IP Packing Technique for High-speed Firewall Rule Verification

Suchart Khummanee

Department of Computer Science, Faculty of Informatics, Mahasarakham University, Thailand
suchart.k@msu.ac.th

Abstract

A network bottleneck is often caused by firewalls installed between network gateways. As a result, the overall performance of networks is significantly dropped. The following solution to resolve such the problem can be achieved by increasing the speed of firewall rule verification. Nowadays, there is an open-source matching framework which is the fastest of rule verification, namely IPSets. It can verify a number of firewall rules against huge packets with O(1) worst case access time. However, IPSets still displays several drawbacks of usability such as rule management, subnet IP address, rule conflicts, and memory usage. This paper proposes a novel firewall structure that can resolve all drawbacks of IPSets, and obtains the optimal speed of firewall rule verification at O(1) of access time, called IPack. According to IPack implementation, the paper applies the sparse matrix to be data structures to maintain firewall rules, the Path Selection Diagram (PSD) to eliminate rule conflicts and IP packing technique to reduce the size of memory space. The experimental results show that IPSets drawbacks can be solved by IPack. Especially, the size of memory space is reduced from $O(2^n)$ to be O(n) with the same optimal access time and the speed of IPack is still equal to IPSets.

Keywords: Firewall, High-speed firewall, Firewall rule matching, IP packing, Path selection diagram

1 Introduction

Firewall is a basic network security tool to protect suspicious packets and unauthorized access between the trusted and untrusted network zone. Generally, installing on the network gateway between private networks (Trusted) and the public network (Untrusted or Internet), a firewall verifies of all packets at inbound and outbound interfaces depends on security rules. The installed firewall on such location, theoretically there are several advantages. For example, it is the best place to filter all the network traffic; likewise, the other is to save the cost of purchasing firewalls because firewalls can handle whole traffic at the only single network gateway. In contrast, it is a critical point which may result a single point of failure (SPOF) and a bottleneck of networks. If SPOF occurs in networks, the entire networks will be stopped from working suddenly. One solution to SPOF is to customize firewalls to be the distributed system [1]. The bottleneck refers to slow communication speeds and limits user's efficiency on the networks [2]. Solving of the bottleneck of firewalls, increasing the speed of rule verification is a good solution to deal with it. In this paper focuses on the bottleneck networks by improving the speed of firewall rule verification.

Due to the bottleneck problem caused by firewalls, in previous years, several researchers have tried to improve the speed of firewall rule verification which can be classified by the speed into three groups as follows. General firewalls, applied to the sequential searching algorithm or some tree structures, are placed in the first group such as IPTables [3], Based-Service grouping [4-5] and the linear time approach [6]. The time complexity of rule verifying of this group is quite slow, and it is O(n). The firewalls in the second group are widely designed by effective tree structures, hence rule verifying time of this is faster than the first group; that is O(log(n)) like [7-11], etc. Currently the last group is an open source framework utility working with IPTables [3] and Netfilter [12] to be as the network firewall-namely, IPSets [13]. It can verify any packet against firewall rules with O(1) worst case access time by using the perfect hashing technique. Although IPSets is the fastest framework to verify the firewall rule nowadays, it has several drawbacks in terms of usage. The first obstacle is to create the unique keys for a perfect hashing function. IPSets does not automatically build unique keys. This responsibility is pushed to an administrator who is an expert about firewall rule management. IPSets offers a variety of key generation formats such as hash:IP:port, hash: net, etc. For example, the hash:IP:port means combining between an IP address and a port to be a key like "1921681180" (Combining between IP 192.168.1.1 and port number 80 by cutting off dots). Another example is that hash:IP:port:net is a combination between an IP address, a port number and a subnet network. For example, 172.16.1.5:80:192. 16.1.0/30 can be enumerated to four unique keys like

*Corresponding Author: Suchart Khummanee; E-mail: suchart.k@msu.ac.th
DOI: 10.3966/160792642019102006006

"17216158019216810", "17216158019216811", "17216158019216812", and "17216158019216813" (Subnet network 192.168.1.0/30 ranging from 192.168.1.0-192.168.1.3). Selecting a key type of IPSets depends on discretion and suitability requirements of an administrator directly.

The second barrier to IPSets deployment is to handle the range of large subnets. For example, it does not recommend to apply to IP addresses of the A classes because the number of generated keys may be larger than the available memory space (Overflowed memory). A subnet 10.0.0.0/8, for instance, can be generated to be a set of unique keys ranging from "10000", "100001", "100002", ..., "10255255253", "10255255254", "10255255255" (~= 16,777,216 keys).

IPSets is built in order to improve the speed of firewall rule verification or packet matching of IPTables only. Other functions of the firewall are handled by IPTables, such as the packet decision, packet filtering, packet forwarding, the network address translation (NAT), etc. Thus, IPSets cannot independently execute by itself without IPTables, which is the third drawback of the IPSets framework. As the selection of key generation methods for the perfect hashing in IPSets also effects on an amount of the active memory. IPSets, therefore, suggests administrators to reduce the number of key sets by setting the size of IP class to a small size such as class C or B instead of IP class A. While IPSets is executing and its own memory is running out, it will always allocate new memory size as 2^n bytes for each time. If the default memory size of IPSets is 2^{10} bytes [13], it will be allocated as 2^{11}, 2^{12}, ..., 2^n bytes respectively. Thus, the space complexity is $O(2^n)$ which is quite large. This is the fourth restriction on IPSets usage.

The last IPSets obstacle is related to firewall rule anomalies or conflicts. Although IPTables cooperates with IPSets to act as the fastest network firewall today (The state of the art of the high-speed firewall rule verification), both still do not support the firewall rule anomaly detection, which is a highly important function of network firewalls. The firewall rule anomaly means two or more different filtering rules that match against the same packet [14]. As an anomaly occurs in a firewall running on a network, it results in low overall network performance, which is a major cause of the network bottleneck.

In this paper, firstly, aims to eliminate the IPSets drawbacks and its speed of rule verification is being still as fast as IPSets, which is O(1) worst case access time. Second, develops a new whole firewall system running on the Linux operating system for kernel version 2.6 or higher. Third, proposes two-dimensional (2D) sparse matrix structures to store firewall rules, and to handle all IP address classes. Forth, presents the IP packing technique which compacts non-zero items from 2D sparse matrices to one-dimensional (1D) arrays for reducing the memory space to O(n). Finally, eradicates rule anomalies by using the path selection diagram algorithm (PSD) applying from the firewall decision diagram [15].

2 Background and Related Work

2.1 Basic Firewall, Rule Definition, and Rule Anomaly

Currently, network infrastructures are widely designed by gigabit networks because they can support high-speed traffic from a large number of users. Firewalls working on such networks also require high performance. The main function of firewalls is to restrict accessing resources on the networks by rules, which will be set properly. For example, firewall rules basically allow all requests from any client in private networks (Trusted networks) to any resource of servers in public networks (Untrusted networks); in contrast, any server cannot access to clients. A common firewall rule consists of two essential parts. The first one is a predicate, and the other is an action. The predicate is combined with six conditions: Source IP address (*SIP*), Destination IP address (*DIP*), Source Port (*SP*), Destination Port (*DP*), Protocol (*Pro*), and Interface (*Int*). The action part (*Act*) will be set either to pass or to drop, if all conditions in the predicate are true. Given r_n to be any firewall rule, and r_n is subset of R ($r_n \subset$ R) where R denotes all rules. In addition, $r_{n\{...\}}$ represents conditions of each rule. So that, $r_{n=\{SIP, DIP, SP, DP, Pro, Int\}}$ is any source and destination IP address, any source and destination port, any protocol, an arriving-packet interface of r_n respectively. Let p_k is any incoming or outgoing packet over the firewall; then $p_{k=\{SIP, DIP, SP, DP, Pro\}}$ is a source and destination IP address, a source and destination port, and a protocol of p_k respectively. Rule r_3 in Table 1, for example, the firewall will allow private packets across to other public networks if $p_{k\{SIP\}} \in r_{3\{SIP\}}$, $p_{k\{DIP\}} \in r_{3\{DIP\}}$, $p_{k\{SP\}} \in r_{3\{SP\}}$, $p_{k\{DP\}} \in r_{3\{DP\}}$, and $p_{k\{Pro\}} \in r_{3\{Pro\}}$ are true by processing over the interface A. Thus, $p_{k\{SIP\}} \in$ {192.168.1.0, ..., 192.168.1.255}, $p_{k\{DIP\}} \in$ {0.0.0.0, ..., 255.255.255.255}, $p_{k\{SP,DP\}} \in$ {0, 1, ..., 65,535}, and $p_{k\{Pro\}} \in$ {0, 1, ..., 255} are true. Rule r_4 exactly drops all public packets that will pass into the private networks via the interface B. The final rule (r_5) always blocks every packet from any to any network.

Table 1. Firewall rule examples

No.	*SIP*	*DIP*	*SP*	*DP*	*Pro*	*Int*	*Act*
r_1	192.168.1.0/24	*		137-445	*	A	*drop*
r_2	192.168.1.0/24	200.30.5.100	*	113	UDP	A	*drop*
r_3	192.168.1.0/24	*	*	*	*	A	*pass*
r_4	*	192.168.1.0/24	*	*	*	B	*drop*
r_5	*	*	*	*	*	*	*drop*

Note. *(*SIP, DIP*) = $(0\text{-}2^{32})\text{-}1$, *(*SP, DP*) = $(0\text{-}2^{16})\text{-}1$, *(*Pro*) = TCP or UDP.

Firewall rule anomalies often arise from unclear understanding of rule definitions. For example, rule r_2 and r_3 are overlapping for all condition fields with different actions (Rule r_2 = *drop*, r_3 = *pass*). Therefore, we can say that the firewall rule anomalies arise from "*The conditions appearing in terms of the predicate of two or more rules overlap, but there are the different actions*". Al-Shaer and Hamed [16] first defined anomaly patterns for firewall rules in 2004. They defined five types of anomalies, which are shadowing, correlation, generalization, redundancy, and irrelevancy anomaly. Later, many researchers endeavored to resolve rule anomaly problems as follows. Khummanee et al. [8] proposed the single domain decision approach to reduce the root cause of anomalies, by the key contribution of this method required acceptable or unacceptable rules only. Liu et al. [17] presented the systematic approach to compress firewall rules, and they claimed that this technique could reduce the number of firewalls more than 50%; however, firewall rule anomalies remained especially the redundancy of the rules. Applying log analysis and dynamic rule re-ordering to improve the efficiency of the anomaly management is contributed by Lubna et al. [18], and adapting the Apriori algorithm to detect anomaly attacks on firewall rules [19]. Although the researchers can resolve some anomaly problems, they cannot eliminate all anomalies of firewall rules completely. Interestingly, there is research that can solve the anomaly problems effectively by using the firewall decision diagram (FDD) contributed by Liu [20]. FDD can eliminate all anomalies, but it must process rules in order from the top to the bottom of the rules only.

