Ph.D. student and Instructor in the Department of Electrical and Computer Engineering at Southern Illinois University Carbondale. He obtained a M.S. in Electrical and Computer Engineering and his research interests include networks and security.

Benfano Soewito received the M.S. degree from Electrical and Computer Engineering at Southern Illinois University Carbondale in 2004. He is currently pursuing the Ph.D. degree in the Department of Electrical and Computer Engineering at Southern Illinois University Carbondale. His research interests are network processing system, network intrusion detection, and multi cores system programming.