The Policy Mapping Algorithm for High-speed Firewall Policy Verifying

Suchart Khummanee and Kitt Tientanopajai
(Corresponding author: Suchart Khummanee)

Department of Computer Engineering, Khon Kaen University
Khon Kaen 40002, Thailand
(Email: khummanee@gmail.com)
(Received Feb. 8, 2015; revised and accepted June 5 & Aug. 12, 2015)

Abstract

In this paper, we have proposed a novel algorithm and data structures to improve the speed of firewall policy verification. it is called the policy mapping (PMAP). Time complexity of the proposed technique is *O(1)* to verify incoming-outgoing packets against the firewall policy. Besides, the algorithm is not limited to handle IP network classes as IPSET which is the top of high-speed firewall open source today. PMAP can also optimize the firewall rule decision by employing the firewall decision state diagram (FDSD) to clarify ordering of policy verifying. The consumed memory of PMAP is reasonable. It consumes the memory usage around 3.27 GB for maintaining rule data structures processing the firewall rule at 5,000 rules.

Keywords: Firewall policy, packet matching, packet verifying, policy mapping, policy verifying

1 Introduction

In the realm of network security, firewalls are an essential tool for protecting against undesirable traffic from untrusted networks. Generally, firewalls are usually equipped at the gateway between trusted (Private network) and untrusted networks (Public network) as shown in Figure 1. Firewalls consider inbound-outbound packets passing through itself by following the defined policies (technically called rules). Firewall rules mean the instruction sets compounded from various conditions e.g., IP address, protocol, port number and action. If a packet matches one of the defined rules in the firewall, either *accept* or *deny* is selected from the action field. The *accept* action allows packets to pass the firewall; on the other hand, the *deny* action entirely drops packets to the trash can as shown in Table 1.

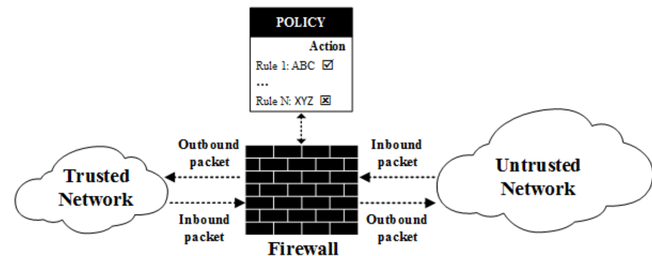

Figure 1: The basic firewall operation and installation

According to Table 1, rule no. 7 (r_7) represents that firewall permits source IP addresses ranging from 10.0.0.0 to 10.0.0.255 (256 hosts) onto destination IP addresses in the range of 20.0.0.0 to 20.0.0.255 (256 hosts), any source ports (0 - 65,535), a destination port number 80 and 443, and TCP or UDP protocol to pass through the firewall. In contrast, rule no. 4 (r_4) always drops every packet form source IP addresses ranging from 10.0.0.0 to 10.0.0.3 onto destination IP addresses in the range of 20.0.0.0 to 20.0.0.3, any source port, a destination port number 80, and both protocols. Besides, firewalls always set the final rule (usually called an implicit denying rule: r_8) at the bottom of the rule list by dropping all packets that are not matched with the above rules.

Basically, firewall rules are executed from the top to bottom, denoted as $r_1 \Longrightarrow r_2 \Longrightarrow \cdots \Longrightarrow r_n$. Firewalls working in this manner are called Rule-List or Rule-Base firewalls. For evaluating the effectiveness of Rule-Base firewall verification, the time complexity has very slow efficiency, that is *O(n)*, where n is the number of firewall rules. Nevertheless, memory consuming is quite small. Rule-Base firewalls are appropriate for individual persons and small businesses. However, it is not proper for large companies because firewall rules are usually diverse. Chapple et al. [3] surveyed and found that the size of firewall rule lists ranged from 2 to 17,000 rules, and an average of rules around 140 - 200 approximately. In a large organization, a firewall has about 2,000 rules, with each rule checking between 4 and 7 fields. Thus, the firewall needs to verify rules against 14,000 times per one

Table 1: Firewall rule examples

No.	Source IP	Destination IP	Source Port	Destination Port	Protocol	Action
r_1	10.0.0.10	20.0.0.2	*	21 - 22	TCP, UDP	ACCEPT
r_2	10.0.0.10	20.0.0.2	5000	23 - 25	TCP, UDP	DENY
r_3	10.0.0.0-3	20.0.0.2	*	80	TCP, UDP	DENY
r_4	10.0.0.0-3	20.0.0.0-3	*	80	TCP, UDP	DENY
r_5	10.0.0.2	20.0.0.2	*	80 - 145	TCP, UDP	ACCEPT
r_6	10.0.0.*	20.0.0.*	*	*	TCP, UDP	ACCEPT
r_7	10.0.0.*	20.0.0.*	*	80, 443	TCP, UDP	ACCEPT
r_8	*	*	*	*	*	DENY

packet in the worst case (7 x 2,000).

To improve performance of firewall rule verification, Clark and Agah [4] presented a firewall policy diagram (FPD) and data structures to seek and solve the problem of a large network behavior in firewall policy. Liu and Gouda [10] proposed a diverse firewall approach that produced time complexity as $O(n^d)$, where n is the number of firewall rules and and d is the number of checked fields. After that Acharya and Gouda [1] claimed that their liner time algorithm improved on the diverse firewall from $O(n^d)$ to $O(nd)$. For dealing with rules and continually increasing traffic, researchers tried to solve this problem. Hamed et al. [6] presented a statistical search tree model for filtering and matching packets, whose computational complexity was $O(n * log(n))$, Gouda and Liu [5] proposed the firewall decision diagram (FDD) which clarified packet matching and increased the speed of verifying to $O(log(n))$. In addition, Puangpronpitag et al. [8] introduced a single domain decision concept to get rid of firewall rule conflicts and improve the verifying speed to be $O(log(n))$ by using tree structure. A tree-rule proposed on cloud computing by Xiangjian et al. [15], produced a processing time for $O(log(n))$. Due to huge traffic of a gigabit in the network today, IPSet [12] which is under the Netfilter project and the top of high-speed firewall open source nowadays is $O(1)$ for firewall rule verification. It rearranges the rules of Rule-Base firewall to groups of similar behavior like IP addresses in the same class before deploying the modified groups to a hashing method. However, one drawback of IPSet is the limitation of IP class management. It can only be implemented in a IP Classes C and B – excluding Class A.

According to the weakness of the IPSet as mentioned above, our research focuses on the firewall policy verification because it is the major key to reduce a firewall's performance. Consequently, in this paper, we propose the novel algorithm for high-speed firewall rule verification, called **the policy mapping (PMAP)**, which is *O(1)*. PMAP also handles all IP network classes. The rest of the paper is organized as follows: Section 2 presents the firewall background, the design of policy mapping and implementation are explained in Section 3. In Section 4, we demonstrate the performance evaluation of firewalls, which is divided into two sorts, i.e., the computation time and space complexity. In addition, we compare the performance of our proposed model against other firewalls. Finally, we give conclusions and future work in Section 5.