2.2 Sparse Matrix and Packing Techniques

A sparse matrix is a matrix where most elements are zero; in other words, it is a matrix that contains very few nonzero elements as following equation (1). Let S be a matrix where $S \in Z^{col * row}$ by col and row $\in N_0$, thus S is a sparse matrix since the number of non-zero elements of S is O(min{col, row}) [21]. Most of the sparse matrices applied to science and engineering are two-dimensional. For example, S is two-dimension sparse matrix consisting of four columns (col = 4) and four rows (row = 4). The total memory space used to store all elements of S is col * row * k, where k is the memory size (Byte) to keep individual elements. The total memory space of S to maintain signed integers is 4 * 4 * 2 (bytes) = 32 bytes in C programming language.

$$S = \begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 \\ 6 & 4 & 3 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix} \in Z^{col=4*row=4} \quad (1)$$

To reduce a total memory size of a sparse matrix can be done by collecting only non-zero elements, and the technique is called the sparse matrix packing. Packing techniques often require complicated data structures to store non-zero elements, and especially it must be able to retrieve non-zero elements, stored in such structures, to the original matrix correctly [22]. Generally, packing techniques are classified by the storage space (Point or block) into two categories [23]: the first one is the point based storage formats, which maintains a single non-zero element with only one element of a matrix space such as the coordinate storage (COO), compressed row storage (CRS) and so on, and the other is the block-based storage formats, which maintains a block of non-zero elements of the matrix such as the block coordinate storage (BCOO), block compressed row storage (BCRS) and so forth. In this paper, the concept of BCRS is only presented because it performed well and was applied to the packing methodology. BCRS is applied from CRS format as shown in Figure 1. It can save the cost of memory size effectively if each sub-block is large and non-zero elements in each sub-block are dense. In the Figure 1, BCRS requires three arrays including the block of non-zero elements (BNZ), column index (col_ind), and row pointer (row_ptr). BNZ contains square dense sub-blocks of non-zero elements. For instance, the dimension of each sub-block is 2 * 2, thus the sub-blocks are stored {{{5, 0}, {0, 9}}, {{6, 4}, {0, 1}}, {{3, 0}, {0, 2}}} by a sequence of the sub-block address number. The col_ind indicates the column of each sub-block; for instance, the sub-blocks of {{5, 0}, {0, 9}} and {{6, 4}, {0, 1}} are in the column 0, and {{3, 0}, {0, 2}} is in the column 1. Last, the row_ptr specifies the first address of sub-blocks in each row; for example, the first sub-block address in the row no. 1 of BCRS is 1. BCRS can save the memory space more than CRS (CRS = 38, BCRS = 36 bytes).

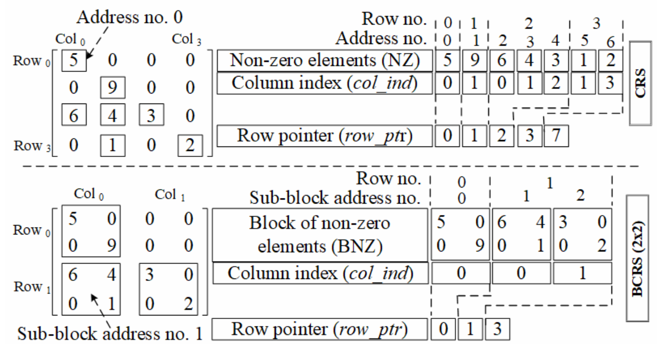

Figure 1. CRS and BCRS storage format

2.3 Hash and Perfect Hashing Techniques

A hash function is used for mapping (Hashing) data of capricious size to fixed size. Returned values from a function are called hash values. A data structure used to map keys to values is called a hash table, which

usually implements an associative array abstract data type [24]. A hash function computes an index from the key into an array of buckets. Indeed, the hash function will attempt to assign a unique free slot in the bucket to any key. However, it might generate the same index for more than one key, called the hashing collision problem. The collision is the big problem of hash functions, and it must be handled or resolved by a systematic approach in some way such as separate chaining, open addressing, robin hood hashing and etc., [25]. The effectiveness of a good hashing depends on a good hash function, therefore choosing a hash function is very important. The basic hashing uses a single or fixed hash function which is not the efficacy. To improve the performance of a basic hashing, it can be done by choosing a randomized function from a family of hash functions instead of the fixed hash function. This technique, called the universal hashing, can significantly reduce the number of collisions. Even though the universal hashing is highly effective, it still cannot resolve the collisions. In fact, we can make the number of collided keys of the universal hashing to be zero if we have known all keys in advance or keys are static. Such technique is called the perfect hashing. The perfect hashing guarantees whether it can build a hash table with no collisions.

2.4 IPSets

2.4.1 IPTables and IPSets Framework

Although IPSets [13] is just an extension framework of the IPTables firewall [3], it has improved the speed of rule verification of IPTables from O(n) to O(1) [26]. That is, IPTables has become the best of high-speed firewalls now. The IPTables and IPSets framework is shown in Figure 2.

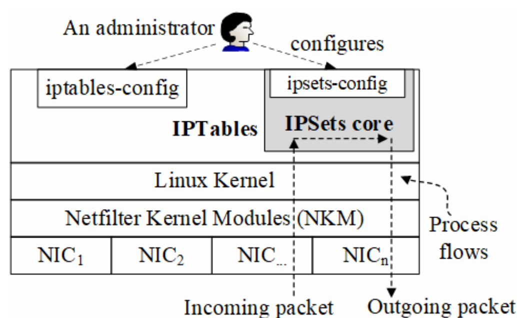

Figure 2. IPTables and IPSets framework on Linux

Supposing that, there is any incoming packet arriving at an inbound interface (Network interface: NIC) of the IPTables firewall. The packet is passed up through the Netfilter Kernel Modules (NKM) layer to filter only required packet depending on the network policy. The network policy is configured by an administrator with the iptables-config utility on IPTables layer. In case that IPSets is enabled, firewall rules will be configured by the ipsets-config, a command-line user interface, instead of the iptables-config. The packet, forwarding from NKM to IPTables layer, will be handled by Linux kernel. However, first, an admin needs to enable IP forwarding feature of Linux kernel by issuing the command as "*sysctl-w net.ipv4.ip forward = 1*". After the packet arriving to IPTables, it will be thrown to IPSets core for verifying against rules in the next step. Rules of IPSets can be configured in several ways, as discussed in Introduction section, for example, hash:IP:port, hash: net, etc. The role of IPSets core matches the packet only, other functions will be handled by IPTables such as IP forwarding, logging, chaining and so on. After the packet validation process has been completed, the packet will be decided either to pass or to drop (Action). If the action is to drop, IPTables will throw the packet away immediately. Otherwise, it will forward the packet to a destination next. The packet, which is forwarded to a destination network over an outbound interface, called an outgoing packet as shown in Figure 2.

2.4.2 IPSets Workflows

As mentioned earlier, the perfect hashing has been applied to IPSets in order to speed up rule verification. In this section will explain the detailed IPSets workflows as shown in Figure 3. The first step is creating a set of IPSets rules (Circle number 1). Supposing the type of rule chosen is hash:IP:port, such as DIP ranging from 192.168.1.0 to 192.168.1.9 (10 IP addresses), *DP* = 80 and Act = *accept.* The second step (Circle number 2) is to generate a set of keys by the Cartesian Product between *DIP* and *DP* from the rule set in step 1, for example, {'1921681080', '1921681180', '...', '1921681980'}. The number of keys in this step is 11 keys. The generated keys from step 2 are hashed and stored in the bucket in order to group duplicate keys as shown in the 3rd step. The algorithm used to hash the keys is FNV [27] because it is a fast hash processing and very low collision rate as shown in Algorithm 1. Inputs of FNV require a hash function from the function family (d) and a key that will be hashed. The output is usually called a digest (p) which is always the fixed length, even though the keys are different lengths. To compute the location of a key in a bucket, the modulus (mod = %) is used between a digest and the table size. If the locations of digests are collided, the perfect hashing will be stored at the same keys in the same bucket in a stacked manner (Step 3 and Algorithm 2). For example, the key_i = "1921681880" and key_j = "1921681980" are the collision while they are hashing by the function H(d, key_i) and H(d, key_j) = 2. The d = 0 means the first function in the function family, and d will be incremented one by one from 0 to n when key collisions occur (d = 1 = 2^{sd} function, d = 2 = 3rd function, ..., d = (n-1) = n^{th} function).

