High-speed Firewall Rule Verification with $O(1)$ Worst-case Access Time

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 Nov. 6, 2015; revised and accepted Jan. 23, 2016)

Abstract

Firewalls enforced by rules are a security measure for verifying huge packets at gateway networks. Therefore, they probably act as bottlenecks of the networks. In this paper, we have presented several techniques to improve the speed of firewall rule verification with *O(1)* worst-case access time. The techniques are: policy mapping (PMAP), sparse matrix packing firewall (SMPF), perfect hashing firewall (PHF) and minimal perfect hashing firewall (MPHF). The experimental results show that they are as fast as IPSet, one of the most famous high-speed firewalls at present. However, they can get rid of IPSet limitations such as IP address classes, subnet size of each rule set and so on. Besides, on average, SMPF, MPHF and PFH can reduce the amount of memory usage of PMAP by 99.9, 87.7 and 62.3 percent respectively.

Keywords: Minimal perfect hashing firewall, perfect hashing firewall, policy mapping, rule verification, sparse matrix packing firewall

1 Introduction

The amount of traffic currently flowing in and out over the networks is massive (Gigabit per second: Gbps). Firewalls equipped at the network gateways also need to process increasingly high-volumes of data. This may lead to bottlenecks on the networks, because firewalls enforced by rules must verify contents in the header of every packet. In traditional firewalls, the packet is verified against firewall rules from the top to bottom as r_1, r_2, $\cdots$, r_{n-1} by order. A summation of packet sending and receiving over networks is proportional to the ability of firewall rule verification. For example, if a network infrastructure is capable to provide throughput over 1 Gbps, but the firewall can verify packets of about 0.5 Gbps only, the rest of packets (50%) being processed will be delayed and collected for processing in the next second. While the packets are increasing steadily, and have not been processed, the connections will be reset by the network protocol (Connection timeout), and connections are lost automatically. It results in a waste of time without benefits, because of applications have to retransmit new connections again and again. In addition, the number of firewall rules is also critical to the overall performance of firewalls. In a large company, for example, there are about 2,000 firewall rules. Each rule is checked 6 times per one packet; therefore, the firewall rules must be matched the total of 12,000 times per one packet. In traditional firewalls as Netfilter/IPTables [29], it verifies rules by the order, thus the worst-case access time is $O(n)$, where n is the number of firewall rules.

To modify traditional firewall rule verification, Alex X. Liu et al. [21] proposed a new concept called the firewall rule decision state diagram (FDD) instead of verification by the sequence. It applied to several contributions such as the verification of distributed firewalls [12], firewall policy queries [23], diverse firewall design [22] and so on [20]. Although, FDD clearly shows the firewall rule conflict paths, the computational complexity of rule verification is $O(n^d)$, where n is the number of rules and d is the number of checked fields in each firewall rule. After, Acharya and Gouda [1] proposed a linear time algorithm that reduced the time complexity of FDD from as $O(n^d)$ to $O(nd)$. Next, Hamed et al. [13] contributed an algorithm for optimizing unwanted rejection flows with a dynamic packet filtering by statistical search and trees structures; moreover, they also improved the speed of the computational complexity of packet matching as $O(n \log n)$.

The speed of rule verification has been improving continuously. Rovniagin and Wool [28] reduced the searching time of rule matching for $O(log\, n)$ and consumed a suitable memory for $O(n^4)$ by the Geometric Efficient Matching (GEM) technique. Khummanee et al. [18] presented the single domain decision approach (SDD) to eliminate firewall rule conflicts and the time for searching is $O(log\, n)$ by using the tree structure. The Tree-Rule firewall running on a cloud computing was introduced by Xiangjian et al. [14]. They showed that Tree-Rule used

tree structure had absolutely no conflicts or redundant rules, and they evaluated their speed of verification as $O(log\, n)$. According to the improved verification speed against tree structures, HiPAC [3, 15] issued the sophisticated HiPAC packet classification algorithm and the advanced tree structure. The result showed that HiPAC could improve the speed of rule matching from as $O(log\, n)$ to $O(log\, w)$, where w is the bit width of the packet field. Currently, the state of the art of high-speed firewall rule verification is IPSet [24] which is the top of high-speed firewall open source. It applied the perfect hashing function [10] for matching the firewall rules, their experimental result is $O(1)$. However, it has few drawbacks. First, the rules must be grouped to be a set of rules before deploying them to the perfect hashing function. Second, it is available for the IP class C and B only, excluding A. If anyone would like to use an IP class A, it needs to divide the IP class A to an IP class C first. Finally, designing rule of IPSet is not easy to understand, it needs an expert in firewall rule relationships. According to the limitations of the IPSet, Khummanee [19] proposed the policy mapping algorithm (PMAP) to solve the drawbacks of IPSet. It is also $O(1)$ of the matching time like IPSet; however, it has still a problem about the memory usage. We conclude the development on the speed of firewall rule verification in Table 1.

In this paper, we have optimized the space complexity of PMAP and compare it against other techniques that are $O(1)$ worst-case access time on the firewall rule verification. The rest of the paper is organized as follows: Section 2 presents the related work, high-speed firewall designs are explained in Section 3. In Section 4, we demonstrate the performance evaluation. Finally, we give conclusions and future work in Section 5.

Table 1: History of the speed of rule verification

No.	Article's Name	Time
1	FDD[22]	$O(n^d)$
2	Linear-Time[1]	$O(n * d)$
3	IPTables[29]	$O(n)$
4	Dynamic Opt[13]	$O(n\, log\, n)$
5	GEM[28], SDD[18], Tree-Rule[14]	$O(log\, n)$
6	HiPAC[3]	$O(log\, w)$
7	IPSet[24], PMAP[19]	$O(1)$

Sort by lowest (No. 1) to highest speed

2 Related Work

2.1 Firewall Basic

Basically, a firewall rule consists of six parts: Source IP address (SIP), Destination IP address (DIP), Source Port (SP), Destination Port (DP), Protocol (Pro) and Action (Act). The first five parts are called the predicate, and the last part is called the action. Every packet flowing in the networks is matched against SIP, DIP, SP, DP and Pro of a firewall rule (r_n, $n \in \mathbb{Z}^+$) by order. If a packet matches all parts of the predicate ($\forall p_{x_i} \in r_{n_i}$, x and $n \in \mathbb{Z}^+$, $i \in \{SIP, DIP, \cdots, Act\}$), an action is operated by an Act part (accept or deny). A packet evaluated to be acceptable (accept: a) is forwarded to a destination IP address defined in its header field (DIP). On the other hand, an unacceptable packet (discard or deny: d) is automatically dropped [19]. According to Table 2, rule No. 1 (r_1) represents that the firewall allows source IP addresses ranging from 0.0.0.10 to 0.0.0.30 (21 hosts) onto destination IP addresses in the range between 0.0.0.20 and 0.0.0.30 (11 hosts), any source ports ($* \in \{0, 1, \cdots, 65,535\}$), a destination port number 80, and TCP or UDP protocol can pass through the firewall ($Act \rightarrow a$). In contrast, rule No. 3 (r_3) drops every packet from source IP addresses ranging from 0.0.0.1 to 0.0.0.40 onto destination IP addresses in the range of 0.0.0.25 to 0.0.0.35, any source port (*), a destination port number 80, and both protocols. Moreover, firewalls have an option that allows an administrator to set the final rule. This rule is to drop all packets that are not explicitly allowed ($r_1 - r_{n-1}$) at the bottom of the rule list (r_n) [31].