2 Firewall Background

2.1 The Rule-Base Firewall

Basically, the Rule-Base firewall rule consists of six conditions, which are source IP address (*src_ip*), destination IP address (*dst_ip*), source port (*src_port*), destination port (*dst_port*), protocol (*pro*) and an action (*act*). The first five conditions (*src_ip* to *pro*) are called the predicate, and an *act* is called the decision, which is formulated as follows:

$$\langle predicate \rangle \rightarrow \langle decision \rangle$$

$$\langle src_ip \wedge dst_ip \wedge src_port \wedge dst_port \wedge pro \rangle \rightarrow \langle act \rangle$$

Where $\wedge$ denotes AND operation. *src_ip* and *dst_ip* are unique addresses used to locate and identify a device over the network. The unique address is a 32-bit number (Internet Protocol Version 4: IPv4) consisting of 4 octets (oct) separated by dots such as 192.168.10.100, 200.0.*.* ($* \in \mathbb{Z}_n^+ : n \geq 0 \wedge n \leq 255$). Indeed, an IP address can be transformed to the positive integer ($\mathbb{Z}_n^+ : n \geq 0, n \leq 2^{32} - 1$) as shown the following equation.

$$Octet_1 \times 2^{24} + Octet_2 \times 2^{16} + Octet_3 \times 2^8 + Octet_4 \times 2^0$$

For example, converting an IP address such as 192.168.10.100 to $\mathbb{Z}_n^+$, the conversion process can be performed as follows:

$$192 \times 2^{24} + 168 \times 2^{16} + 10 \times 2^8 + 100 \times 2^0 = 3,232,238,180.$$

src_port and *dst_port* are used to distinguish itself from other applications running over TCP or UDP by reserving and using a 16-bit port number. The port number 80 [14], for example, is reserved for the Hyper text transfer protocol (HTTP), Domain Name System (DNS) is assigned to port number 53, etc. Also, the reserved ports are called the well-known ports. The last one of the predicates is a standard protocol [7] which defines a method of

exchanging data over a computer network such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). It has an 8-bit number ranging from 0 - 255, i.e., the port no. 6 for TCP, no. 17 for UDP respectively.

Finally, *act* is the decision that is either *accept* or *deny*. The *accept* allows the traffic or packets that can pass through the firewalls to the destination networks, while the others are discarded.

2.2 Defining Rule-based Firewall Rule

Let r denote a firewall rule, and n denote the rule number ($n \in \mathbb{Z}^+$ and $n \neq 0$). We frequently specify the firewall by writing:

$r_1 : \langle predicate_1 \rangle \langle decision_1 \rangle \Rightarrow 1^{st}$ rule.
$r_2 : \langle predicate_2 \rangle \langle decision_2 \rangle \Rightarrow 2^{nd}$ rule.
...
$r_n : \langle predicate_n \rangle \langle decision_n \rangle \Rightarrow$ the final rule.

Table 1 demonstrates examples of the actual Rule-Base firewall rules. For the sake of simplicity, we represent the range of host IP addresses using "-", and "*" for any range of IP addresses. In r_3, for example, *src_ip* is 10.0.0.0-3, which represents the range of IP address from 10.0.0.0 - 10.0.0.3 (or 167,772,160 - 167,772,163 in a decimal format), and 10.0.0.* ($src_ip_{r_6}$) substitutes a set of IP addresses between 10.0.0.0 and 10.0.0.255 (256 hosts).

2.3 Packet Matching vs. Mismatching

Let p denote packets flowing in and out of firewalls, and x denote the packet number by $x \in \mathbb{Z}^+$ and $x \neq 0$. We can form a set of packets to be $p_x = \{p_1, p_2, ..., p_x\}$. The packet is a formatted unit of data carried by a packet mode in the computer network. Generally, a packet consists of two types of data, i.e., control commands and user information (payload). The control commands provide the communication standards that the network needs to deliver the user information, i.e., source and destination IP addresses, error detection codes, sequencing information and so forth. In fact, control commands are set in both headers and trailers of the packet, and the payload is placed in the middle. In this paper, we only take four key fields from the packet for verifying against firewall rules, which are *src_ip*, *dst_ip*, *dst_port* and *pro*. Assuming that an incoming packet (p_1) flows into the firewall; it is formed from *src_ip* = 10.0.0.10, *dst_ip* = 20.0.0.2, *src_port* = 1,024, *dst_port* = 21 and *pro* = TCP. We can rewrite a set of packets $p_1 = \{src_ip_1, dst_ip_1, src_port_1, dst_port_1, pro_1\} = \{10.0.0.10_1,\ 20.0.0.2_1,\ 1024_1,\ 21_1,\ \mathrm{TCP}_1\}$. The packet p_1 is matched with firewall rules r_1, r_6 and r_8 in the Table 1. However, for Rule-Base firewalls, the packet p_1 will be always executed with the top of firewall rules; which is r_1 only.

The Packet Matching Definition. The packet p_i matches the firewall rule r_n if $(src_ip_{p_i} \in src_ip_{r_n}) \wedge (dst_ip_{p_i} \in dst_ip_{r_n}) \wedge (src_port_{p_i} \in src_port_{r_n}) \wedge (dst_port_{p_i} \in dst_port_{r_n}) \wedge (pro_{p_i} \in pro_{r_n}) =$ TRUE. The statement "p_i **matches** r_n" is written $p_i \in r_n$. For example, given the packet $P_1 = \{10.0.0.2_1,\ 20.0.0.2_1,\ 1234_1,\ 80_1,\ TCP_1\}$ and $P_2 = \{22.2.0.10_2, 20.0.0.5_2, 5000_2, 37_2, UDP_2\}$, thus $P_1 \in$ firewall rule $r_3 - r_8$ (Table 1), and $P_2 \in r_8$ only.

The Packet Mismatching Definition. The packet p_i mismatches the firewall rule r_n if $(src_ip_{p_i} \notin src_ip_{r_n}) \vee (dst_ip_{p_i} \notin dst_ip_{r_n}) \vee (src_port_{p_i} \notin src_port_{r_n}) \vee (dst_port_{p_i} \notin dst_port_{r_n}) \vee (pro_{p_i} \notin pro_{r_n}) =$ TRUE. The statement "p_i **mismatches** r_n" is written $p_i \notin r_n$. For example, let the packet $P_3 = \{10.0.0.10_3, 20.0.0.2_3, 1234_3, 20_3, TCP_3\}$, $P_3 \in r_6$ and r_8 only. On the other hand, $P_3 \notin r_1 - r_5$ and r_7.

2.4 Firewall Rule Verification

Firewall rule verification is the matching process between inbound-outbound packets against the defined rules. The result of matching in a normal case is either an *acceptance* or a *denial*. On the other hand, if firewall rules are the anomaly or conflict, the matching result is probably both acceptance and denial simultaneously. We can distinguish the processing of firewall rules verification into three steps. Firstly, searching the first firewall rule that matches with the packet as fast as possible. Secondly, investigating conflicts of firewall rules. Lastly, analyzing for vulnerabilities and security risks in the rules. The performance evaluation of firewall rule verification is proportional to the speed of searching algorithms. For example, matching a packet with Rule-Base firewall (Sequential searching) is $O(n)$, $O(log_2(n))$ for tree structures, and $O(1)$ for hashing approach, where n is the number of firewall rules.

3 The Policy Mapping Design and Implementation

In this section, we explain the concept and development of our policy mapping model. The aims of the model are to improve the processing time of policy verifying, defeat limitations of IPSet and suitably consume the memory space. Thus, we then focus on hashing functions (the fastest algorithm of searching) to solve the speed of the firewall rule verification. However, hashing functions have one major weakness, which is the collision of hashed keys (key = $\boldsymbol{H}(x)$, H is hashing function and x is the information). With the massive size of data or information, the collision probability of hashed keys will be increasingly high. To avoid this problem, we deployed arrays to be the data structure instead of the $\boldsymbol{H}(x)$. Besides, arrays are also very fast to access and retrieve the stored data without any collision, and they are also easier to understand and implement array structures.

3.1 Key Contributions

In this paper, we make four major contributions. First, we optimize rules of the Rule-Base firewall to a state diagram in order that the order of rules are cleaned form an ambiguous packet matching, called the firewall decision state diagram (FDSD). Second, we present the policy mapping algorithm (PMAP) and data structures for fast handling firewall policies. Third, we get rid of limitations of IPSet and key duplication of hashing function by PMAP. Last, we conduct extensive experiments to evaluate our proposed model against other models.