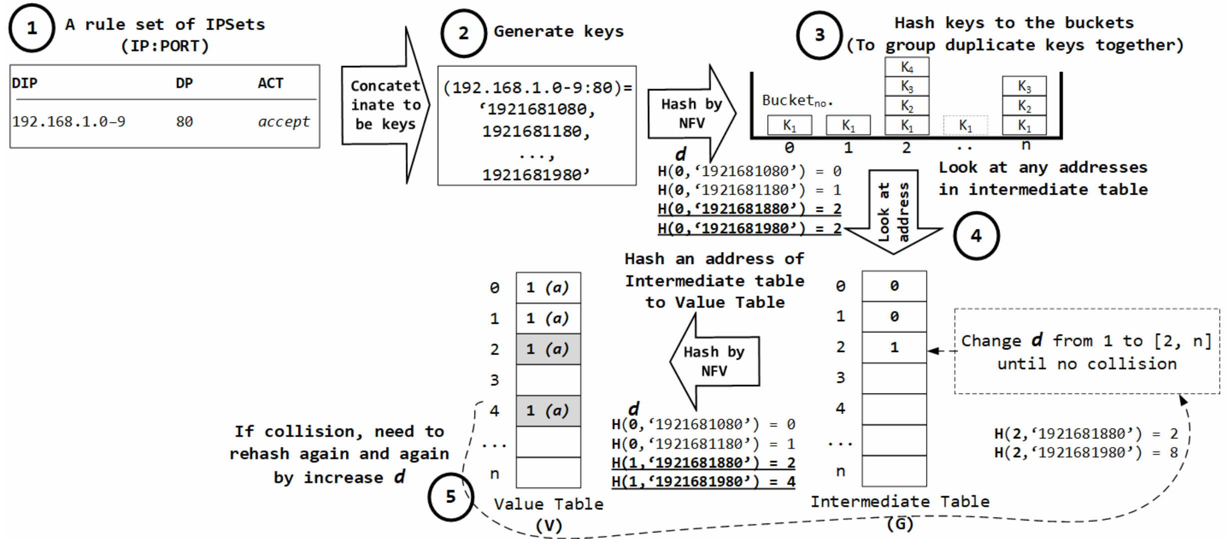

Figure 3. IPSets workflows with the perfect hashing

Algorithm 1. The FNV hash algorithm

1. **Input:** *d*, a key
2. **Output:** *p* (a hashed key)
3. **if** Input has both *d* and a key **then**
4.   **if** *d* == 0 **then**
5.   *d* ← 0x01000193
6.   **end if**
7.   while c ← read a char from a key until NULL do
8.   *d* ← ((*d* * 0x01000193) ^ ord(c)) & 0xffffffff
9.   **end while**
10. **Else**
11.   **return** NULL # *any input error*
12. **end if**
13. return *p* ← *d*
14. **End**

Algorithm 2. Perfect hashing of IPSets

1. **Input:** dictionary (dict) of key and value ('key':value)
2. **Output:** G, V (G = intermediate, V = values table)
3. set size >> |dict|
4. # *put all keys to buckets*
5. create bucket[size][], G[size], V[size]
6. **while** key_i ← read dict until NULL **do**
7.   *p* ← FNV-hash (0, key_i) mod size
8.   bucket [*p*].append(key_i)
9. **end while**
10. # *put all unique keys from buckets to G, V table*
11. **while** key ← read bucket[*d*][*] until NULL **do**
12.   **if** |key| == 1 **then**
13.   put 0 ← G[*d*] ($d \in N_0$)
14.   put value from dict(key) → V[*d*]
15.   **end if**
16. **end while**
17. # *put all collided keys from buckets to G, V table*
18. **while** keys ← read bucket[*d*][*] until NULL **do**
19.   **if** |keys| > 1 **then**
20.   d = 1
21.   **while** key ← read keys until NULL **do**
22.   p ← FNV-hash(*d*, key) mod size
23.   **if** put *d* → G[*p*] then collision **then**
24.   *p* ← rehash(*d*++, key) until no collision
25.   **end if**
26.   V[*p*] ← read value from dict(key), G[*p*] ← *d*
27.   **end while**
28.   **end if**
29. **end while**
30. **End**

After all keys in step 2 have been successfully hashed by the function H(0, key_i) into the buckets, IPSets will hash all keys in each bucket to the intermediate (G) and value table (V) again as illustrated in step 4. If any bucket holds only one key, the key does not collide like the bucket number 0 and 1 in step 3. In the opposite case, if any bucket holds more than one key, the keys in these buckets are collisions as the bucket number 2. Step 4, a key in each bucket (No collision) will be hashed by the function H(0, key_i) again. The hash function 0 will be put in the table G, and the action will be placed in the table V. For example, the bucket 0 keeps the key '1921681080', which is hashed by the function 0, the algorithm will put 0 at the address 0 in G and place 1 (Action = *accept*) at the address 0 in V. For another example, keys in the bucket 2 (Collisions) will be hashed by H(1, key_i) such as H(1, "1921681880") = 2 and H (1, "1921681980") = 4. Fortunately, in this example, both keys are not the collision, so the function number 1 (1) will be put at the address 2 in G, and the action (*accept* = 1) will be put at the address 2 (for "1921681880") and 4 ("1921681980") in the table V. Noticeably, if both key '1921681880' and "1921681980" in the bucket 2 are hashed and then collided. They can be solved by increasing the function number (*d*) from 1 to 2, ..., *n* and re-hashing until there is no collision as illustrated in step 5.

2.4.3 IPSets Matching Process

The packet matching (Verifying) of IPSets is shown in Figure 4 and Algorithm 3. Supposing that an incoming packet p_i consisting of DIP = 192.168.1.9 and DP = 80 is arriving to the IPSets, the p_i will be formatted to be a key as “1921681980” (Step 1). After that, the key will be hashed by H (0, “1921681980”) = 2. The number 2 indicates the address that holds the hash function number 1 in G table (Step 3). The key “1921681980” will be hashed again by H(1, “1921681980”) = 4 in V table (Step 4). This address (Number 4) keeps the accepted action (*accept* = 1) for processing this packet. That is, this packet can pass to any destination address (Step 5).

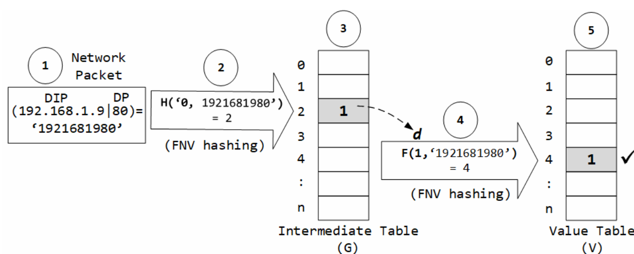

Figure 4. Packet matching procedure of IPSets

Algorithm 3. Look up the value in hash table G, V

1. **Input:** G, V and a key
2. **Output:** a value
3. d = G[FNV-hash(0, key) mod len(G)]
4. **if** d == 0 **then**
5. **return** V[FNV-hash(0, key) mod len(G)]
6. **else**
7. **if** d > 0 **then**
8. **return** V[FNV-hash(d, key) mod len(G)]
9. **else**
10. **print** ”Not found”
11. **end if**
12. **end if**
13. **End**

2.4.4 IPSets Memory Consumption

According to IPSets memory management, the default memory of G and V table is 1,024 bytes [13]; however, the system administrator can make changes at any time. In the situation where G and V table are not sufficient for all key generation process, both will be automatically increased by 2^n such as 2^{11}, 2^{12}, ..., 2^n. Thus, IPSets, consuming the memory usage, is $O(2^n)$.

2.4.5 IPSets Configuration

There are two steps to configure IPSets to act as the firewall in conjunction with IPTables:

(1) Declaring a set of firewall rule is based on an option type, such as hash:IP, hash: net, and so on. For this example, this paper will only show hash: net as follows:

\# create a set of IPSets rules named myset as hash: net **ipset** create myset hash: net

\# *add the myset subnet network addresses to IPSets engine* **ipset** add myset 192.168.1.0/24 **ipset** add myset 192.168.2.0/24:

(2) Binding the set of IPSets rules to IPTables is: # *configure IPTables to drop src addresses* **iptables**-I INPUT-m set-match-set myset src-j DROP.

3 Key Contributions

Nowadays, IPSets is the fastest open source matching framework for IPTables. It is able to verify any rule against any packet by O(1). However, it still has several drawbacks. In this paper, a new structured firewall is proposed. The new firewall can overcome the drawback of IPSets, and can obtain the same optimal speed as performed by IPSets, which is O(1). This paper consolidates six main contributions as follows:

(1) The paper develops a new structured firewall by C language on Linux kernel, called IPack. It is the full firewall, not just an extension like IPSets.

(2) IPack improves the memory consumption of IPSets from $O(2^n)$ to O(n) by applying the sparse matrix structure and IP packing algorithm.

(3) It does not declare rule types like IPSets, it can declare firewall rules in the general pattern freely.

(4) It can detect and correct rule anomalies by using the Path Section Diagram (PSD), unlike IPSets does not.

(5) It can handle all IP address classes (A-D) but IPSets is not supported the large class such as the A class.

(6) IPSets requires a lot of skills to design firewall rules, but IPack does not.

4 IP Packing Design

The design process of IPack has five phases shown in Figure 5. Each design phase can resolve the IPSets drawbacks respectively. The details of each phase are follows:

4.1 Phase 1: Creating Rule-Based Firewall

Rule-based firewall is now a popular rule fashion such as IPTables, Cisco ASA, Windows defender, and so on. A rule base is a set of rules that declares what packets can be passed or dropped. Rule base normally works on a top-down approach in which the first rule in the rule list is executed first. If the traffic is allowed by the first rule, the subsequence rules will never be executed. Rule base typically has the format of {Source, Destination, Service, Interface, Action}, such

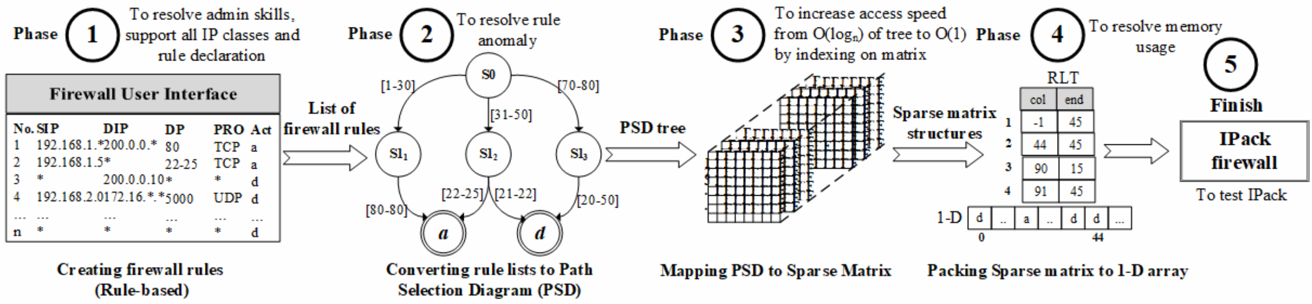

Figure 5. IPack design process

as {192.168.1.0/24, 200.150.10.0/28, TCP, eth0, DROP}. The advantages of rule base: admins can create rules freely with basic firewall knowledge (No need to be expert), and rules are arranged in order to make them easy to understand. Therefore, this paper still chooses the rule-based interface to make firewall rules for the phase 1 in Figure 5. In order to explain the IPack design, this paper selects the five elements of the rule as *SIP*, *DIP*, *DP*, *Pro*, and *Act*, shown in Table 2 because the elements are unique to make a decision path for PSD in the next phase.