Table 2: Firewall rule examples

No.	*SIP*	*DIP*	*SP*	*DP*	*Pro*	*Act*
r_1	0.0.0.10-30	0.0.0.20-30	*	80	*	a
r_2	0.0.0.1-15	0.0.0.50-60	*	25-30	*	a
r_3	0.0.0.1-40	0.0.0.25-35	*	80	*	d
r_4	0.0.0.15-45	0.0.0.1-100	*	60-90	*	d
r_{n-1}	...	...	...	...	...	...
r_n	*	*	*	*	*	d

$*(SIP, DIP) = 0$ - $2^{32} - 1$, $*(SP, DP) = 0$ - $2^{16} - 1$, $*(Pro) =$ TCP and UDP, $a =$ accept, $d =$ deny

2.2 Minimal and Perfect Hashing

The firewall problem adapted with tree structures is to speed up of searching, because of the trees have a limit of the worse-case runtime as $O(log\, n)$ only. In this section, we represent the data structures that can be searched in $O(1)$ time, this concept is referred to as hashing. A ***hash table*** is a collection of stored data items. Let **U** be universal keys $= \{0, 1, \cdots, m-1\}$, where $m \in \mathbb{N}_0$. **T** denotes a hash table $T[0, 1, \cdots, m-1]$, in which each position, or slot, corresponds to a key k_i, where $i \in \mathbb{N}_1$, in the universal U. Initially, the hash table contains no values, thus every slot is empty (NULL) as shown in Figure 1. Mapping between an value and a slot where that value belongs in the hash table is called the ***hash function (h)***. The hash function (h) computes the slot from the key k_i. In other words, h maps the universe U of keys into the slot of hash table $T[0, 1, \cdots, m-1]$, denotes as $h : U \rightarrow \{0, 1, \cdots, m-1\}$, where the $|m|$ of the hash table T is commonly more less than $|U|$. The function (h) hashes an value with the key k_i to slot $h(k_i)$ of T, we say that

$h(k_i)$ is the ***hash value*** of the key k_i. Figure 1 illustrates the basic concept of hashing. The hash function can reduce the size of $|U|$ to size m.

The first simply hash function, sometimes referred to as the "remainder method", divides an key k_i by the table size ($|T|$), returning remainder as its hash value $\boldsymbol{h}(k_i) = (k_i \,\%\, |T|)$. Assume that we have the set of keys and values (k_i:'$value_i$') of the ASCII code = {65:'A', 72:'H', 71:'G', 74:'J', 85:'U'}, and $|T| = 10$. Thus, the results of the hash values for our example: 5 (65 % 10), 2, 1, 4 and 5 respectively. Note that 5 of the 10 slots are now occupied. This is referred to as the ***load factor***, and is normally denoted by $\lambda = \frac{number\_of\_values}{table\_size}$. For this example, $\lambda = \frac{5}{10} = 0.5$. Once the hash values have been executed, we can insert each value into the hash table at the assigned position as shown in Figure 2. This h works well when each value is mapped to a unique position in T. However, in this example, there are two keys hashed to the same slot ($h(k_1) = h(k_5) = 5$). We refer this situation to as a ***collision*** (also called a "clash"). We need to select a systematic approach for replacing the second value ($h(k_5) = h(85)$) in the hash table by without overlapping with the others. Fortunately, effective techniques are unfolded in several data structures and algorithm books [7, 30]. In this section, we present the simplest technique to resolve the collision, called 'chaining' only.

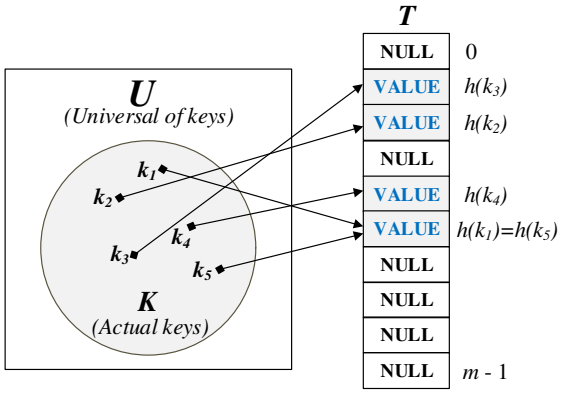

Figure 1: Using hash function (h) maps actual keys with values to hash-table slots

In the ***chaining*** approach, we put all values that hash to the same slot in T into the chain of the link list as Figure 2. The slot $h(k_i) = 5$ contains a pointer to the head of the link list where all values hashed by $h(k_i) = 5$ are stored, the tail of the link list always contains NULL. Unfortunately, the worst-case searching time is proportional to the length of the link list, that is $O(n)$, where $n <= k$, $k = |K|$. There are several effective techniques that solve the collision, such as open addressing, liner probing, and so on [7, 30]; however, these topics are beyond this paper.

The remainder method is a single hash function or fixed hash function. To yield an average retrieval time where the fixed hash function is $\Theta(n)$ in which all keys are hashed to the same slot, means a high collision rate.

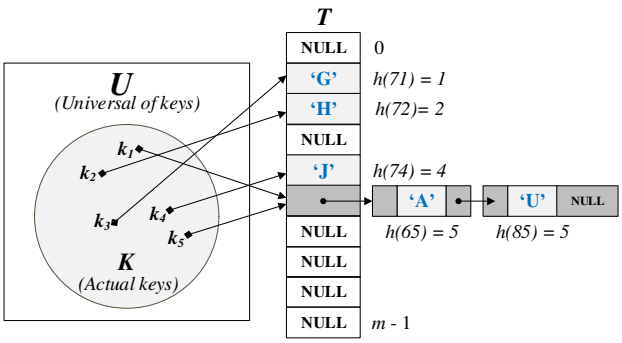

Figure 2: Five hash values $h(k_1), \cdots, h(k_5)$ in hash table

The effective technique used to improve the vulnerable fixed hash is to choose a hash function at random from a family of hash functions instead of the fixed hash function. This technique, called ***universal hashing***, has an affect on a good average access time, and guarantees a low number of collisions.