3.2 PMAP Design

There are five procedures to be included in the design of PMAP:

1) Make firewall rules in the traditional style (Rule-Base) as shown in Step 1 of Figure 2 and Table 2;

2) Build a decision state diagram structure (DSD) from the rule list from Step 1 by using the firewall decision state diagram algorithm (FDSD);

3) Map DSD from step 2 to array structures by the policy mapping algorithm (PMAP);

4) Get the mapped DSD in a format of array data structures;

5) Test the matching speed of PMAP and evaluate performance.

Step 1: Making Rule-Base Firewall Rules

Among user interfaces of firewalls such as Rule-Base ([2, 12, 13]), tree styles ([15]) and the structured query language (SQL) ([11]); almost all of firewall interfaces are Rule-Base. The reason is that it is influenced by the nature of the reading and writing of humans who generally read rules from left-to-right and top-to-bottom; and it is still a popular user interface nowadays. Therefore, we still use the Rule-Base interface to create firewall rules in Step 1. In order to simplify our model, we have presented easy firewall rules that consist of four fields: src_ip, dst_ip, dst_port and act as shown in Table 2.

The src_ip, dst_ip and dst_port has a maximum scope in the range from 1 to 100 only. An a means an acceptance action and d indicates a denial action. For instance, r_1 has source IP addresses between 10 and 30 (src_ip), destination IP addresses (dst_ip) ranging from 20 to 30, a destination port (dst_port) as 80 and an acceptance action (a). In real experiments, we have added one field into firewall rules, that is pro.

Step 2: Building the DSD

The firewall decision state diagram (FDSD) is a great tool for optimizing confusing Rule-Base firewall rules to a clearly firewall decision route. For example, in Table 2, assume that a packet p_i is composed of $src_ip_i = 15$, $dst_ip_i = 25$ and $dst_port_i = 80$. Thus, p_i matches both r_1 and r_3 but both of them are in conflict ($act_{r_1} \neq act_{r_3}$). In case of this rule base, p_i is only matched with r_1. However, rule bases often make administrators confused. To solve the problem, we demonstrate the FDSD to correct uncertain rules. This model was adapted from the firewall decision diagram of Liu [9]. The demonstrations for building DSD by FDSD are shown in Algorithm 1.

Table 2: Easy firewall rules for proving PMAP

no.	src_ip	dst_ip	dst_port	act
r_1	10 - 30	20 - 30	80	***a***
r_2	1 - 15	50 - 60	25 - 30	***a***
r_3	1 - 40	25 - 35	80	***d***
r_4	15 - 45	1 - 100	60 - 90	***d***

We first start a full description of the FDSD procedure in Situation 1 (in the black circle) of Figure 3. The FDSD first reads sets of dst_port of $r_{1,2,3,4}$ from the rule list of the firewall in Table 2. After that, it builds a start state (S0) which is a first state of DSD. Next, it makes a new state ($S1_1$) and creates a link from S0 to $S1_1$ (S0-$S1_1$). This link means a transition state from S0 to $S1_1$ and contains a set of $dst_port_{r_1}$ as $\{80-80\}$. In Situation 2, FDSD reads $dst_port_{r_2}$ form the rule list. The $dst_port_{r_2}$ ({25-30}) is compared with a set of the transition states from S0 to $S1_1$ (S0-$S1_1$ = {80}) which $dst_port_{r_2}$ is not a subset of $dst_port_{r_1}$ ($dst_port_{r_2} \nsubseteq dst_port_{r_1}$). Consequently, FDSD needs to build a new state, namely $S1_2$ in Situation 2, and it establishes a transitional link from S0 to $S1_2$ (S0-$S1_2$). It assigns a set of $dst_port_{r_2}$ to the S0-$S1_2$. In Situation 3 of Figure 3, FDSD reads $dst_port_{r_3}$ form firewall rules and compares it with the transitional state S0-$S1_1$ first. As a result, $dst_port_{r_3}$ is the subset of S0-$S1_1$ ($dst_port_{r_3} \subseteq dst_port_{r_1}$: $\{80\} \subseteq \{80\}$). Hence, FDSD does not need to take any action. Lastly, FDSD inserts $dst_port_{r_4}$ ({60-90}) to DSD, it verifies $dst_port_{r_4}$ with the transitional state S0-$S1_1$ and S0-$S1_2$ respectively. The $dst_port_{r_4}$ is not subset of S0-$S1_2$ but it is a proper superset of S0-$S1_1$ ($dst_port_{r_4} \supset dst_port_{r_1}$: $\{60\text{-}90\} \supset \{80\}$). So, FDSD builds a new state $S1_3$, and makes a new transitional state S0-$S1_3$ in DSD. FDSD computes a set which assigns to S0-$S1_3$ by S0-$S1_3$ = $dst_port_{r_4}$ - $dst_port_{r_1}$ ({60-90} - {80} = {60-79, 81-90}). The first level of DSD is finished in Situation 4 of Figure 3 (dst_port level).

To establish the Level 2 (dst_ip level) of DSD, FDSD reads $dst_ip_{r1,2,3,4}$ from the firewall rules. First, it makes a new state $S2_1$ which is the first state in Level 2 as shown in Situation 5 of Figure 4. Then it links a new transition $S1_1$-$S2_1$ from a state $S1_1$ to $S2_1$, and sets dst_ip_{r1} to the link, that is {20-30}. In Situation 6, dst_ip_{r2} ({50-60}) is verified with $S1_1$-$S2_1$ ({20-30}), the result shows that dst_ip_{r2} is not a subset of $S1_1$-$S2_1$ ($\{50\text{-}60\} \not\subset \{20\text{-}30\}$). Consequently, FDSD builds a $S2_2$ state, links a new $S1_2$-$S2_2$ transition and assigns a set of dst_ip_{r2} ({50-60}) to

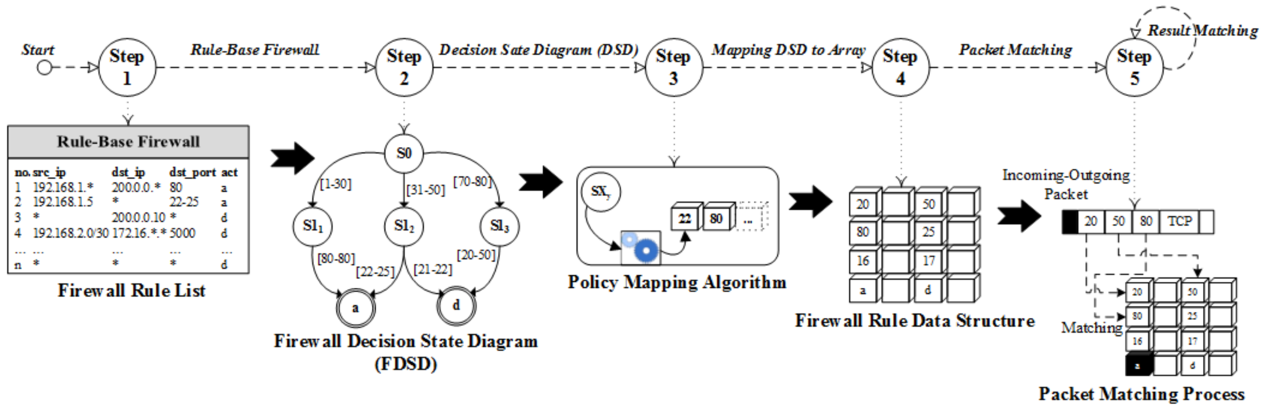

Figure 2: An overview of PMAP design

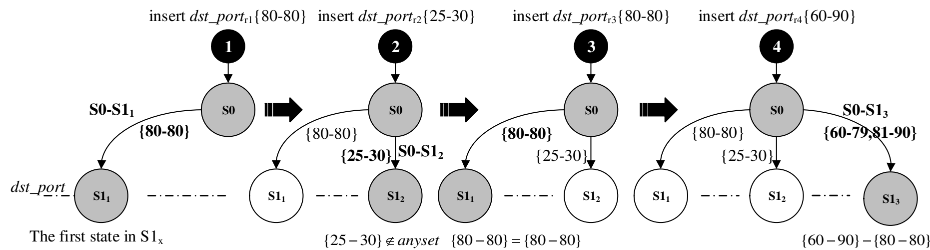

Figure 3: Inserting $dst_port_{r_{1,2,3,4}}$ to Level 1 of DSD (dst_port)