Table 2. Simple rules to demonstrate IPack design

No.	*SIP*	*DIP*	*DP*	*Pro*	*Act*
r_1	192.168.1.0/28	*	80	*	*a*
r_2	192.168.1.16/28	*	25-30	*	*a*
r_3	192.168.1.0/24	*	60-90	*	*d*

In Table 2, there are three rules. The 1st rule (r_1) allows (*a*) source IP addresses (*SIP*) ranging from 192.168.1.0-15 (16 IP addresses) to any destination address (*DIP* = *), a destination port number 80 (*DP*), and any protocol (*Pro* = * = *TCP* or *UDP*). The rule (r_2) accepts *SIP* addresses from 192.168.1.16-.31 (16 IPs), *DIP* = any, *DP* = 25-30, and *Pro* = any. The last rule (r_3) drops (*d*) *SIP* from 192.168.1.0-.255 (256 IPs), both *DIP* and *Pro* = any and *DP* = 60-90. Noticeably, rule r_3 conflicts against r_1 and r_2 (Overlapping between *SIP* and *DP*, and different action), but rule r_1 does not conflict with rule r_2.

The phase 1's conclusions. Rules are created according to user requirements independently, and not assigned any group type like IPSets (To resolve the admin skills, IP classes, and difficult rule declaration).

4.2 Phase 2: Converting Rule-Based to PSD Tree

The rules from the phase 1 will be converted to Path Selection Diagram (PSD) in order to eliminate conflict problems. The key principle of PSD is to merge all duplicated rules to unique rules. The second phase begins by reading the rules from the first phase in the sequence as shown in Figure 5. For example, the r_1 in Table 2 is read first. For creating the PSD tree structure, the root node is constructed on the tree first as illustrated in step 1 of Figure 6. After the root node is created on PSD tree, PSD algorithm inserts the ports of each rule by a sequence to PSD. For example, the port number 80 ([80]) of the r_1 is inserted to the first child node on PSD-named, $N_{\{i,j\}}$ (i = level name, j = node number in each level). So, $N_{\{i=1,j=1\}}$ means the first node in the *DP* level (Step 2 in Figure 6). Next, the algorithm connects the root node to each child node in *DP* level by building a link state-named, $L_{\{k,l\}}$ (k = level name, l = link number in each level). The $L_{\{k=1,l=1\}}$, for example, is to link between the root node to the first child node ($N_{\{i=1,j=1\}}$) in *DP* level. Upon $r_{1\{DP\}}$ is already created on the tree, then the ports of $r_{2\{DP\}}$ ([25, 30]) will be constructed on the tree in step 3. $r_{2\{DP\}} \not\subset r_{1\{DP\}}$, so PSD algorithm will create a new node named $N_{\{1,2\}}$ and a new link named $L_{\{1,2\}}$ from the root node to $N_{\{1,2\}}$ on the tree. The last step in this level is creating all ports of r_3. The $r_{3\{DP\}}$ is the superset of $r_{1\{DP\}}$ ($r_{3\{DP\}} \supseteq r_{1\{DP\}}$), so $r_{3\{DP\}}$-$r_{1\{DP\}}$ = [60, 90]-[80] = [60, 79] and [81, 90] respectively. New node $N_{\{1,3\}}$ and $N_{\{1,4\}}$ are created on the tree in step 4. The port number 80 of r_3 is merged to $N_{\{1,1\}}$ because it is duplicated with $r_{1\{DP\}}$. The last process in this *DP* level is to link between the root to $N_{\{1,3\}}$ and $N_{\{1,4\}}$ by $L_{\{1,3\}}$ and $L_{\{1,4\}}$ respectively. After all, the nodes in the *DP* level are created successfully; the next process is to construct all the destination IP addresses (*DIP*) of rules in the *DIP* level on the tree. The first operation at this level is constructing *DIP* of r_1. $r_{1\{DIP\}}$ is created as node $N_{\{2,1\}}$ ranging from 0.0.0.0 to 255.255.255.255(*), and the $L_{\{2,1\}}$ links between $N_{\{1,1\}}$ and $N_{\{2,1\}}$ as illustrated in step 5 of Figure 7. While the $r_{2\{DIP\}}$ can be inserted to the tree as the node $N_{\{2,2\}}$ immediately because of $r_{2\{DP\}} \not\subset r_{1\{DP\}}$ (Step 6). The port numbers of r_3 is the superset of $r_{1\{DP\}}$, so inserting $r_{3\{DIP\}}$ to the tree has to consider with three paths: the path of $L_{\{1,1\}}$ ($N_{\{2,1\}}$), $L_{\{1,3\}}$ ($N_{\{2,3\}}$), and $L_{\{1,4\}}$ ($N_{\{2,4\}}$) in the step 7 of Figure 7. Similar to inserting *DIP*, source IP addresses of r_1 ranging from 192.168.1.0 to 192.168.1.15 (/28) are inserted as the node $N_{\{3,1\}}$ in the first path of PSD tree (Step 8 of Figure 8). After that, PSD algorithm establishes a link $L_{\{3,1\}}$ from $N_{\{2,1\}}$ at *DIP* level to $N_{\{3,1\}}$ at *SIP* level. The $r_{2\{DP\}}$ is not the subset of any rule, so $r_{2\{SIP\}}$ will be inserted into $N_{\{3,2\}}$ in the second path on the tree without any action (Step 9).

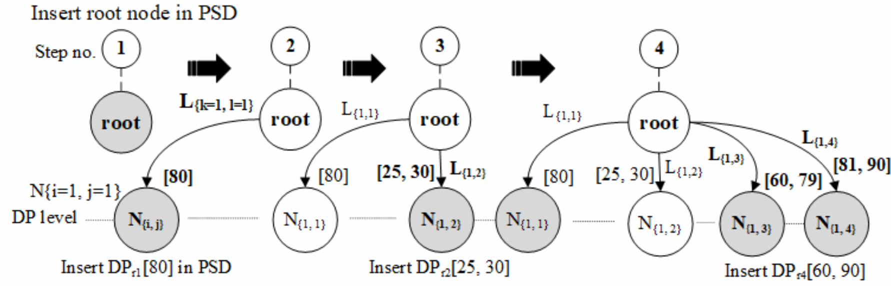

Figure 6. Inserting all ports from rules to PSD (DP level)

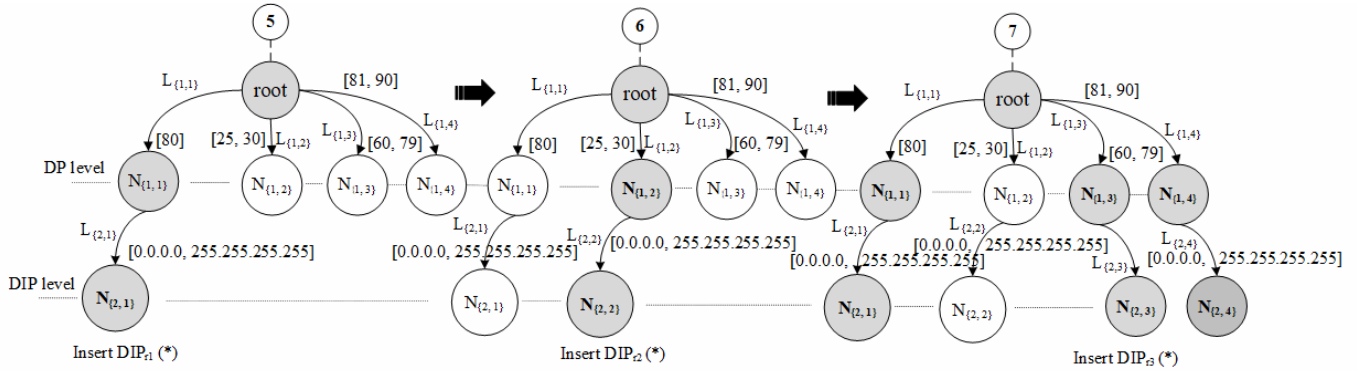

Figure 7. Inserting all destination IP addresses from the rules to PSD (*DIP* level)

After that, PSD algorithm establishes a link $L_{\{3,1\}}$ from $N_{\{2,1\}}$ at *DIP* level to $N_{\{3,1\}}$ at *SIP* level. The $r_{2\{DP\}}$ is not the subset of any rule, so $r_{2\{SIP\}}$ will be inserted into $N_{\{3,2\}}$ in the second path on the tree without any action (Step 9). This node maintains source IP addresses ranging from 192.168.1.16-.31 (/28 = 16 IPs), and a link connects from $N_{\{2,2\}}$ to $N_{\{3,2\}}$, called $L_{\{3,2\}}$. The last step (10) of the *SIP* level is inserting $r_{3\{SIP\}}$. Owing to $r_{3\{DP\}} \supseteq r_{1\{DP\}}$ and $r_{3\{SIP\}} \supseteq r_{1\{SIP\}}$, so $r_{3\{SIP\}}$ will be created on three paths: the path $L_{\{1,1\}}$, $L_{\{1,3\}}$, and $L_{\{1,4\}}$ respectively. Node $N_{\{3,3\}}$, $N_{\{3,4\}}$ and $N_{\{3,5\}}$ are created on the tree. $N_{\{3,3\}}$ contains 240 IPs ranging from192.168.1.16-.255, $N_{\{3,4\}}$, and $N_{\{3,5\}}$ holds 256 IPs (192.168.1.0/24). The link $L_{\{3,3\}}$, $L_{\{3,4\}}$, and $L_{\{3,5\}}$ connect to their nodes from *DIP* to *SIP* level respectively as shown in step 10 of Figure 8. The final step (Step 11 in Figure 9) of creating PSD phase is to build the protocol node into the tree at the *Pro* level. For example, protocols of r_1 is any (*), which is both TCP and UDP protocol. The action of r_1 is the acceptance (*a*), and it will be stored in the first node $N_{\{4,1\}}$ of the *Pro* level. In practice, the *a* is equal to 1 and *d* is equal to 0. The protocols of the remaining rules can be implemented similarly to inserting the *SIP*. Noticeably, the ordering of the rule fields on the tree structure is *DP*, *DIP*, *SIP*, and *Pro* respectively. This arrangement will minimize the number of branches of the tree. In addition, the four fields (*DP*, *DIP*, *SIP*, *PRO*) of a firewall rule are sufficient to verify how to match a packet against any rule. The Figure 9 is the completed PSD tree structure after phase 2 is finished. The data type used to store rules is represented below, and the algorithms, converting rules to the tree structure, are shown in Algorithm 4 and Algorithm 5.