Let U be the set of universe keys and $\mathcal{H}$ be a finite collection of hashing functions mapping U into the range of integer $M = \{0, 1, \cdots, m-1\}$. Then $\mathcal{H}$ is called a universal family if $\forall x, y \in U, x \neq y : |\{h \in \mathcal{H} : h(x) = h(y)\}| = \frac{|\mathcal{H}|}{m}$. In the other words, the probability of a collision for any two different keys x and y hashed by a hashing function randomly which is chosen from $\mathcal{H} = \frac{1}{m}$. Choosing the randomness hashing function family and the uniform hashing are shown in [7]. We can see that the universal hashing is the best situation for the $average - case$ performance; however, the universal hashing can also improve $worst - case$ performance when the set of keys is $static$ (i.e. the set of keys are known). It is possible to compute the effective hashing function that can find any key in one probe ($O(1)$) in the hash table and guarantees that it has no collisions at all. Such hash functions are called ***perfect hashing***.

Let S be a set of keys, we say that a hash function h: $U \rightarrow M = \{0, 1, \cdots, m-1\}$ is a perfect hash function for S if h is injection on S, that is, there are no collisions among the keys in S if $\forall x, y \in S, x \neq y, h(x) \neq h(y)$ [10]. The Figure 3 (a) illustrates a perfect hash function concept with no collision. Any $k_i \in S$ can be retrieved from the hash table T by hash function h only once (single probe). If $n = (m-1)$, then the hash table size ($|T|$) is equal to the key size ($|S|$). We say that h is ***minimal perfect hash function*** of S as shown in Figure 3 (b). The minimal perfect hash functions are designed to totally avoid the wasted space and time problem. They are also widely applied for several applications where keys are static sets, such as words in natural languages, reserved words in programming languages, data mining and also network security.

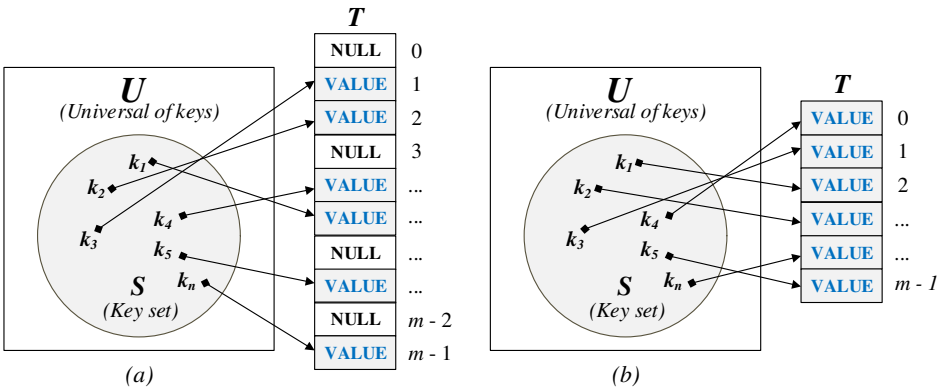

Figure 3: (a) Perfect hash function, and (b) Minimal perfect hash function

2.3 Sparse Matrix and Storage Formats

A ***sparse matrix*** is a matrix where the majority of elements have zero. In other words, the matrix which has a very few nonzero elements. Let A be a matrix and $A \in \mathbb{R}^{m\,x\,n}$, where $m, n \in \mathbb{N}_0$. We say that A is sparse if its number of nonzero entries is $O(min\{m, n\})$ [8, 26]. The following is example of a sparse matrix as shown in Figure 4.

$$A = m \begin{bmatrix} (0,0) & & n & & \\ 11 & 0 & 0 & 0 & 0 \\ 0 & 22 & 23 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 \\ 0 & 0 & 0 & 44 & 0 \\ 0 & 0 & 0 & 0 & 55 \end{bmatrix} \in \mathbf{R}^{m=5,\,n=5} \quad (4,4)$$

Figure 4: A sparse matrix (m = rows, n = columns) contains only 8 nonzero elements.

The sparse matrices are involved a large number of applications, especially in science and engineering. Basically, the sparse matrices are represented in the two-dimensional array. Thus, the capacity of memory allocated for a matrix is m x n x s, where s is the size of the data type (bytes) required to store the value. In the following sparse matrix A above, there are 5 rows (m) and 5 columns (n). We need to store the integer values, and then the memory consumption of A is 5 x 5 x 2 = 50 bytes; therefore, the space complexity is $O(m$ x $n)$. Representing the sparse matrices in array structures, each element in the array is presented by $A_{i,j}$ to access an address stored the data of the matrices. In general, i indicates to the row index, and j means the column index. To perform any operations on a sparse matrix, such as multiplying the elements, the time complexity will be $O(n^2)$, where $n = m$, because the operations that are executed on matrices need to operate in two nested loops. However, provided that the operation has known the index of i and j to access an element in the matrix, the worst-case access time will be $O(1)$.

Sparse matrix storage formats. Reduction of space and time complexity of a sparse matrix can be realized by collecting only the nonzero elements. The data structures for supporting this approach will be more complex to access the individual elements, and will be able to be restored to the original matrix properly [27, 33]. There are two groups of storage formats for storing sparse matrices: the first group is designed for efficiently modification, such as DOK (Dictionary of keys), LIL (List of lists), COO (Coordinate list) and so forth, and the second group is for access and matrix operations, such as CSR (Compressed Sparse Row) or CSC (Compressed Sparse Column). In this paper, we have showed the CSR format only, because of CSR is the most general format and widely used for implementing and referring. CSR places a set of nonzero (nz) of the matrix rows to contiguous memory spaces (array) which have the memory size to be $|nz|$. Suppose that we have a nonsymmetric sparse matrix $A \in \mathbb{R}^{m=5,x\,n=5}$ in Figure 4, and we then create three arrays for storing its: one of floating-point numbers (val), and two for integers (col_ind and row_ptr) as shown in Figure 5. The total size the arrays are: $val = |nz| = 8$, $col\_ind = |nz| = 8$ and $row\_ptr = m + 1 = 6$ respectively. The val maintains only nonzero elements of matrix A by collecting in row order. The col_ind array stores the column indexes of the elements in the val array, that is, if $val(k) = A_{i,j}$, then $col\_ind(k) = j$ as well. Last, row_ptr keeps the starting location of each row stored in val, that is, if $val(k) = A_{i,j}$, then $row\_prt(i) \leq k < row\_prt(i + 1)$. This storage format can save the space capacity to store the sparse matrix A from n^2 to 2 x $|nz| + m + 1$. In this example, it can be reduced from 50 bytes (5 x 5 x 2 bytes) to 44 bytes ((2 x 8 + 5 + 1) x 2 bytes). The time complexity of CSR to operate any operations such as the matrix-vector multiplication is $O(n$ x $m)$, where

$n = |val|$, $m = |row\_ptr|$.