the $S2_2$ state respectively. Next (Situation 7), a set of dst_ip_{r3} ({25-35}) is inserted into the DSD. It is compared with $S1_1$-$S2_1$ and $S1_2$-$S2_2$, there are several elements in dst_ip_{r3} to be members of $S1_1$-$S2_1$ ($\{20\text{-}30\} \cap \{25\text{-}35\} = \{25, ..., 30\}$) but none of dst_ip_{r3} elements are in $S1_2$-$S2_2$. As a result, FDSD builds a new $S2_3$ state, links a transitive $S1_1$-$S2_3$ to $S2_3$ and sets a set {31-35} ({20-30} - {25-35} = {31, ..., 35}) to $S1_1$-$S2_3$ transition. The last situation (8) of Level 2, the dst_ip_{r4} ({1-100}) is a superset of $S1_1$-$S2_1$ and $S1_1$-$S2_3$ but it is not a subset of $S1_2$-$S2_2$ because they are different port numbers (port numbers of $S0\text{-}S1_2 \not\subset S0\text{-}S1_1$). So, FDSD needs to build a new state as $S2_4$ and $S2_5$, establishes a link $S1_1$-$S2_5$ to this state and defines two sets to $S1_1$-$S2_5$ transition to be {1-19, 36-100}(dst_ip_{r4} - ($S1_1$-$S2_1$) - ($S1_1$-$S2_3$) = {1-100} - {20-30} - {31-35} = {1-19, 36-100}), and defines a set to link $S1_3$-$S2_4$ as {1-100} respectively.

In the last level for building DSD (src_ip), the FDSD feeds $src_ip_{r1,2,3,4}$ to DSD like the previous situation. For example, src_ip_{r1} is {10-30}, it is processed in a state path from S0, $S1_1$ and $S2_1$ respectively as shown in Situation 9 of Figure 5. It suddenly builds the new sate $S3_1$ because it is the first state in this level there. In this state, $S3_1$ is assigned the a that means an acceptance state. Because of the action of r_1 is an acceptance ($act_{r1} = a$). According to Situation 10, FDSD inserts src_ip_{r2} ({1-15}) to the diagram. The src_ip_{r2} traverses in a path from S0 - $S1_2$ - $S2_2$ and is not a subset of any state. So, FDSD builds a new $S3_2$ state and assigns the status of this state to be a (act_{r2} = accept). Situation 11 in Figure 5, some elements of src_ip_{r3} ({1-40}) are a subset of $S2_1$-$S3_1$ ({10-30}) and some elements are not. Thus, FDSD creates a new $S3_3$ state and the $S2_1$-$S3_3$ transition, assigns the status of this node to be d. It sets a set of {1-9, 31-40} to the $S2_1$-$S3_3$ transition (src_ip_{r3} - $S2_1$-$S3_1$ = {1-40} - {10-30} = {1-9, 31-40}). For some elements that are not a subset of $S2_1$-$S3_1$, the FDSD builds a new state as $S3_4$ for storing a denial action (d) of r_3. The last situation of Level 3 is shown in Situation 12 of Figure 5.

Step 3: Mapping DSD to array data structures

In this section, we fully describe the process of mapping DSD to arrays applied to maintain clearness of firewall rules as shown in Algorithm 2.

From the Algorithm 2, the policy mapping (PMAP) starts by creating an array of one dimension that has the size equal to 1 x a maximum of the port number (in this example equal to 1 x 100), namely S0-S1 as shown in Figure 6. In case of real experiments, we set the size of the S0-S1 array to be 1 x 65,536 (Maximum port number = 0 - $(2^{16} - 1)$). Next, PMAP map sets in a transitional state $S0$-$S1_1$, $S0$-$S1_2$ and $S0$-$S1_3$ to S0-S1 array respectively. In case of $S0$-$S1_1$, it has a single destination port, that is the port number 80 ({80-80}). T_1 in Figure 6 shows the first tree or state path in DSD Level 1 which is used to mark a position for referring to an array of DSD Level 2. PMAP sets 1 (T_1) to an S0-S1 array in the position 80.

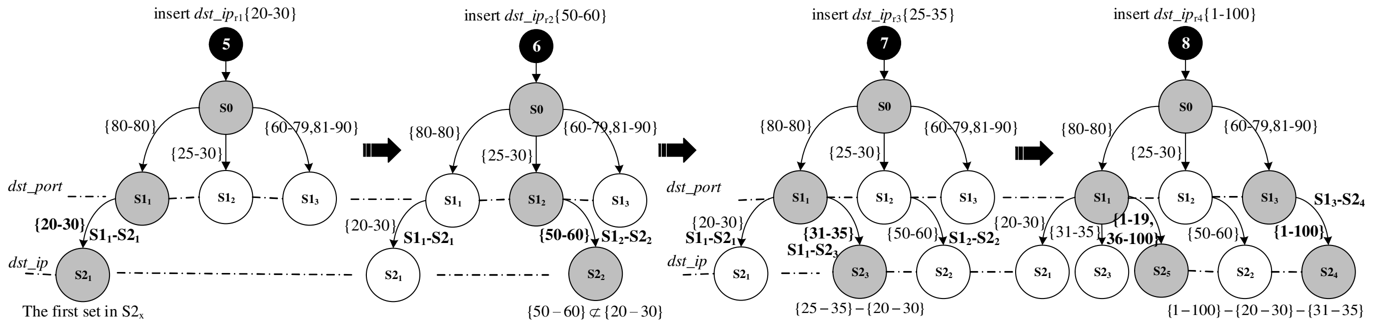

Figure 4: Inserting $dst_ip_{r1,2,3,4}$ to Level 2 of DSD (dst_ip)

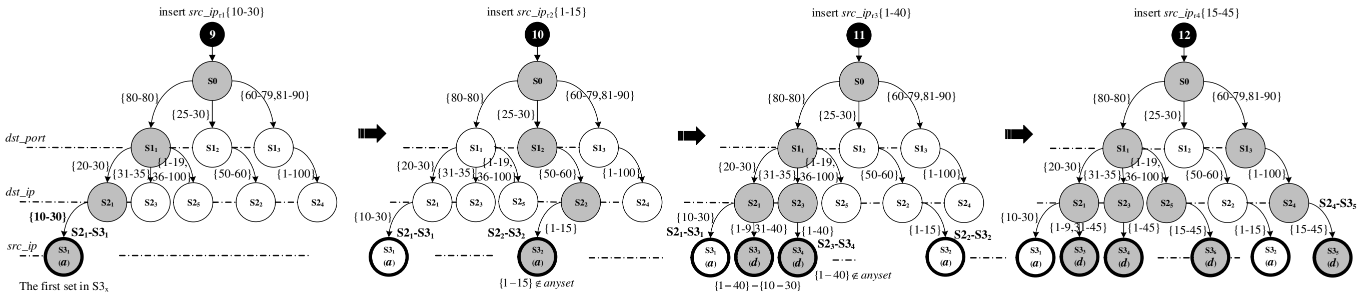

Figure 5: Inserting $src_ip_{r_{1,2,3,4}}$ to Level 3 of DSD (src_ip)

The number 1 refers to S1-S2 array in the row = 1 and column = $dst_ip_{p_i}$ (p_i = any packet) of the next level demonstrated in Figure 7. The S0-S1_2, which has the number of ports between 25 and 30, maps to S0-S1 array in the position 25 to 30. These positions are set to be 2 (T_2 in DSD). Lastly, S0-S1_3 has two subsets consisting of {60-79} and {81-90}. They are mapped to S0-S1 array in position 60 to 79, and 81 to 90 which positions are set to be 3 (T_3 in DSD). The total memory size that maintains DSD Level 1 is 200 bytes (1 row x 100 ports x unsigned 16-bit). An array of an unsigned 16-bit can refer to firewall rules between 1 and 65,536.