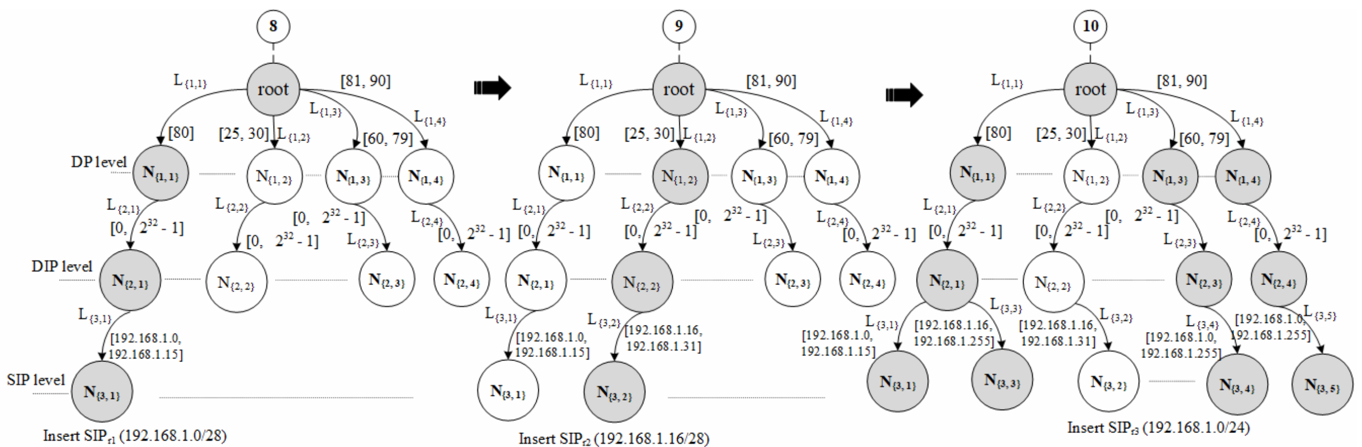

Figure 8. Inserting all source IP addresses from the rules to PSD (*SIP* level)

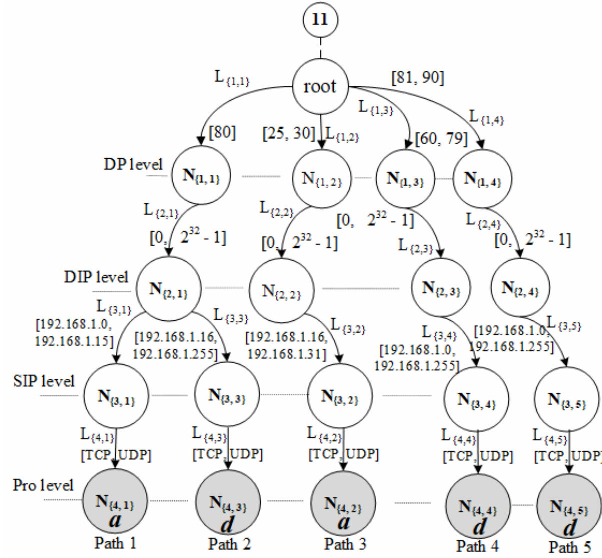

Figure 9. Inserting protocols and actions from the rules to PSD (*Pro* level)

Algorithm 4. The Path Selection Diagram (PSD)

1. **Input:** Rules $\{r_1, r_2, ..., r_n\}$, where $n \in Z^+$, $n \neq 0$
2. **Output:** The Path Selection Diagram (PSD)
3. **if** PSD = NULL **then**
4. *# Create the root node and first path of PSD tree*
5. r_1 ← Read firewall Rules
6. node* root ← new node(NULL), node* head ← root
7. **while** f ← Read each field from r_1 until NULL **do**
8. head.child = new node(f)
9. head = head.child
10. **end while** *# The first path was successfully created*
11. **while** rule ← Read each rule from Rules **do**
12. head ← root.child
13. f ← Read first field from rule
14. Call Subtree(head, f, rule)
15. **end while**
16. **end if**
17. **Return** PSD
18. **End**

Algorithm 5. Subtree (node, f, rule)

1. **Input:** head, f = *each field of rule*, rule
2. **Output:** Subtree
3. *# First case:* $f \not\subset$ *all fields in flat level*
4. **if** head.data-f == ϕ **and** head.sibling == NULL **then**
5. head.sibling ← new node(f)
6. head ← head.sibling
7. **while** f ← Read each field from rule **do**
8. head.child ← new node(f)
9. head ← head.child
10. **end while**
11. **Return**
12. **Else**
13. **if** head.data-f == ϕ **and** head.sibling ≠ NULL **then**
14. head ← head.sibling
15. Call Subtree(head, f, rule)
16. **end if**
17. **Return**
18. **end if** *# Finished first case*
19. *# Second case:* f == *all fields in deep level*
20. **if** head.data-f == 0 **and** head.child == NULL **then**
21. **Return**
22. **else**
23. **if** head.data-f == 0 and head.child ≠ NULL **then**
24. head ← head.child
25. f ← head.data
26. Call Subtree(head, f, rule)
27. **end if**
28. **end if**
29. *# Third case:* $f \subset$ *of all fields in flat and deep level*
30. **if** head.data-f ≠ ϕ **then**
31. d ← head.data-f
32. head ← head.sibling
33. Call Subtree(head, d, rule)
34. head ← head.child
35. Call Subtree(head, f, rule)
36. **end if**
37. **End**

```
struct {
    SET data; #data as SET data type
    struct node* sibling;
    struct node* child;
} node;
```

The phase 2's conclusions. After PSD tree has been built successfully, it is guaranteed that conflicts will not occur. Supposing a packet p_i consists of DP = 80,

DIP = any, SIP = 192.168.1.10, and Pro = TCP, this packet will be matched both r_1 and r_3 in Table 2. In traditional firewalls, the conflict arises because the action of r_1 and r_3 is different. However, this situation does not occur with PSD tree because the packet will be traversed by the path number 1 only, and this packet will be adjudged to be acceptable (*a*).

4.3 Phase 3: Mapping PSD Tree to Sparse Matrices

The PSD tree from phase 2 is fetched to the input of phase 3. This phase aims to speed up the memory access from O($\log_n$) of the PSD tree to O(1) by indexing on the matrix. Furthermore, firewall rules stored in the PSD tree are also shared so the size of the tree can be optimal. The sparse matrix is two-dimensional (2D), excluding the matrix used to store the port number is one dimensional (1D) as the following Figure 10. The *DP* matrix is used to store port numbers ranging from 0 to 65535. If each port consumes 16-bit integer, the total memory consumption of *DP* is ≃ 131 kilobytes (65,536 * 16 bits for supporting 65,536 rules). The number of protocols used for IPv4 is 2^8, so the memory space for Pro is ≃ 33.55 megabytes (256 * 8 bits (for actions) * 65,536). Unlike the case of *DIP* and *SIP*, both are four octets for IPv4, which is divided into four parts by dots such as 192.168.1.10. Thus, the matrix space for storing both *DIP* and *SIP* is 4 octets * 256 * 16 bits * 65,536 * 2, which is ≃ 268 megabytes. Tree mapping to matrices can be done simply by starting each breadth level. For example, the *DP* level is mapped first, the port number 80 is mapped to the 80th address of the 1D matrix of *DP*, and this address keeps the path number 1 as shown in Figure 11. Other ports in this flat level are also implemented in the same way. The port number ranges from 25 to 30, pointing to the address 25 to 30 in the *DP* matrix and storing the path number 2. The addresses ranging from 60 to 79 store the path number 3, and addresses ranging from 81 to 90 store the path number 4. For the second flat level (*DIP*), IPs of all routes range from 0.0.0.0 to 255.255.255.255, but they differ in path directions. Thus, the row number 1 of all *DIP* matrices is set to be 1 (Path no. 1), and the number 2, 3 and 4 are set to all elements of the row number 2, 3 and 4 respectively. For *SIP* level, the path 1 and 2 share *DIP* endpoint, so they use the same memory space in the first row of *SIP* matrices together. For example, the matrix of octet 1 (Address 192), octet 2 (168), and octet 3 (1) are set to be 'X' (Don't care term), and other addresses are set to 0. In practice, 'X' is set to anything (Usually to be-1). The continuous memory from the position 0 to 15 of the matrix (Octet 4) keeps number 1 (Path 1), and the position 16 to 255 keep number 2 (Path 2). For the other routes in *SIP* level, they have the similar mapping. In the *Pro* level, it only uses the TCP and UDP protocol, thus the position number 6 (UDP) and 17 (TCP) is set to be 0 (*accept*) or 1 (*deny*).