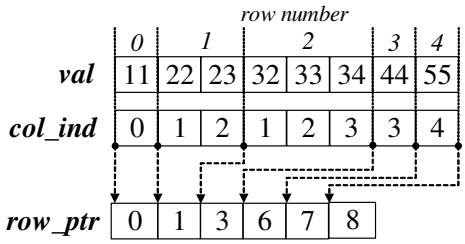

Figure 5: The CSR format for matrix A is specified by the arrays val, col_ind and row_ptr

Sparse matrix compression. To improve the memory space of the sparse matrix storage formats, Horowitz et al. [16] showed a simple and effective sparse matrix compression, called the nonzero-term method (NM). NM stores nonzero elements of a sparse matrix into a (row_index, column_index, value) format, for example, (1,2,4) means the row = 1, column = 2 and nonzero value = 4. Although, NM is easily for implementation, it takes the linear searching time against the nonzero elements. Ziegler's method (ZM) [32] is widely famous technique that applies shift-left and merging instructions to compress the nonzero values. In addition, rehash method (RM) [6] is to accomplish the sparse binary-matrix compression. This method is only applied with the binary number. Ji-Han Jiang et al. [17] extended the rehash method of RM by using the random hash function. Aiyoub Farzaneh et al. [9] contributed the Compressed Sparse Vector (CSV), which reduces the storage value of large non-symmetric sparse matrices more than CSR.

3 High-speed Firewall Structures

In this section, we depict the design of high-speed firewall structures. The high-speed firewalls in this paper mean firewalls that can match a packet against the predefined firewall rule with $O(1)$ worst-case access time. We represent four high-speed firewall structures: the policy mapping firewall (PMAP) [19], sparse matrix packing firewall (SMPF), perfect hashing firewall (PHF) and minimal perfect hashing firewall (MPHF) successively.

3.1 Key Contributions

We make four major contributions as follows:

1) Firstly, we optimize the memory space of the policy mapping (PMAP) proposed in [19] by applying the sparse matrix compression approaches, called SMPF;

2) Propose new techniques that are adapted from perfect hashing and minimal perfect hashing to improve the high-speed firewall rule verification, called PHF and MPHF;

3) Evaluate and compare the performance of all proposed techniques including PMAP, SMPF, PHF, MPHF, and also IPSet;

4) And finally, conclude the comparison results in several aspects.

3.2 High-speed Firewall Designing

There are six milestones to design high-speed firewalls:

Step 1. Designs a firewall rule user interface, in this step we choose the Rule-Base firewall (traditional style), because of it is popularly used nowadays as shown in Figure 6 in the Step 1;

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

Step 3. Maps the DSD from the Step 2 to the array structures by the policy mapping algorithm (PMAP);

Step 3.1. Packs the array structures from the Step 3 to SMPF by the sparse matrix compression technique;

Step 4. Creates keys and values from DSD to build the perfect hashing firewall (PHF);

Step 4,1. Compacts the perfect hashing tables from the Step 4 to the minimal perfect hashing firewall (MPHF).

According to the steps of high-speed firewall design, we thoroughly describe each step like this:

Step 1: Designing the firewall user interface.
Nowadays, almost all firewall user interfaces are Rule-Base or Rule-List. The interfaces are displayed in a tuple format compounded from SIP, DIP, SP, DP, Pro and Act as {192.168.1.0-255, *.*.*.*, 1234, 80, *, a}. They have been influenced by the nature of reading and writing from left-to-right and top-to-bottom. Other user interface aspects, several researchers tried to suggest new firewall interfaces like [23] or [14]; however, they were not popular. Therefore, we still use the Rule-Base interface to make firewall rules in the Step 1. In order to easily describe firewall structures, we have presented easy firewall rules that consist of five fields: SIP, DIP, DP, Pro and Act as shown in Table 3. We also transform both SIP and DIP from the IPv4 addressing format to the positive decimal format by the equation to as $octate_4$ x 2^{24} + $octate_3$ x 2^{16} + $octate_2$ x 2^8 + $octate_1$ x 2^0. An IP address 0.0.0.10, for example, is transformed to 0_4 x 2^{24} + 0_3 x 2^{16} + 0_2 x 2^8 + 10_1 x 2^0 = 10.

Step 2: Building the DSD.
The decision state diagram (DSD) is built from the firewall decision state diagram algorithm (FDSD)

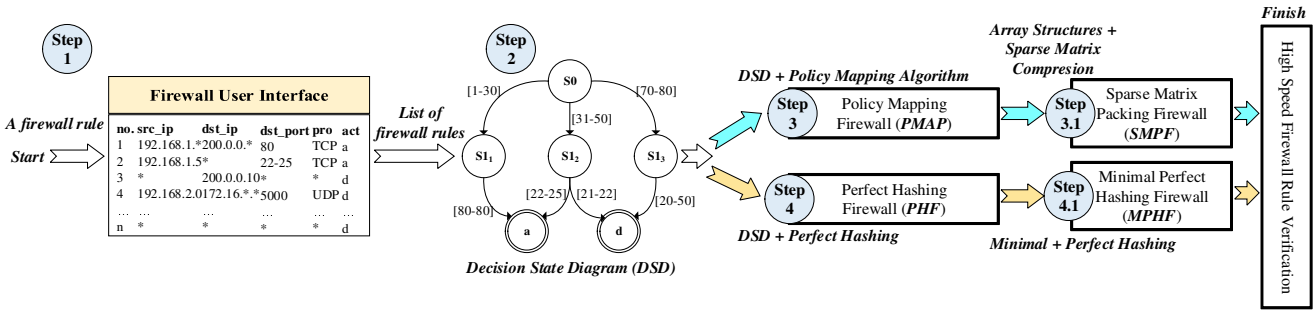

Figure 6: Steps of high-speed firewall designing

Table 3: Easy firewall rules for building DSD

No.	*SIP*	*DIP*	*DP*	*Pro*	*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***

proposed by Khummanee et al. [19]. It has several roles, for example, it makes sure that the firewall decision paths are not any confusion, and eliminates decision paths duplicated in firewall rules. The details on how to build the DSD is described in [19]. In this paper, we only show the final decision state diagram which was created from firewall rules in Table 3 successfully as shown in Figure 7. Referring to the DSD in Figure 7, suppose that there is a packet arriving from somewhere to our networks. It consists of DP = 25, DIP = 0.0.0.55, SIP = 0.0.0.10 and Pro = TCP. Thus, the firewall makes a decision to allow this packet into the networks because it follows the 1^{st} state decision path of the DSD diagram (25 $\in$ {25-30}, 55 $\in$ {50-60}, 10 $\in$ {1-15} and TCP $\in$ {TCP, UDP} $\rightarrow$ a). Another example, if a packet is composed of DP = 80, DIP = 0.0.0.10, SIP = 0.0.0.45 and Pro = TCP. This packet thereby will be dropped by the 3^{rd} path of DSD.