Tht map of Level 2 of DSD to S1-S2 array is illustrated in Figure 7. Firstly, PMAP builds the two dimensional array named S1-S2 that allocates the memory size as 20 Kb for handling 100 state paths (100 state paths x 100 of dst_ip x unsigned 16-bit). PMAP maps the transitional state S1_1-S2_1, S1_1-S2_3, S1_2-S2_2 and S1_3-S2_4 to the S1-S2 array respectively. The results show that S1-S2[1][20 - 30] = 1 (T_1), S1-S2[1][31 - 35] = 2 (T_2), S1-S2[2][50 - 60] = 3 (T_3), S1-S2[3][1 - 19] = 4 (T_4) and S1-S2[3][36 - 100] = 4 (T_4).

Mapping DSD Level 3 to S2-S3 array is displayed in Figure 8. PMAP maps a transition state S2_1-S3_1 to S2-S3[1][10-30] = a (T_1), S2_1-S3_3 to S2-S3[1][1-9] = d (T_2), S2_1-S3_3 to S2-S3[1][31-45] = d (T_2), S2_3-S3_4 to S2-S3[2][1-45] = d (T_3), S2_2-S3_2 to S2-S3[3][1-15] = a (T_4) and finally S2_4-S3_5 to S2-S3[4][15-45] = d (T_52) respectively.

Step 4: Getting the completed DSD in array structures

Firewall rules and state paths: A state path is a route that traverses from the starting state to the accepting state. For example, in Situation 12 of Figure 5, there are five state paths. The first state path is traversable from the starting state (S0) to S1_1, S2_1 and S3_1 (an accepting state) respectively. The second state path is from the starting state (S0), S1_1, S2_1 and S3_3 (an accepting state) and so forth. The number of firewall rules always exceeds the number of state paths (firewall rules $\geq$ state paths). For instance, in Table 2, there are four rules; however, they are generated to five state paths as shown in Situation 12 of Figure 5.

After Step 3 is complete, the arrays contained the DSD consist of S0-S1 for DSD Level 1, S1-S2 for DSD Level 2 and S2-S3 of DSD Level 3. The total memory size of arrays is around 40.2 Kb for holding 100 state paths in this simulation case. However, in the real implementations, the total size of arrays successively increases following by the number of state paths. In case of real implementations, arrays are complicated due to the size of src_ip (32 bits), dst_ip (32 bits), dst_port (16 bits) and pro (8 bits) as shown in Figure 9. Thus, the total of memory size for handling 5,000 state paths is 3.2 GB. That is S0-S1 (dst_port) = 1 x 65,536 x unsigned 16 bits integer x 5,000 = 655.36 MB; S1-S2 (dst_ip) and S2-S3 (src_ip) = 256 x 256 x 16 bits x 5,000 x 4 = 2,621.44 MB and S3-S4 (pro) = 1 x 256 x 16 bits x 5,000 = 2.56 MB.

Step 5: Testing the matching speed and evaluating the performance

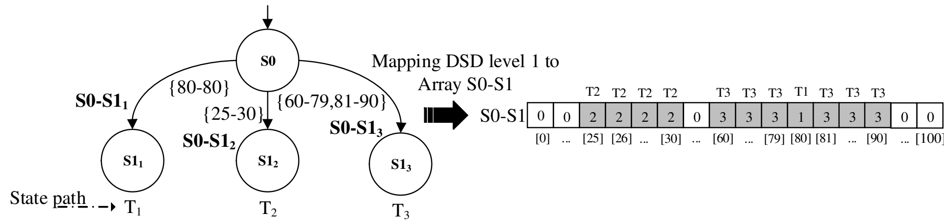

Figure 6: Mapping DSD Level 1 to one dimension array (S0-S1)

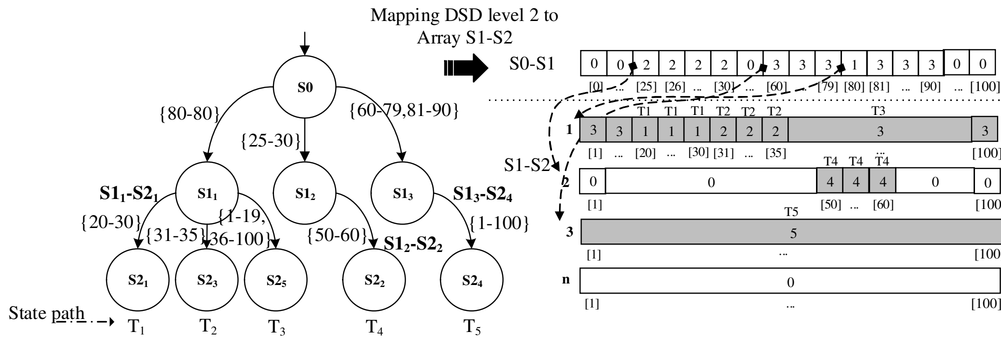

Figure 7: Mapping DSD Level 2 to two dimension array (S1-S2)

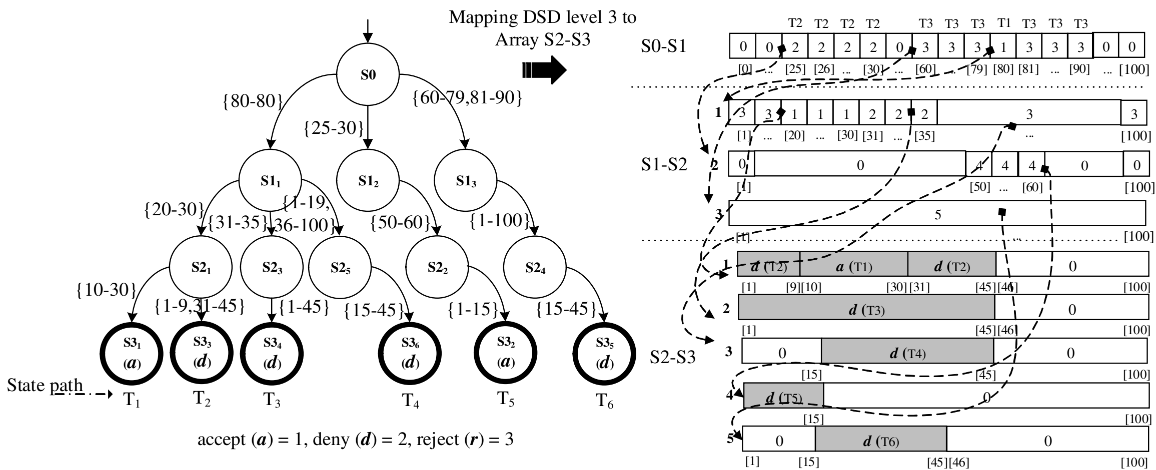

Figure 8: Mapping DSD Level 3 to two dimension array (S2-S3)