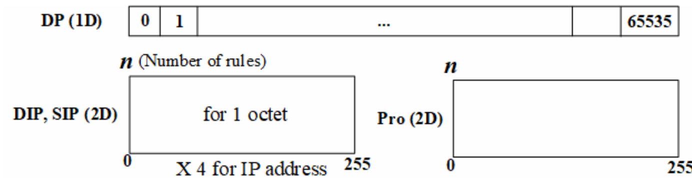

Figure 10. 1D and 2D matrices for storing firewall rules

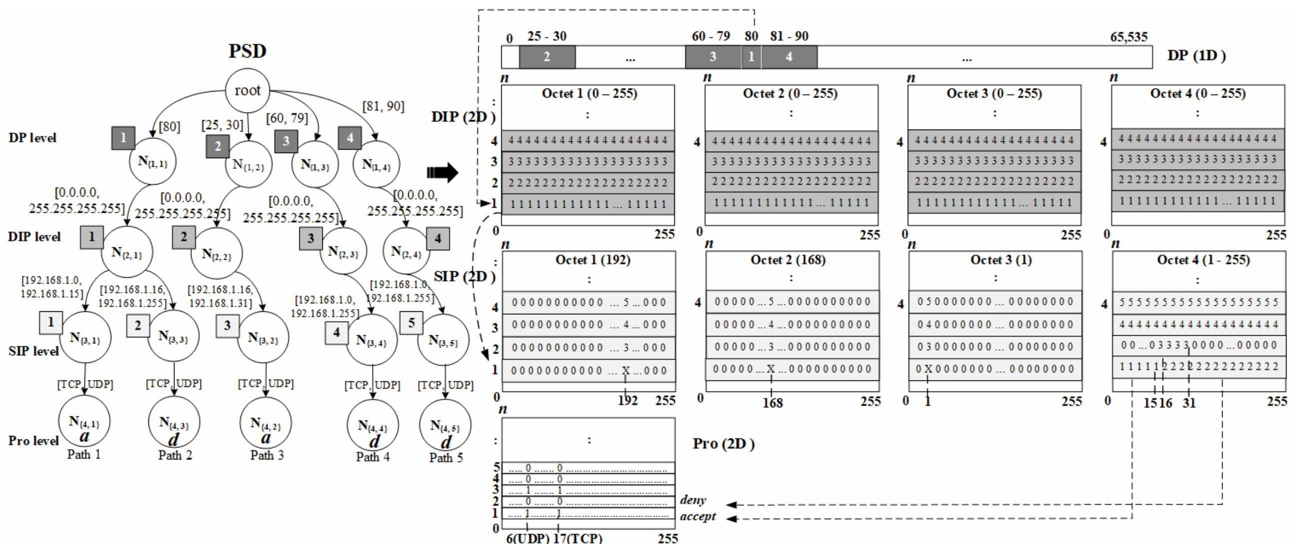

Figure 11. Mapping all paths and nodes from the PSD tree to matrices

The phase 3's conclusions. Searching time of tree structures are usually $O(\log_n)$ but the matrices are O(1). The goals of this phase are to change the searching time from $O(\log_n)$ to O(1) and represent redundant data from the shared rules.

4.4 Phase 4: Packing Sparse Matrices to 1D Arrays

Interestingly, the data in the matrices from phase 3 are very duplicate and low non-zero elements. In other words, it means a wasted memory space. The object of this phase is to decrease the amount of a wasted memory by the packing method, called IP Packing. The principle of this method is to compact the data from the 2D matrix structures to 1D arrays as shown in Figure 12. First, the *DP* matrix is already 1D array structure, so we do not pack this matrix. Next, *DIP* matrices are packed to the 1D array. The Row Lookup Table (RLT) is used to address to the actual data in the 1D array of each *DIP* octet. RLT has two columns: COL (Column index) for calculating an address of an actual data in the 1D array and END for identifying the last number of each *DIP* octet. If the END is an asterisk (*), it means any number from 0 to 255. For example, the first row of the *DIP* octet 1 ranging from 0 to 255 contains the path number 1, thus the RLT records the number 0 (Starting position of the 1D array) to the COL and "*" to the END. The second row of the *DIP* octet 1 (Containing the path number 2) is compacted to the second row in RLT. The COL contains the number 1 pointing to the address number 1 in the 1D array. For the remaining rows of the *DIP* octet 1, they have the same operations. The data of each row of *DIP* octet 2, 3, and 4 are compacted to the $1D_{\{o2-o4\}}$ array and $RTL_{\{o2-o4\}}$ respectively. The next step is a compactness of all 2D of *SIP* octets to 1D arrays. For example, the row no. 1 of the *SIP* octet 1 contains only one non-zero value, which is "X" (Don't care term) in the column number 192. The address of this value in the 1D array of $SIP_{\{O1\}}$ is 0 because this value is the first element in its 1D array; therefore, the COL in the first row of RLT must be subtracted by-192 in order to point to the zero address in the 1D array. The END of this row is 192 because it has only one value. In case of the *SIP* octet 4, the row no.1 contains a set of several path numbers; for example, the positions ranging from 0 to 15 contain the number 1 (Path number 1), and the positions starting from 16 to 255 pack up the number 2. The compactness algorithm copies all values from the row no.1 in *SIP* octet 4 to the 1D array positioning 0 to 255 (256 addresses), and the index of the 1D array points to the address 256 automatically for storing the next data set. The COL and END record 0 and 255 in the RLT table respectively. The next row of *SIP* octet 4 (Row no.2) contains the number 3, starting at position 16 to 31, so the COL of RLT records 240 (256-16), and the END records 31 (The last number of *SIP* octet 4), and the 1D array of *SIP* octet 4 starting from 256 to 271 is replaced by 3. Other rows have the same compacted operation as mentioned above, and as shown in the following Figure 12.

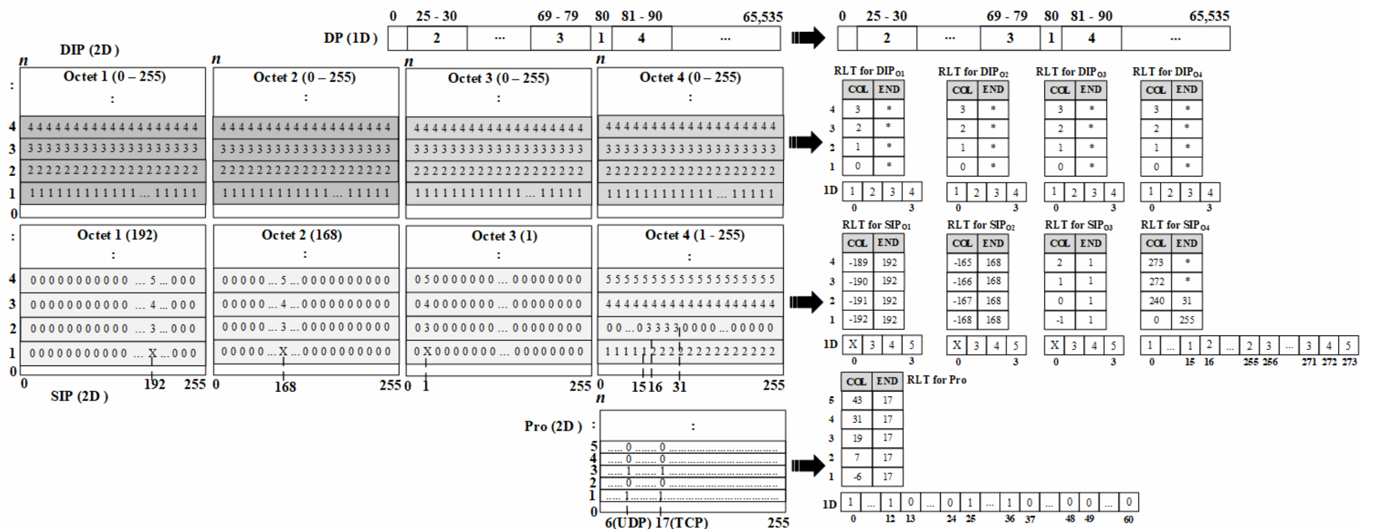

Figure 12. Packing all matrices to 1D arrays

The phase 4's conclusions. The data structures after packing process: 1) one 1D for *DP*; 2) 4 Row Look up Tables (RLT of $DIP_{\{o1-o4\}}$) and one 1D for *DIP*; 3) 4 Row Look up Tables (RLT of $SIP_{\{o1-o4\}}$) and one 1D for *SIP*; and 4) one Row Look up Table (RLT of *PRO*) and one 1D for *Pro*. The memory space of IPack after this phrase is O(n).

4.5 Phase 5: Testing Firewall Rule Matching

This section shows the processes of the firewall rule matching or verifying against an incoming or outgoing packet by IPack. Supposing an incoming packet p_i consists of *DIP* = 200.10.0.100, *SIP* = 192.168.1.16, *DP* = 80, and the TCP protocol (Number 17) will pass

through IPack. Step 1, the *DP* of p_i is used to point to the address number 80 in the 1D array of *DP* as shown in Figure 13. The data in this address is used to point to the row number 1 of RLT for *DIP*. The row no. 1 of all RLT of *DIP* contains COL = 0 and END = '*', which means that it is any IP address (END = *) and points to the column 0 (COL = 0) in the 1D array of *DIP* (Step 2). The column 0 of all 1D array of *DIP* contains the number 1 (Row no.1) pointing to their row number in RLT of *SIP*. Step 3, each *SIP* octet of p_i is computed with each COL in RTL of *SIP*, such as the first octet of *SIP* of p_i subtracts the COL in row number 1 of RLT for $SIP_{\{o1\}}$ (192-192) = 0, the 2nd octet of *SIP* of p_i subtracts the COL in row no. 1 of RLT for $SIP_{\{o2\}}$ (168-168) = 0, the 3rd octet of *SIP* of p_i subtracts the COL in row no. 1 of RLT for $SIP_{\{o3\}}$ (1-1) = 0, and the last octet of *SIP* of p_i subtracts the COL in row no. 1 of RLT for $SIP_{\{o4\}}$ (16-0) = 16. The address number 0 of the 1D array of $SIP_{\{o1,o2,o3\}}$ is 'X', and it refers to the ignored terms. However, there will be at least one term that is not 'X', which is the $SIP_{\{o4\}}$(16). This address (16) stores the row number of RLT of the next level (*Pro*), which is the number 2 (Row no. 2 of RLT for *Pro*). In Figure 13, COL keeps the number 7, and END collects the number 17 (Protocol TCP). Thus, the firewall matching algorithm will calculate the *Pro* of p_i (17) with the data stored in COL (7). The result is 24 (17 + 7), which is the pointer to the firewall decision at the address 24 of the 1D for *Pro*. The packet p_i is decided to be a drop (0) immediately. Matching any packet against IPack is shown in Algorithm 6.