Step 3: Mapping DSD to array structures.

In this step, we map the DSD from the Step 2 to arrays. The DSD has four levels are DP, DIP, SIP and Pro respectively. The DP level (the 1^{st} level) in Figure 7 is mapped to one dimension array, named S0-S1 as shown in Figure 8. The algorithm for mapping DSD to array structures is illustrated in [19]. The memory size of S0-S1 equals to 65,536 elements. Each element has 16-bits integer; therefore, the total of memory size of the S0-S1 array is $\approx$ 131 kilobyte (KB). The 2^{nd} level (DIP) is transformed to the three-dimensional array of two cubes namely S1-S2$_{upper}$ and S1-S2$_{lower}$. The S1-S2$_{upper}$ is used to store $octate_3$ (x-axis) and $octate_4$ (y-axis), and the S1-S2$_{lower}$ collects $octate_1$ (x-axis) and $octate_2$ (y-axis) of DIP. The z-axis of both three-dimensional arrays is referred to the decision state path (state path number) of DSD. The 3^{rd} level (SIP) is as similar as DIP. The algorithm converts SIP level to S2-S3 arrays, that is, S2-S3$_{upper}$ maintains $octate_3$ and $octate_4$, and S2-S3$_{lower}$ collects $octate_1$ and $octate_2$. The total of maximum memory usage of both S1-S2 and S2-S3 is $\approx$ 16.97 GB (x-axis_size $\times$ y-axis_size $\times$ data_size $\times$ state_path_size = 256 $\times$ 256 $\times$ 16 $\times$ 65,536 bit) to deal with 65,536 state paths of the firewall rule. The final state level of DSD (4^{th}) is Pro, it is mapped to two dimensional array, namely S3-S4. The total size of S3-S4 is about $\approx$ 33.55 MB. The x-axis of S3-S4 array is used to store the decision (Act) of the firewall rule state path, and y-axis points to the firewall rule state path.

For example, the 4^{st} state path of DSD diagram in Figure 7, DP is equal to {80-80}, it means a destination port number 80. Thus, S0-S1[80] is set to be 3 (the state path in DIP level). DIP is subset of {20-30}, it indicates the range of destination IP addresses between 0.0.0.20 and 0.0.0.30. Consequently, S1-S2[0][0][3]$_{upper}$ is equal to $'X'$ (X = don't care term), S1-S2[20-30][0][3]$_{lower}$ are assigned to be 4 (state path of SIP level). Likewise, the SIP is subset of source IP addresses ranging from 0.0.0.1 to 0.0.0.9 ($SIP \in$ {1-9}). Therefore, S2-S3[0][0][4]$_{upper}$ is stored $'X'$, and S2-S3[1-9][0][4]$_{lower}$ keep the positive integers (the number 4) that point to an array stored protocols in S3-S4. Lastly, the 4^{th} state path of Pro level is the subset of both TCP and UDP, so S3-S4[4][6] (TCP) and S3-S4[4][17] (UDP) are assigned to be $'d'$ (deny) as shown in Figure 8. The sum of all memory space used to support the number of 65,536 rules is about 17 GB.

Step 3.1: Packing array data structures.

Notice that the stored values in array data structures in Figure 8 after finishing the Step 3 are similar to the sparse matrices discussed in Section 2.3. Furthermore, the total memory size of the arrays is not optimal yet. To optimize the memory usage of arrays, we have applied the sparse matrix

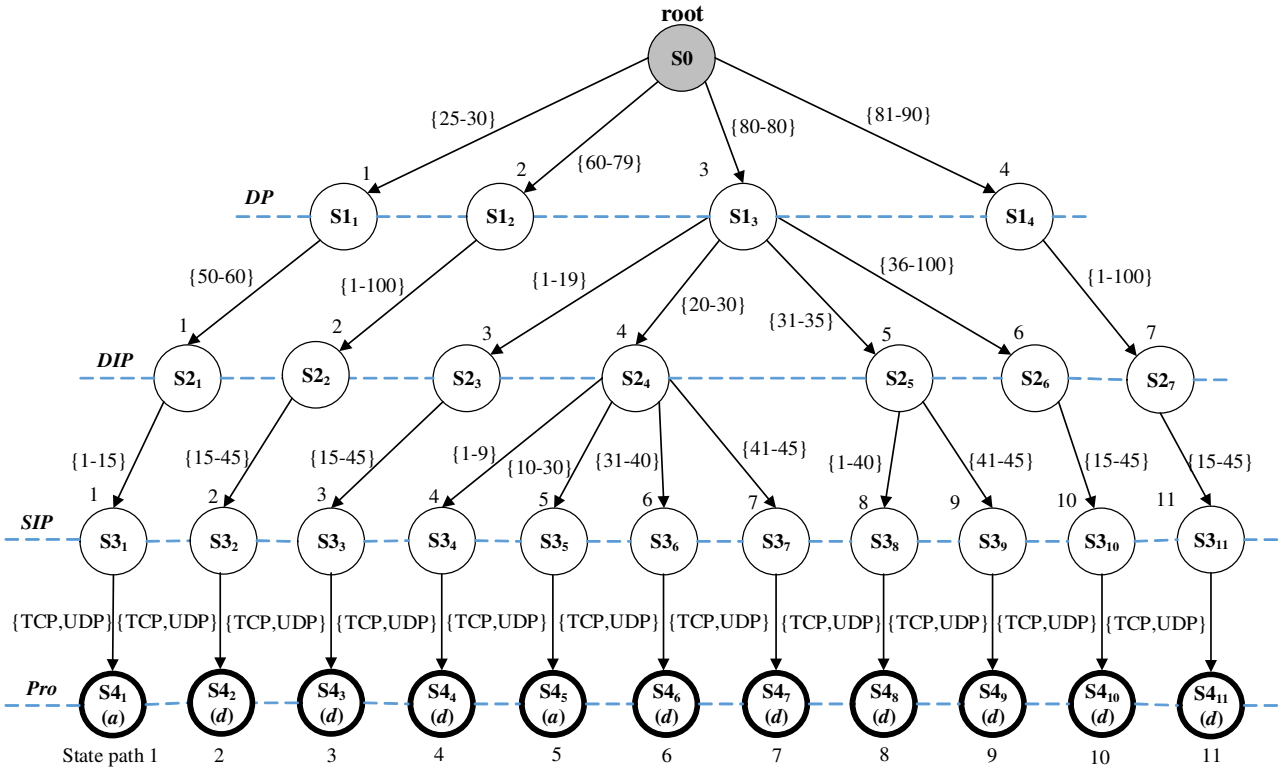

Figure 7: The final decision state diagram (DSD)