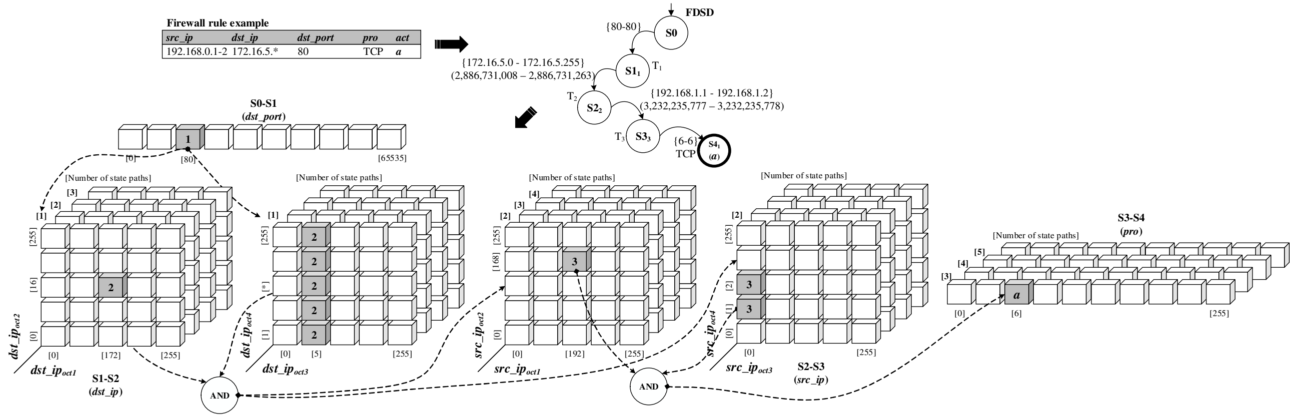

Figure 9: Arrays for real implementation

Algorithm 1 Firewall Decision State Diagram (FDSD)

1: **Input**: Rules $\{r_1, r_2, ..., r_n\}$, where $n \in \mathbb{Z}^+, n \neq 0$
2: **Output**: A decision state diagram (DSD)
3: **if** DSD = $\emptyset$ **then**
4: Build S0 *(starting state)*
5: Set l = 1
6: Set max_level = i *(i = the number of levels)*
7: **while** l $\leq$ max_level **do**
8: Set state, path, loop = 1
9: **while** loop $\leq$ n **do**
10: **if** $dst_port_{r_{loop}} \subseteq$ S(l-1)-S(l)$_{path}$ **then**
11: do nothing and break
12: **else**
13: Build the state S(l)$_{path}$
14: Create transitional state S(l-1)-S(l)$_{path}$
15: temp = S(l-1)-S(l)$_{path-1}$ - $dst_port_{r_{loop}}$
16: S(l-1)-S1(l)$_{path}$ = temp
17: **if** l = max_level **then**
18: Set S(l)$_{path}$ = $act_{r_{loop}}$ *(acceptance state)*
19: **end if**
20: path = path + 1
21: **end if**
22: loop = loop + 1
23: **end while**
24: l = l + 1
25: **end while**
26: **end if**
27: Return DSD
28: End

According to Steps 3 and 4 in Figure 2, the firewall rules are already stored in the arrays. Accessing the rules in arrays uses indexes which can directly access the data quickly. So, to operate any instructions on data in arrays is always one instruction ($O(1)$). The algorithm that is used for matching firewall rules in arrays is shown in Algorithm 3 and Figure 10.

3.3 The Policy Mapping Implementation

In this section we reveal the implementation of the policy mapping approach. The details are as follows.

Hardware and Software Development Tools. The policy mapping is developed on the Intel 64-bits processor, Core i7, 2GHz, installed memory (RAM) 8 GB DDR3 and hard disk 750 GB 5400 RPM. In addition to the software development, we chose Python language (version 3.4), Numpy and Psutil to implement our policy mapping running on MS Windows 8 operating system.

Firewall Rule and Packet Generator. In each performance testing scenario, the firewall rule generator software generates the random policies (or rules) from 5 to 5,000 rules, and the intelligently random packets (10,000 packets per epoch) from the packet generator software. According to the random packet algorithm, the algorithm is able to define the ratio of matched packets between 5 and 95% of packets that pass though the firewall.

4 The Performance Evaluation

This section details the performance evaluation of the firewall rule matching by comparing several approaches such as Rule-Base firewall, tree rule, and hashing. We set up the experimental scenarios as shown in Figure 11.

Firstly, the firewall rule generator software generates a number of firewall rules from 5, 10, 50, 100, 200, 350, 500, 750, 1,000, 2,000, 3,000, 4,000 and 5,000 respectively for evaluating the time-space complexity of the algorithms. The number of generated firewall rules in each round is executed 30 rounds per algorithm; that is, the number of generated rules x 30 x the number of tested algorithms (three algorithms for this paper) = 13 x 30 x 3 = 1,170 rounds.

In the next step, the packet generator generates the intelligent random packets for evaluating time-space com-

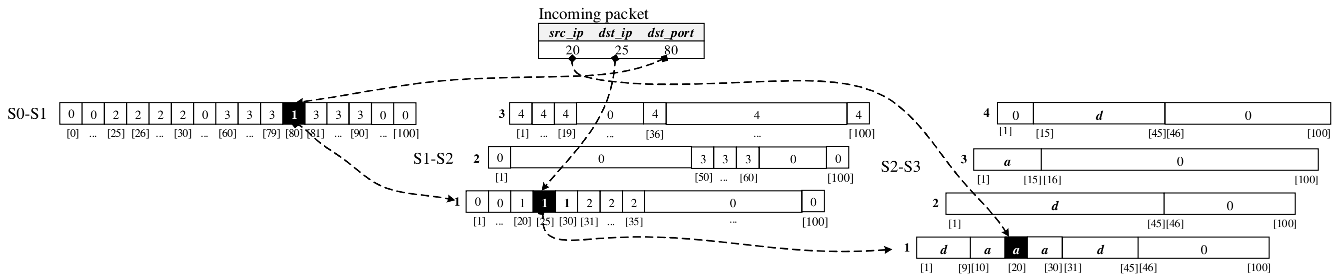

Figure 10: A packet_i matching example

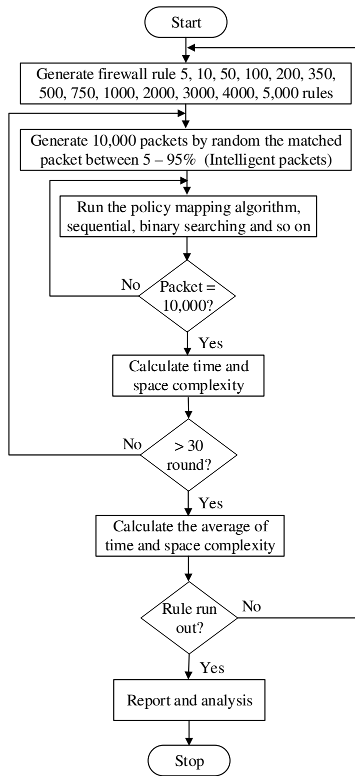

Figure 11: Performance evaluation for matching rules

plexity of each algorithm by 10,000 packets per round. The intelligent random packet applies the random algorithm of the packet generator, which is able to designate the percentage of packet matching with firewall rules. For example, 10 percent of intelligently packet matching of 10,000 packets is 1,000. In our testing, we chose the ratio of random for matching packet from 5 to 95 percent.

Next, we execute each algorithm against generated rules and random packets by a number of tested packets in each algorithm, which is 4,200,000 packets (13 (generated rule) x 10,000 (random packet) x 30 (round)). The chosen algorithms in our experiments are sequential-based approach (generic or Rule-Base firewall), tree and hashing (policy mapping) approach. Finally, we record and plot results (time and space complexity) of each algorithm for analyzing in the next section.