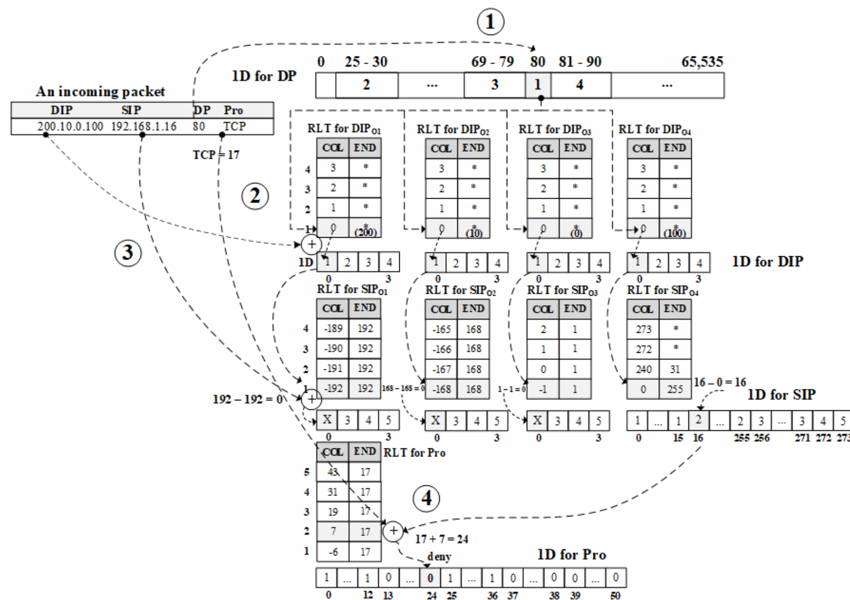

Figure 13. Matching an incoming packet against firewall rules in IPack structures

Algorithm 6. Packet matching algorithm of IPack

1. **Input:** a packet pi, 1-D of DP, RLT and 1-D of DIP_{o1-o4}, RLT and 1-D of SIP_{o1-o4}, RLT and 1-D of Pro
2. **Output:** *drop* (0) or *accept* (1)
3. dp ← read DP from pi, dp ← read 1D-DP[dp]
4. **if** RLT-DIP[dp][END]$_{o1-o4}$ == * **then**
5. dip_{o1-o4} ← RLT-DIP[dp][COL]$_{o1-o4}$
6. dip-i_{o1} ← 1D-DIP[dip_{o1}]$_{o1}$, dip-i_{o2} ← 1D-DIP[dip_{o2}]$_{o2}$, dip-i_{o3} ← 1D-DIP[dip_{o3}]$_{o3}$, dip-i_{o4} ← 1D-DIP[dip_{o4}]$_{o4}$
7. **Else**
8. dip_{o1-o4} ← read DIP from pi
9. **if** dip_{o1-o4} 0 1D-DIP[dp][END]$_{o1-o4}$ **then**
10. dip-i_{o1} ← 1D-DIP[dp][COL]$_{o1}$ + dip_{o1}, dip-i_{o2} ← 1D-DIP[dp][COL]$_{o2}$ + dip_{o2}, dip-i_{o3} ←1D-DIP[dp][COL]$_{o3}$ + dip_{o3}, dip-i_{o4} ← 1D-DIP[dp][COL]$_{o4}$ + dip_{o4}
11. **Else**
12. **return** drop # *a packet mismatch while check DIP*
13. **end if**
14. **end if**
15. **if** RLT-SIP[dip-i_{o1-o4}][END]$_{o1-o4}$ == * **then**
16. sip_{o1-o4} ← RLT-SIP[dip-i_{o1-o4}][COL]$_{o1-o4}$
17. sip-i_{o1} ← 1D-SIP[sip_{o1}]$_{o1}$, sip-i_{o2} ← 1D-SIP[sip_{o2}]$_{o2}$, $sipi_{o3}$ ← 1D-SIP[sip_{o3}]$_{o3}$, sip-i_{o4} ← 1D-SIP[sip_{o4}]$_{o4}$
18. **else**
19. sip_{o1-o4} ← read SIP from pi
20. **if** sip_{o1-o4} ≤ 1D-SIP[dip-i_{o1-o4}][END]$_{o1-o4}$ **then**
21. sip-i_{o1} ← 1D-SIP[dip-i_{o1}][COL]o1 + sip_{o1}, sip-i_{o2} ← 1D-SIP[dip-i_{o2}][COL]$_{o2}$ + sip_{o2}, sip-i_{o3} ← 1D-SIP[dip-i_{o3}][COL]$_{o3}$ + sip_{o3}, sip-i_{o4} ← 1D-SIP[dip-i_{o4}][COL]$_{o4}$ + sip_{o4}
22. **else**
23. **return** drop # *a packet miss match while check SIP*
24. **end if**
25. **end if**

26. pro-i ← read only sip-io_{1-o4} ≠ ‘X’
27. pro ← read Pro from pi
28. **if** pro ≤ RLT[pro-i][END] **then**
29. pro-i ← RLT-PRO[pro-i][COL] + pro
30. **return** 1D-PRO[pro-i]
31. **else**
32. **return** drop # *a packet miss match while check Pro*
33. **end if**
34. **End**

The final phase (5) conclusion: the concept of IPack matching process is to use the information header of an incoming or outgoing packet to act as pointers to RLTs and 1D structures, thus the speed of matching packets is O(1).

5 IPack Implementation

The IPack firewall has been developed by the C/C++ language (GCC 4.4.7) and GNU Make 3.8 on 64-bit Linux kernel version 2.6. IPack consists of two parts: the user space and kernel space.

The user space (IPack.o) parses the firewall rules from an administrator like adding, editing, and removing rules; it also builds the PSD tree. If firewall rules are valid according to grammar rules, they will pass through to the kernel space by /procf file system. The kernel space (IPack_klm.ko) executes the rules passed by an administrator from /procf file by using the procf_read() and procf_write() function. The main function of IPack kernel space is to block, pass, and capture packets based on the built-in inbound and outbound filtering function as illustrated in Figure 14. IPack source files can be downloaded from https://isan.msu.ac.th/suchart/IPack/.

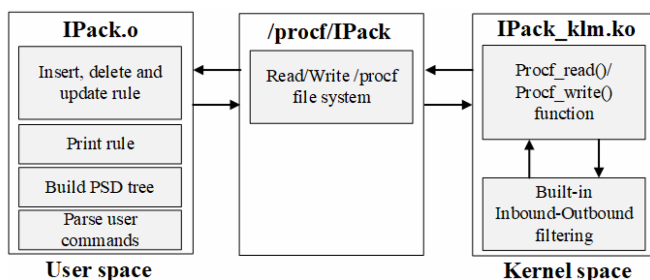

Figure 14. Implementing IPack on real Linux system

6 Testbed Network Environment

IPack and IPsets are tested on the real-world network as shown in Figure 15. This paper uses IPERF software [28], a tool for active measurements of the maximum achievable bandwidth on IP networks, for both firewall throughput testing. The firewalls are installed on the client site [29]. Throughput testing is achieved by evaluating the response time of packets from the IPERF server, the packets are generated from the IPERF client via the installed firewalls. The client-side network connectivity starts from the 3BB (ISP) using VDSL technology, which has a bandwidth of 30/5 Mbps (Download/Upload stream), connecting to the Internet. The server site network starts at the Internet to UNINET (ISP), and the endpoint connection is at Mahasarakham University, Thailand, and the IPERF server is installed here. Firewalls are tested by the most popular protocols [30], i.e., TCP and UDP. The criteria for testing TCP against firewalls: the window size = 16, 32, and 64 KB respectively, the interval time = 1 sec, and the concurrent = 1 connection. In UDP testing, the bandwidth size used to test firewalls is 100 Mbps. The number of rule sets are 100, 500, 1,000, 2,000, 3,000, 4,000, 5,000, 10,000, 20,000, 30,000 respectively.

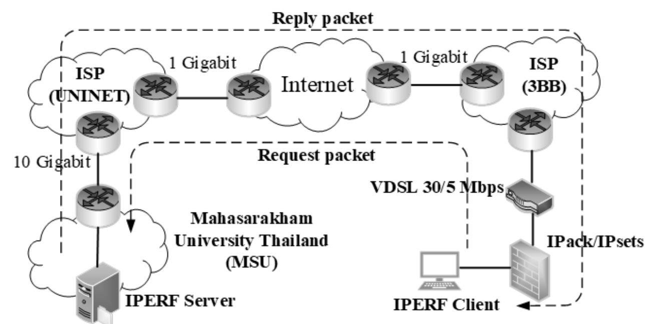

Figure 15. IPack vs IPSets testbed on the real network

7 Firewall Performance Evaluation

Both firewall performances are evaluated by two approaches: the processing speed and memory usage. The processing speed is divided into two parts: the speed for building data structures and speed for matching rules. First, the time to build IPack data structures using Algorithm 4 and Algorithm 5 takes $O(n^2)$, which is similar to IPSets using Algorithm 2. Second, the matching time of both techniques is equal to O(1) because IPSets uses hash tables and IPack applies array structures for accessing the data. In the case of memory consumption, if hash tables of IPSets are not enough to store the data, the hash table size will be increased by 2^n. Thus, the space complexity is $O(2^n)$. However, IPack is slightly different for allocating the memory because it uses the array as the data structures. For example, in the worst case (Figure 12), IPack allocates 2 * *n* (Number of rules) * 9 (Number of RTLs) + 65,536 (1D of *DP*) + 256 * n * 9 (1D of *DIP*, *SIP* and *Pro*) = 65,536 + 2,322 * *n*; therefore, the space complexity of IPack is O(n).

The firewalls’ TCP throughputs running on the real-world network are shown in Figure 16 and Figure 17. Figure 16 shows throughputs of the packet transfers between IPSets and IPack by different TCP window sizes. The number of firewall rules is evaluated with

both firewalls running from 100 to 30,000. Observed by the overall, throughputs of both firewalls are similar. For example, the throughputs are around 275, 482 and 659 KB/sec by the windows sizes as 16, 32, 64 KB respectively. In Figure 17, throughputs are measured by the amount of data received and sent at the bit rate. Both firewalls can be processed consistently without depending on the number of firewall rules. The throughputs of windows sizes are 16, 32, 64 KB around 2.2, 3.7 and 5.35 Mbps respectively. According to the UDP transfers, they have tested in the same way as TCP, measuring packet-level and bit-rate traffic at a data transfer rate of 100 Mbps. The results are still constant about 11 KB/sec at packet-level transfers (Figure 18) and 97 Mbps/sec at bit-rate transfers (Figure 19).