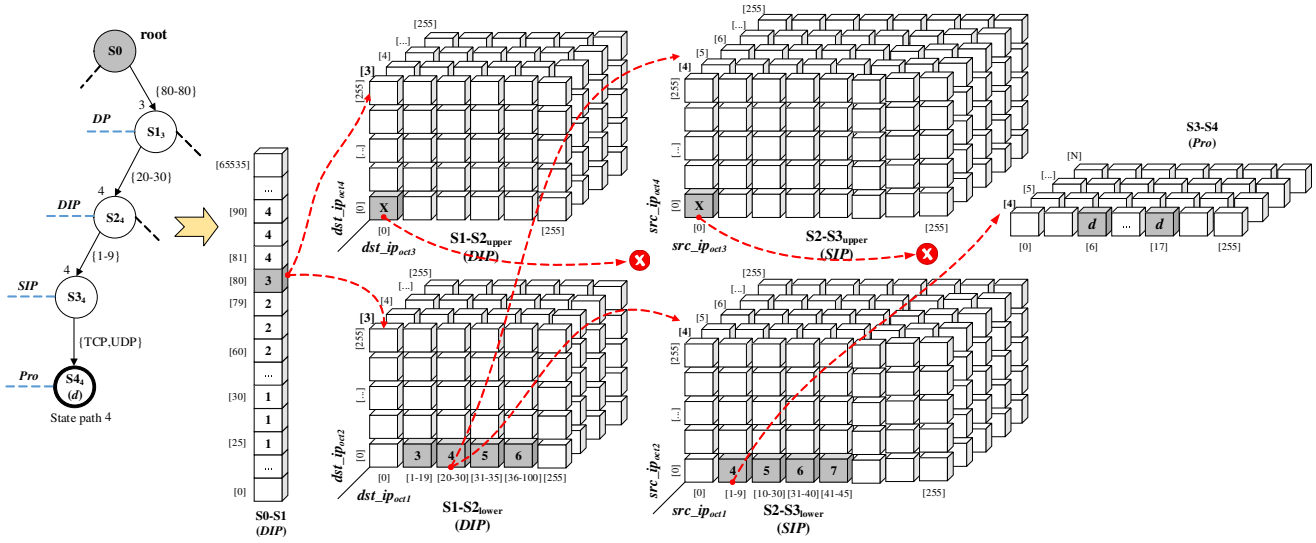

Figure 8: Mapping the 4^{th} state path of DSD to array structures

compression techniques to reduce them (also optimize PMAP), referred to as Sparse Matrix Packing Firewall (SMPF). The concepts of the sparse matrix packing are adapted from [5, 2]; however, they have limitations for directly deploying with the firewall rule. For example, they only support two-dimensional arrays, and do not appropriate for firewall's array data structures which are the three-dimensional array. Besides, they are tested on the small sparse matrices, but sparse matrices for the firewall rules are very large. Generic sparse matrices can freely swap any positions of rows, but the rows in sparse matrices of firewall rule cannot arbitrarily change because the matching may be wrong as a result. In order for the sparse matrix to be able to perfectly handle firewall rule structures, we have improved the mechanisms to store and pack sparse matrices as shown in Algorithm 1, referred to as the Sparse Matrix Packing (SMP).

Algorithm 1 Sparse matrix packing algorithm (SMP)

Input: S1-S2$_{upper,lower}$, S2-S3$_{upper,lower}$, and S3-S4
Output: RLT$_{S1-S2_{upper,lower}}$, RLT$_{S2-S3_{upper,lower}}$, RLT$_{S3-S4}$ and 1-D$_{S1-S2,S2-S3,S3-S4}$
set size = 256, path = N *(size = array size, $N \in \mathbb{N}_1$)*
create 1-D$_{S1-S2,S2-S3_{upper}}$,1-D$_{S1-S2,S2-S3_{lower}}$=[ ]
create RLT$_{S1-S2,S2-S3_{upper,lower}}$ = [path][size]
create 1-D$_{S3-S4}$ = [ ], RLT$_{S3-S4}$ = [path]
set pc, rc = 1 *(pc=page count,rc=row count)*
while path $\leq$ pc **do**
  while rc $\leq$ size **do**
    read data from S1-S2[rc][*][pc]$_{upper}$
    append data to 1-D$_{S1-S2_{upper}}$ consecutively
    add 1^{st} index of data to RLT[pc][rc]$_{S1-S2_{upper}}$
    read data from S1-S2[rc][*][pc]$_{lower}$
    append data to 1-D$_{S1-S2_{lower}}$ consecutively
    add 1^{st} index of data to RLT[pc][rc]$_{S1-S2_{lower}}$
    read data from S2-S3[rc][*][pc]$_{upper}$
    append data to 1-D$_{S2-S3_{upper}}$ consecutively
    add 1^{st} index of data to RLT[pc][rc]$_{S2-S3_{upper}}$
    read data from S2-S3[rc][*][pc]$_{lower}$
    append data to 1-D$_{S2-S3_{lower}}$ consecutively
    add 1^{st} index of data to RLT[pc][rc]$_{S2-S3_{lower}}$
    read data from S3-S4[rc][pc]
    append data to 1-D$_{S3-S4}$ consecutively
    add 1^{st} index of data to RLT[pc][rc]$_{S3-S4}$
    rc++
  end while
  pc++
end while
End

The SMP starts with: (1) reading 3-D or 2-D sparse matrices from array data structures of the firewall rule as shown in Figure 8, (2) packing them to 1-D arrays, and (3) recording the indexes pointed to the packed 3-D or 2-D data in *Row Lookup Table* (*RLT*) used later to access the 1-D array during retrieval. The 1-D array stores non-zero items of the packed 3-D or 2-D array while the RLT is held at the starting position of each row stored in 1-D array. Packing begins by separating the 3-D or 2-D array into the tuples or records. This process is shown in Figure 9. In Figure 9 (a), suppose that the page number 4 (the stat path number 4 in *SIP* level of DSD) of the 3-D array is first packed into the 1-D array. The 1^{st} packed record is S2-S3[0][*]$_{lower}$ (S2-S3[*row*][*column*], * = all columns), 2^{nd} packed record is S2-S3[1][*]$_{lower}$, and final packed record is S2-S3[255][*]$_{lower}$ respectively. The page number 4 of S2-S3$_{lower}$ has only a single row that has non-zero items between the location 1 and 45. These non-zero items are placed into the first location of 1-D array, this process is shown in Figure 9 (b). To retrieve the non-zero items in 1-D array later, the RLT marks how they were packed by recording the integer position for each row of the original 2-D arrays into the RLT. For example, the value -1 in the 4^{th} row ([4]) of RLT table indicates to the 1^{st} row in the page no. 4 of S2-S3$_{lower}$ array (S2-S3[0][*][4]$_{lower}$) because that is the offset for replacing the non-zero items ranging from 1 to 45 (starting free slot in 1-D array - starting non-zero item list $\rightarrow$ 0 - 1 = -1). As a result, the starting free slot of 1-D array is shifted from the position 0 to 44 immediately. The next non-zero items allocated in the page no. 5 ([5]), there are ranging from the position 1 to 45. The packing process places them to 1-D array in the position 45 which is currently pointed to the starting free slot index, then the index is updated to the location 90 (45 + 45). The offset of the page no. 5 in 5^{th} row in RLT is also recorded to 44 (starting free slot - starting non-item list $\rightarrow$ 45 - 1 = 44). The last example of the packing process, the list of non-zero items in page no. 6, is arrayed from 15 to 45 (30 positions). This list is allocated in the position 90 to 120 in the 1-D array, the starting free slot index is updated to 121, and the offset in RLT of this list (6^{th} row) is set to 75 (90 - 15). On the other hand, the rows that have not non-zero items are set to the blank record in RLT table like the 1^{st}, 2^{nd}, 3^{rd} row and so forth. To retrieve any non-zero items in the 1-D array, we use the formula as following:

Offset = RLT[Page No.][Row No.];
Index = Offset + Column;
Non-zero item = 1-D[Index]

For example, suppose that we would like to retrieve the non-zero item of the S2-S3$_{lower}$ array by the row = 0, column = 41 and page no. 5 in 1-D array. Thus, we compute the position of this item as Offset = RTL[5][0] = 44, Index = Offset + Column = 44 + 41 = 85, and then the non-zero item = 1-D[85]$_{S2-S3_{lower}}$ = 9 as shown in Figure 9. The SMP algorithm matches the firewall rules against any packet$_i$ shows in Algorithm 2.

Step 4: Building the perfect hashing firewall.

IPSet [24] running on IPTables [29] and Netfilter was successfully applied the perfect hash function to improve the speed of firewall rule verification. It offers

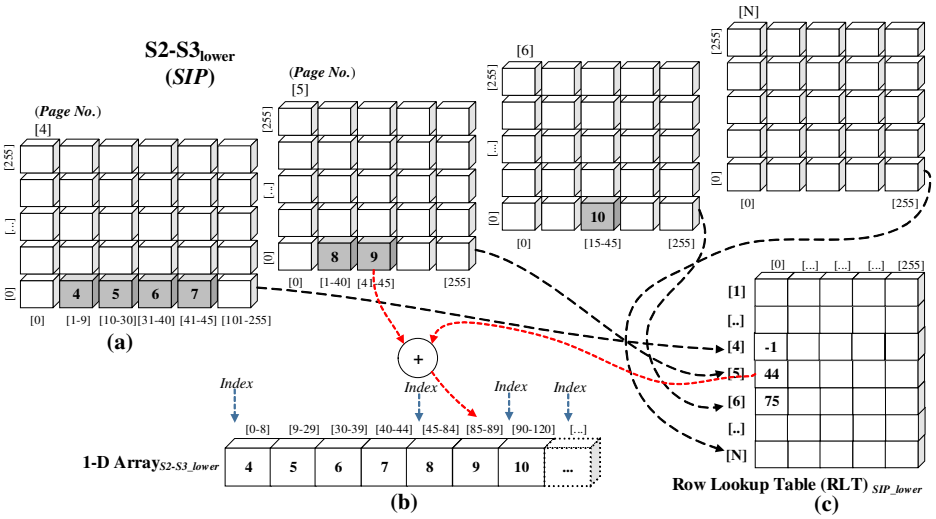

Figure 9: (a) separating the 3-D array of SIP namely S2-S3$_{lower}$ to records, (b) packing records to 1-D array, and (c) adding the starting position of each packed record to RLT

Algorithm 2 Sparse matrix matching

Input: S0-S1, RLT$_{S1-S2_{up,lo}}$, RLT$_{S2-S3_{up,lo}}$, RLT$_{S3-S4}$ and 1-D$_{S1-S2,S2-S3,S3-S4}$, packet p_i
Output: a = accept or d = deny
get DP, DIP, SIP, $Pro \leftarrow p_i$
split DIP_{oct_1}, DIP_{oct_2}, DIP_{oct_3}, $DIP_{oct_4} \leftarrow DIP$
split SIP_{oct_1}, SIP_{oct_2}, SIP_{oct_3}, $SIP_{oct_4} \leftarrow SIP$
page $\leftarrow$ S0-S1$[DIP]$
offset $\leftarrow$ RTL[page]$[DIP_{oct_2}]_{S1-S2_{lower}}$
index $\leftarrow$ offset + DIP_{oct_1}, p1 $\leftarrow$ 1-D[index]$_{S1-S2_{lower}}$
offset $\leftarrow$ RTL[page]$[DIP_{oct_4}]_{S1-S2_{upper}}$
index $\leftarrow$ offset + DIP_{oct_3}, p2 $\leftarrow$ 1-D[index]$_{S1-S2_{upper}}$
page $\leftarrow$ get value from p1 or p2 that is not 'X'
offset $\leftarrow$ RTL[page]$[SIP_{oct_2}]_{S1-S2_{lower}}$
index $\leftarrow$ offset + SIP_{oct_1}, p1 $\leftarrow$ 1-D[index]$_{S2-S3_{lower}}$
offset $\leftarrow$ RTL[page]$[SIP_{oct_4}]_{S1-S2_{upper}}$
index $\leftarrow$ offset + SIP_{oct_3}, p2 $\leftarrow$ 1-D[index]$_{S2-S3_{upper}}$
page $\leftarrow$ get value from p1 or p2 that is not 'X'
offset $\leftarrow$ RTL[page]$_{S3-S4}$, index $\leftarrow$ offset + Pro
result $\leftarrow$ 1-D[index]$_{S3-S4}$
if result == $'a'$ **then**
  print $'accept'$
else
  print $'deny'$ *(result == 'd')*
end if
End

several features of the keys which are used for referring to the hashing table, for instance, the key is combined by the IP address against port number (IP:Port) or IP address and port and IP address (IP:Port:IP) or etc. In this section, we have showed how to apply the perfect hash function as same as IPSet, but we have chosen the different key aspect by acquiring the key from DSD instead. With the acquisition of keys from DSD, we have picked the remarkable features which are combined to the unique key. The 3^{rd} state path of DSD diagram in Figure 7, for example, we have jointed the port number (DP), destination IP address (DIP) and source IP address (SIP) to be the key such as $'801015'$ (DP = 80, DIP = 10, SIP = 15). Yielding to the number of keys in each state path can be computed from the number of $DPs \times DIPs \times SIPs$; therefore, the 3^{rd} state path in Figure 7 is $1 \times 19 \times 30 = 570$ keys ($'80115'$, $'80216'$, $'80317'$, $\cdots$, $'801945'$). Notice that the number of keys which will be hashed by a perfect function (h) are huge. IPSet also faces such problem; as a result, it requires a set of rules that do not exceed over the subnet of IP class C only. To avoid this problem, we have reduced the number of keys by choosing for each state path where the action (Act) is an acceptation (a) only. For example, we have only chosen the 1^{st} and 5^{th} state path from all paths in Figure 7 to be the keys.