Table 3: Time for generating firewall rule structures (Generic = Rule-Base, Tree rule = Tree structure, and Policy mapping = hashing algorithm)

Time for generating rule structures (sec)			
No.	Generic	Tree rule	Policy mapping
5	0.00208	0.00190	0.01566
10	0.00156	0.00476	0.01562
50	0.00625	0.02187	0.04687
100	0.00521	0.07422	0.13188
200	0.01667	0.27411	0.17188
350	0.02761	0.83719	1.00080
500	0.04416	1.66761	1.32480
750	0.07992	3.78536	1.48440
1,000	0.11055	6.39945	1.78319
2,000	0.22052	24.11578	2.14030
3,000	0.22572	50.02818	2.30150
4,000	0.39503	93.91504	2.92790
5,000	0.49315	148.19096	5.62890

4.1 Time Complexity Analysis

We set up the criteria to evaluate the time complexity of each algorithm in this paper as follows: (i) the time for generating firewall rule structures, and (ii) the time for verifying (matching) firewall rules. The generating time denotes the period of declaring structure variables for storing the data (rules) including time for collecting data to the variables. The verifying time refers to the processing time from the first bounced packet to the last packet on the firewall. The experimental results of the generating time is shown in Table 3 and the graph of the results is illustrated in Figure 12. The results of verifying time is shown in Table 4 and Figure 13 respectively.

From Table 3, the tree rule firewall spends more time on generating data structures than other algorithms. The time complexity will be dramatically grown up from 93.91 to 148.19 seconds while processing firewall rules between 4,000 and 5,000 – as the binary firewall also needs to sort the data. Similarly, the time generating of policy mapping also grew up from 2.92 to 5.62 seconds between executing firewall from 4,000 to 5,000 rules. Because of the majority of the consumption is spent by mapping the firewall policy to arrays. On the other hand, the generic firewall is hardly

Algorithm 2 The Policy Mapping (PMAP)

```
1: Input: DSD
2: Output: Arrays that maintain firewall policies
3: if DSD ≠ ∅ then
4:    Set Array Sx-Sy = 0
5:    Set l = 1
6:    Set max_port = p (p = 0 - 65535)
7:    Set max_level = i (i = the number of levels)
8:    Set max_src-dst_ip = k (k = 0 - 2^(32−1))
9:    while l ≤ max_level do
10:      count = Count state node of level l
11:      if l = 1 then
12:         Set i = 1
13:         S(l-1)-S(l) = Create array 1 x p
14:         while i ≤ count do
15:            T_i = Read set in S(l-1)-S(l)_i
16:            while Read (T_i) ≠ ∅ do
17:               Set S(l-1)-S(l)[T_i] = i
18:            end while
19:            i = i + 1
20:         end while
21:      end if
22:      l = l + 1
23:      while l ≤ max_level do
24:         Set i = 1
25:         S(l-1)-S(l) = Create array 1 x max_src-dst_ip x
            count
26:         while i ≤ count do
27:            T_i = Read set in S(l-1)-S(l)_i
28:            while Read (T_i) ≠ ∅ do
29:               if l = max_level then
30:                  Set S(l-1)-S(l)[1][T_i][S(l-2)-S(l-1)[i]] =
                     S(l)_count
31:               else
32:                  Set S(l-1)-S(l)[1][T_i][S(l-2)-S(l-1)[i]] = i
33:               end if
34:            end while
35:            i = i + 1
36:         end while
37:      end while
38:   end while
39: end if
40: Return Arrays S0-S1, S1-S2, S2-S3
41: End
```

changed.

In the case of matching firewall rule (in Figure 13), the consumed time of the generic firewall increased in a linear aspect ($O(n)$), the tree rule firewall is the logarithmic nature ($O(log_2 n)$) and $O(1)$ for policy mapping. Indeed, policy mapping directly accesses data in any arrays by using indexes, so the speed of accessing is very fast. From Table 4, the time of verification of the policy mapping is faster than a generic and tree rule firewall in every scenarios, and constant. Although, the policy mapping is the best of a verifying speed; however, it has a tiny overhead to access arrays by 3 times in the simulation case or 6 times in real experiment. So, the policy mapping usually consumes a time for verifying to be $O(1)$ x C, where $C = 6$ (*dst_port*:S0-S1, *dst_ip*:S1-S2 x 2, *src_ip*:S2-S3 x 2 and *pro*:S3-S4).

Algorithm 3 Matching firewall rules in arrays

```
1: Input: Array Sx-Sy, Packet_i (p_i), (i, x, y ∈ Z+ | i, y ≠ 0)
2: Output: a (accept) or d (deny)
3: while TRUE do
4:    if (dst_port = S0-S1[dst_port_pi]) != 0 then
5:       if (dst_ip = S1-S2[dst_port][dst_ip_pi]) != 0 then
6:          if (result = S2-S3[dst_ip][src_ip_pi]) != 0 then
7:             return result
8:          end if
9:       end if
10:   end if
11: end while
12: End
```

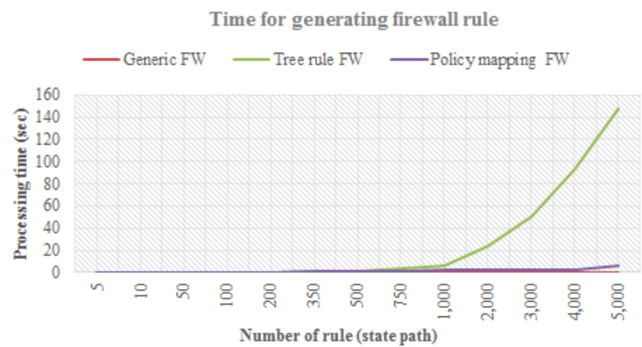

Figure 12: Time for creating firewall rule structures

Table 4: Time for verifying of each algorithm

Time for verifying firewall rule (sec)			
No.	Generic	Tree rule	Policy mapping
5	0.06771	0.05505	0.01559
10	0.11094	0.06632	0.01563
50	0.35054	0.07073	0.01563
100	0.44377	0.07949	0.01563
200	1.11984	0.07725	0.01563
350	1.75737	0.07805	0.01562
500	2.68214	0.07985	0.01563
750	4.45257	0.08125	0.01601
1,000	6.55203	0.08453	0.01501
2,000	12.66570	0.08486	0.01563
3,000	13.92543	0.09997	0.01562
4,000	22.35969	0.10676	0.01401
5,000	28.08013	0.10781	0.01801

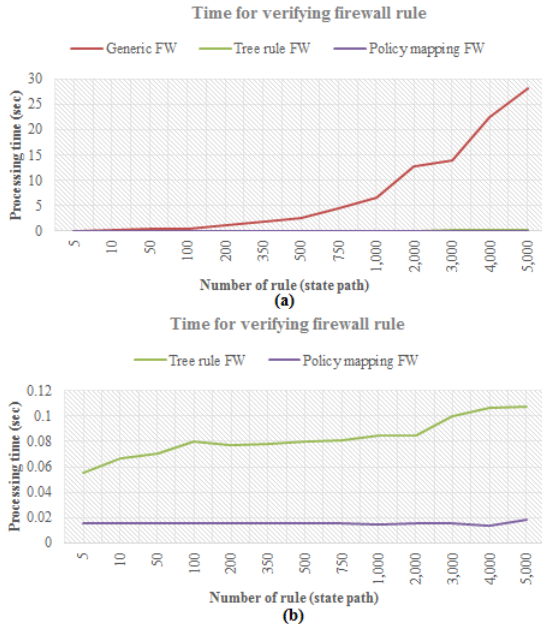

Figure 13: Time verifying for each algorithm. (a) comparing all algorithms, and (b) correlating between Tree rule and Policy mapping only

4.2 Space Complexity Analysis

We classify the memory allocation of algorithms to two groups, which are: (i) the memory space for generating firewall rule structures, and (ii) the memory for running firewall. Table 5 and Figure 14 represent the memory for generating rule structures. From the experimental results, the tree rule firewall consumed the memory space more than other algorithms. As running the tree rule firewall at the rule number 1,000; the memory usage quickly grew up around 1 GB approximately. Because the memory is used for building tree structures of rules. In contrast, the memory usage of generic firewall increased around 4.52 Kb only (at rule no. 1,000); and lightly grew up to 21.52 Kb to execute the firewall rule no. 5,000. The policy mapping continuously consumed about 131 MB (rule no. 1,000) to 655 MB (rule no. 5,000). Because the policy mapping needs more memory to build a firewall decision state diagram (DSD) and arrays. Also, the memory usage of a tree rule firewall tremendously increased to 22.6 GB while it processed the rule number 5,000.