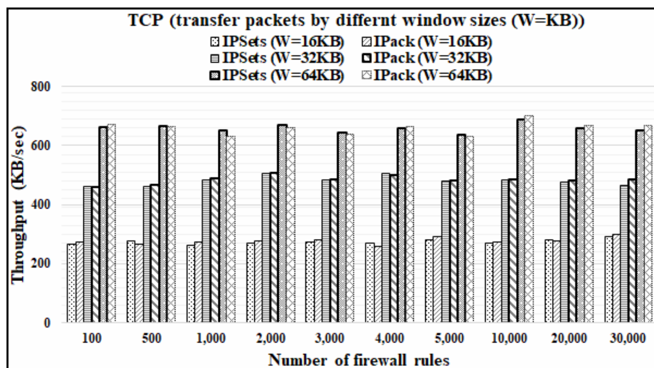

Figure 16. Packet transfers of IPSets and IPack (TCP)

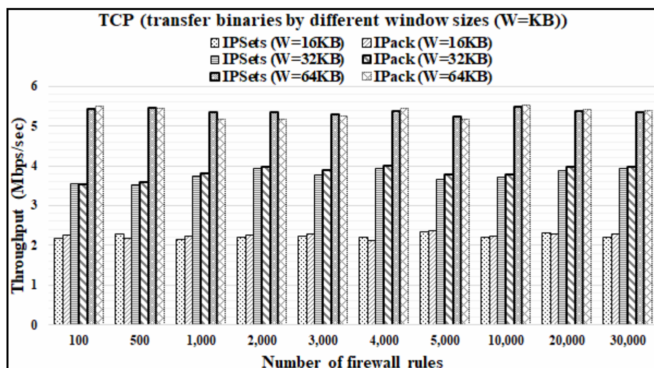

Figure 17. Binary transfers of IPSets and IPack (TCP)

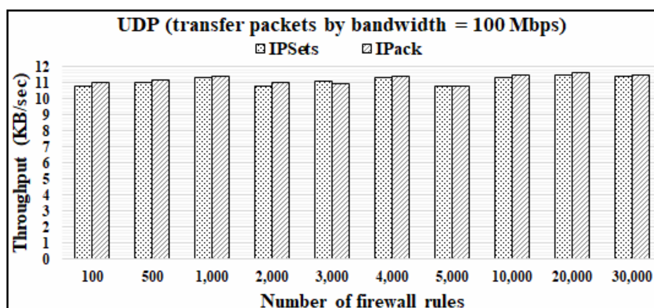

Figure 18. Packet transfers of IPSets and IPack (UDP)

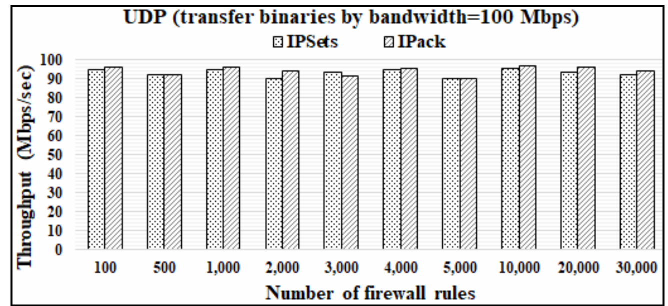

Figure 19. Binary transfers of IPSets and IPack (UDP)

8 Conclusion

This paper designs and develops the firewall with O(1) worst case access time, called IPack. It can resolve the drawbacks of IPSets such as the rule management, rule conflicts, and the IP classes. Especially, it consumes memory usage less than IPSets. IPack is also tested on the real-world network, the results show that it can operate on the high-speed networks as IPSets. The algorithm applied with IPSets is the perfect hashing, and it works on IPTables. However, IPack uses the path selection diagram (PSD), sparse matrix and packing algorithm, and operates on Netfilter. IPSets handles with packet verification (Matching rules against packets) only for the packet decisions (*accept* or *deny*) to be processed by IPTables. Unlike IPack, which is designed to fully support firewall operations, excluding NAT (Network Address Translation) and the packet filtering. The final conclusions between IPSets and IPack are shown in Table 3.

Table 3. Summaries of IPSets vs IPack performance

Firewall Feature	Description of	
	IPSets	IPack
Time complexity for building structures	$O(n^2)$	$O(n^2)$
Time complexity for rule verifying	O(1)	O(1)
Space complexity	$O(2^n)$	O(n)
IP class supports (A, B, C and D)	B, C, D	All
Basic operations of firewalls	Matching	All
Rule anomaly detection and correction	No	Yes
Rule definition	Manual	Auto
Admin skills for handling firewalls	Expert	Normal

Note. basic operations: matching, decision and forwarding.

References

[1] X. Jia, J. K. H. Wang, Distributed Firewall for p2p Network in Data Center, *2013 IEEE International Conference on Consumer Electronics*, Shenzhen, China, 2013, pp. 15-19.

[2] C. Wang, D. Zhang, H. Lu, J. Zhao, Z. Zhang, Z. Zheng, An Experimental Study on Firewall Performance: Dive into the Bottleneck for Firewall Effectiveness, *2014 10th International Conference on Information Assurance and Security*, Okinawa, Japan, 2014, pp. 71-76.

[3] H. Welte, P. N. Ayuso, *The Iptables Project*, IPTables Research Report 1077, June, 2014.

[4] R. Zhan-rui, W. Yong-jun, S. Yong-lin, Based-service Grouping Firewall Policy Conflict Checking, *CSAE 2011-2011 IEEE International Conference on Information Science and Engineering*, Jeju Island, Korea (South), 2011, pp. 33-37.

[5] L. Zhang, M. Huang, A Firewall Rules Optimized Model Based on Service-Grouping, *WISA 2015-2015 12th Web Information System and Application Conference*, Jinan, China, 2015, pp. 142-146.

[6] H. B. Acharya, M. G. Gouda, Linear-time verification of Firewalls, *2009 17th IEEE International Conference on Network Protocols*, Princeton, NJ, USA, 2009, pp. 133-140

[7] H. Hamed, A. El-Atawy, 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, October, 2006.

[8] S. Khummanee, A. Khumseela, S. Puangpronpitag, Towards a New Design of Firewall: Anomaly Elimination and Fast Verifying of Firewall Rules, *JCSSE 2013-2013 The 2013 10th International Joint Conference on Computer Science and Software Engineering*, Maha Sarakham, Thailand, 2013, pp. 93-98.

[9] T. Chomsiri, X. He, P. Nanda, Z. Tan, Hybrid Tree-rule Firewall for High Speed Data Transmission, *IEEE Transactions on Cloud Computing*, Vol. 4, No. 99, pp. 1-13, April, 2016.

[10] D. Rovniagin, A. Wool, The Geometric Efficient Matching Algorithm for Firewalls, *IEEE Transactions on Dependable and Secure Computing*, Vol. 8, No. 1, pp. 147-159, July, 2011.

[11] H. Thomas, *HiPAC High Performance Packet Classification for Netfilter*, Master Thesis, Universitat des Saarlandes, Fachbereich, German, 2004.

[12] H. Welte, P. N. Ayuso, *The Packet Filtering Framework,* Netfiler Research Report 2016, May, 2016.

[13] J. Kadlecsik, *IP Sets Framework,* IPSet Research Report 2017, April, 2017.

[14] H. Hu, G. J. Ahn, K. Kulkarni, Detecting and Resolving Firewall Policy Anomalies, *IEEE Transactions on Dependable and Secure Computing*, Vol. 9, No. 3, pp. 318-331, January, 2012.

[15] F. Chen, B. Bruhadeshwar, A. X. Liu, A Cross-domain Privacy-preserving Protocol for Cooperative Firewall Optimization, *2011 IEEE Proceedings INFOCOM*, Shanghai, China, 2011, pp. 2903-2911.

[16] E. S. Al-Shaer, H. H. Hamed, Modeling and Management of Firewall Policies, *IEEE Transactions on Network and Service Management*, Vol. 1, No. 1, pp. 2-10, April, 2004.

[17] A. X. Liu, E. Torng, C. R. Meiners, Firewall Compressor: An Algorithm for Minimizing Firewall Policies, *IEEE INFOCOM 2008-2008 The 27th Conference on Computer Communications*, Phoenix, AZ, USA, 2008, pp. 691-699.

[18] K. Lubna, R. Cyiac, Kavitha Karun A., Firewall Log Analysis and Dynamic Rule Re-Ordering in Firewall Policy Anomaly Management Framework, *ICGCE 2013-2013 International Conference on Green Computing*, Communication and Conservation of Energy, Chennai, India, 2013, pp. 853-856.

[19] E. Saboori, S. Parsazad, Y. Sanatkhani, Automatic Firewall Rules Generator for Anomaly Detection Systems with Apriori Algorithm, *ICACTE 2010-2010 3rd International Conference on Advanced Computer Theory and Engineering*, Chengdu, China, 2010, pp. 57-60.

[20] A. X. Liu, Formal Verification of Firewall Policies, *2008 IEEE International Conference on Communications*, Beijing, China, 2008, pp. 1494-1498.

[21] S. Pissanetsky, *SparseMatrix Technology*, Academic Press Inc, 1984.

[22] L. Robert, *Data Structures and Algorithms in Java*, 2nd Edition, Pearson Education, 2017.

[23] R. Shahnaz, A. Usman, I. R. Chughtai, Review of Storage Techniques for Sparse Matrices, *2005 Pakistan Section Multitopic Conference*, Karachi, Pakistan, 2005, pp. 1-7.

[24] S. A. V. Alfred, J. Menezes, P. C. van Oorschot, *Handbook of Applied Cryptography*, CRC Press, 1996.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms,* Third Edition, The MIT Press, 2009.

[26] J. Kadlecsik, G. Psztor, *Netfilter Performance Testing*, Netfiler Research Report 2004, December, 2004.

[27] L. C. Noll, *The Core of the FNV Hash*, FNV Research Report 2013, April, 2013.

[28] B. A. Jon Dugan, E. Seth, I*PERF-the Ultimate Speed Test Tool for TCP*, UDP and SCTP, IPERF Testing Report *2017*, June, 2017.

[29] K. M. Elleithy, R. C. Reddy, Comparison of Personal Firewalls Security and Performance Issues, *Journal of Internet Technology*, Vol. 3, No. 3 , pp. 175-186, Jul. 2002.

[30] D.-R. Lin, C.-I. Wang, D. J. Guan, An Efficient Blind Verifying for Firewall Using Online/Offline Signcryption, *Journal of Internet Technology*, Vol. 13, No. 4 , pp. 607-613, July, 2012.

Biography

Suchart Khummanee received the B.Eng. degree in Computer Engineering from the King Mongkut's Institute of Technology Ladkrabang, the M.Sc. degree in Computer Science from the Khon Kaen University, and the Ph.D. degree in Computer Engineering from the Khon Kaen University, Thailand. He is currently a full lecturer of Computer Science at the Mahasarakham University, Thailand. His research interests in the network security, computer networks, and Internet of things (IoT).