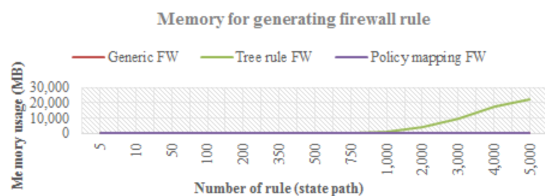

Figure 14: Memory usage for creating rule structures

In the last experiment, we estimated the memory usage of algorithms as shown in Table 6 and Figure 15 by using psutil module in Python while each algorithm was running. Policy mapping consumed the most memory, next order is the tree rule and the last is the generic firewall respectively.

Table 5: memory usage for generating rule structures

Memory for generating firewall rule (MB)			
No.	Generic	Tree rule	Policy mapping
5	0.00007	0.10	0.66
10	0.00010	0.18	1.32
50	0.00027	3.12	6.56
100	0.00046	10.53	13.11
200	0.00084	43.82	26.21
350	0.00145	126.82	45.88
500	0.00214	257.29	65.54
750	0.00312	587.03	98.31
1,000	0.00452	1,057.98	131.08
2,000	0.00828	4,348.74	262.16
3,000	0.01337	9,918.38	393.25
4,000	0.01697	17,873.39	524.33
5,000	0.02152	22,621.06	655.41

Table 6: Memory usage for running firewalls

Memory for running firewalls (MB)			
No.	Generic	Tree rule	Policy mapping
5	22.08	22.10	27.12
10	22.16	22.11	32.05
50	22.39	22.42	72.22
100	22.59	23.05	128.82
200	22.65	25.68	228.91
350	22.49	31.71	372.16
500	22.65	41.39	522.22
750	22.98	63.83	772.25
1,000	23.30	95.83	1,022.57
2,000	23.79	316.58	2,029.97
3,000	24.80	663.22	3,030.30
4,000	25.82	1,172.54	4,031.13
5,000	26.57	1,600.76	5,031.65

5 Conclusions and Future Work

The verification techniques of firewall rule are classified to three major groups: the sequence, tree and hashing. In case of a sequential approach, which is a primitive verifying technique, it is easy to understand and implement, and consumes a small memory space. However, time complexity of a sequence is very slow, that is $O(n)$. In contrast, a tree approach is difficult to understand and implement, and it consumes a large memory space – though it has an excellent processing speed ($O(log_2 n)$). The best speed to verify data is the hashing approach ($O(1)$), but it encounters a trouble of key duplication while performing on enormous data. Unfortunately, it can not satisfactorily be applied to the firewall rule because rules are huge ($\approx 2^{104-bit}$). Another disadvantage of hashing functions

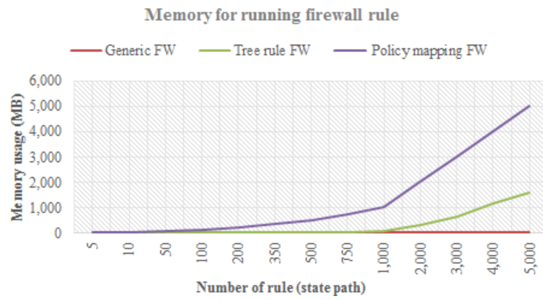

Figure 15: Memory for running firewalls

is that they consume a huge memory space to maintain data.

In this paper, we propose a new algorithm to speed up firewall rule verification, called the policy mapping (PMAP). The distinctive points of the algorithm are: (i) it is as fast as hashing approaches, (ii) it can perform without duplicate keys, (iii) it reasonably consumes the memory space and (iv) it is easy to understand and implement. Our experimental results show that the speed of the policy mapping is faster than sequential and tree rule firewalls, and it also consumes a suitable memory space. Moreover, the policy mapping is as fast as IPSet [12] (top of the high-speed and popular open source firewall today). However, the policy mapping is not limited to the IP network class management like IPSet which is only available for the IP class C and B. Moreover, IPSet always requires to rearrange rules before deploying to the firewall engine.

In the future work, we will optimize the memory size of the PMAP to be better.

References

[1] M. G. Acharya, H. B. Gouda, "Projection and division: Linear-space verification of firewalls," in *Proceedings of The International Conference on Distributed Computing Systems (ICDCS'10)*, pp. 176–180, Genova, June 2010.

[2] A. Blyth, "An architecture for an XML enabled firewall," *International Journal of Network Security*, vol. 8, no. 1, pp. 31–36, 2009.

[3] M. Chapple, A. Striegel, "An analysis of firewall rulebase (mis) management practices," *ISSA: The Global Voice of Information Security*, 2009. http://mike.chapple.org/an-analysis-of-firewall-rulebase-mismanagement-practices/

[4] P. G. Clark and A. Agah, "Firewall policy diagram: Structures for firewall behavior comprehension," *International Journal of Network Security*, vol. 17, no. 2, pp. 150–159, 2013.

[5] M. G. Gouda and A. X. Liu, "Structured firewall design," *Computer Networks*, vol. 51, pp. 1106–1120, 2007.

[6] H. Hamed, A. El-Atawy, and E. Al-Shaer, "On dynamic optimization of packet matching in high-speed firewalls," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1817–1830, 2006.

[7] IETF, *Internet Official Protocol Standards*, 2008. `http://tools.ietf.org/html/rfc5000`

[8] S. Khummanee, A. Khumseela, and S. Puangpronpitag, "Towards a new design of firewall: Anomaly elimination and fast verifying of firewall rules," in *Proceedings of The Computer Science and Software Engineering (JCSSE'13)*, pp. 93–98, Abu Dhabi, May 2013.

[9] A. X. Liu, "Formal verification of firewall policies," in *IEEE International Conference on Communications*, pp. 1494–1498, Beijing, May 2008.

[10] A. X. Liu and M. G. Gouda, "Diverse firewall design," *IEEE Transaction on Parallel and Distributed Systems*, vol. 19, no. 9, pp. 1237–1251, 2008.

[11] A. X. Liu and M.G. Gouda, "Firewall policy queries," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 6, pp. 766–777, 2009.

[12] Netfilter, *IP Set*, Aug. 24, 2015. `http://ipset.netfilter.org/index.html`

[13] Netfilter, *IPTables*, June 25, 2015. `http://ipset.netfilter.org/iptables.man.html`

[14] J. Touch, E. Lear, et al., *Service Name and Transport Protocol Port Number Registry*, RFC 6335, Aug. 6, 2015. `http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`

[15] P. N. Zhiyuan, X. Tan, T. C. He, "Improving cloud network security using the tree-rule firewall," *Future Generation Computer Systems*, vol. 30, pp. 116–126, 2013.

Suchart Khummanee is a Ph.D student at Khon Kaen University, Khon Kaen, Thailand. His research is in the field of Computer Networks and Security.

Kitt Tientanopajai is a full Professor of computer engineering with Khon Kaen University, Khon Kaen, Thailand. His research interests focus on Free/Open Source Software, Information Security, Quality of Service Routing, Computer Networks and Educational Technology.