The keys got from the DSD will be hashed by the perfect hash function h for referring to any address storing a value. We have applied the hashing algorithm from [4, 11] as shown in Algorithm 3, the algorithm uses two levels of hash functions. The first function is $\boldsymbol{h}_1(0, key)$, it gets a position in an intermediate array, named G. The second function is $\boldsymbol{h}_2(d, key)$, it hashes the key and the information got from G to search the unique position for the key as shown in Figure 10. For example, both of $'801235'$ and $'256811'$ key are hashed to the same position (1234) in the intermediate table (G) by using function $\boldsymbol{h}_1$. However, the second hash function $\boldsymbol{h}_2$ hashes the keys again by combining the same position against old keys, and puts hashing results into different slots in value table

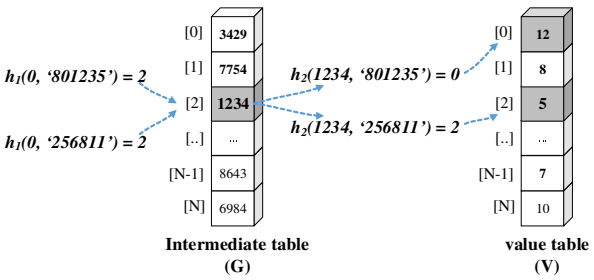

Figure 10: Perfect hash table lookup

(V) so no collision occurs. To avoid the collision from hashing, we use the FNV [25] hash function (Algorithm 4) which guarantees the very low collision rate. To look up a value in hash table G and V, we first get the DP, DIP and SIP field from a packet, and compound them to the key. For example, suppose that a packet$_i$ consists of $DP = 80$, $DIP = 200$, $SIP = 100$ and $Pro =$ TCP, so the compounded key is $'80200100'$. We can compute a position stored the value from the key by Algorithm 5.

Algorithm 3 Perfect hash function (double hash h_1, h_2)

```
1: Input: dictionary (dict) of key and value ('key':value)
2: Output: G, V (G=intermediate, V=values table)
3: set size ≫ |dict|
4: create bucket[size][], G[size], V[size]
5: while key_i ← read dict until NULL do
6:    p ← FNV-hash(0, key_i) mod size
7:    bucket[p].append(key_i) #put all keys to buckets
8: end while
9: while key ← read bucket[d][*] until NULL do
10:   if |key| == 1 then
11:      put 0 → G[d] (d ∈ ℕ0)
12:      put value from dict(key) → V[d]
13:   end if
14: end while
15: while keys ← read bucket[d][*] until NULL do
16:   if |keys| > 1 then
17:      d = 1
18:      while key ← read keys until NULL do
19:         p ← FNV-hash(d, key) mod size
20:         if put d → G[p] then collision then
21:            p ← rehash(d++, key) until no collision
22:         end if
23:         V[p] ← read value from dict(key), G[p] ← d
24:      end while
25:   end if
26: end while
27: End
```

Step 4.1: Building the minimal perfect hashing firewall. The perfect hashing is a method which guarantees no collision upon building a hash table. It is only possible to build it when we have known all of the keys in advance. Besides, the space for storing values of perfect hashing is usually more than the total size of values ($|storage| \gg |values|$) because it can reduce the percentage of collision rate. One effective method for reducing the size of memory space is the ***Minimal Perfect Hashing*** (MPH). MPH ensures that the hashing table contains one key per one slot only, and also has no free slots. In this section, we have optimized the memory space of the perfect hashing described in previous section by customizing Algorithm 3 in the line of code 3 from $set\,size \gg |dict|$ to be $set\,size = |dict|$ instead. This code determines the size of the memory to be fitted with the number of keys. The rest of the codes are as same as Algorithm 3.

Algorithm 4 The FNV algorithm

```
1: Input: d, key
2: Output: d
3: if Input has both d and key then
4:    while c ← read a char from key until NULL do
5:       d ← ((d ∗ 0x01000193) ^ ord(c)) & 0xffffffff
6:    end while
7: else
8:    d ← 0x01000193
9: end if
10: End
```

Algorithm 5 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
```

3.3 High-speed Firewall Implementation

In this section, we detail about implementing high-speed firewalls. The tested environments of this paper are similar to [19], but they are slightly different as follows:

Hardware and Software Development Tools. The high-speed firewalls are developed on the Intel 64-bits processor, Core i7, 2GHz, installed memory (RAM) 8 GB. For the developing software, we chose Python language (version 3.4 for 64 bits), Numpy and Psutil to implement firewalls running on MS Windows 8 operating system.

Firewall Rule and Packet Generator.
In each test case, the firewall rule generator generates the randomness rules from 1,000, 2,000, 3,000, 4,000, 5,000 and 10,000, and the random packets for 10,000 packets per round by the packet generator software. We have experimented about 30 times per each algorithm, and calculated the average of speed and space for each algorithm.

4 The Performance Evaluation

All proposed firewalls are high-speed. Therefore, they can match any rules against packet$_i$ by very low constant time; in other words, they take $O(1)$ worst-case access time. However, their memory and time consumption for constructing rule structures are not equal depending on the complexity of firewall rule data structures. Thus, this paper we aim to compare the amount of memory space used for storing rule structures and the processing time for building them. In case of the computation time, there are three kinds: the time for building the decision state diagram (DSD), constructing rule structures, and matching firewall rule as shown in Table 4. The space complexity and percentage of memory space optimized are shown in Table 5.

The speed of firewall rule verification of all techniques is similar, for example, in Table 4, the average of time verifying of all approaches is quite stable at about 0.031 seconds on average. Notice that while processing the Path No. 10,000, the PMAP consumes the verifying time more than another algorithms by about 0.124 seconds because the main memory is not enough (running memory > available memory) for supporting PMAP running. As a result, the Python interpreter needs to allocate an extra memory from the virtual memory (usually the hard disk drive) to accomplish PMAP process. According to the constructing time, MPHF spends the most time on the firewall rule construction ≈ 3,983 seconds at the number of state paths (Path No.) = 5,000. Moreover, it cannot be executed successfully at the Path No. 10,000 because it takes too much constructing time, and Python interpreter sometimes crashes. The reason that it is slow is repeated searching of a free slot in a limited hash table to place the value. In contrast, PMAP is the fastest technique to build the rule data structures; on the other hand, it also consumes the most of memory usage. SMPF and PHF take the average amount of rule constructing time in the same trend by the linear manner.

In case of the memory optimization, SMPF is the best high-speed firewall to consume the minimal space stored structural rules, it can highly reduce the percentage of memory usage of PMAP by around 99.9 percent on average. While MPHF is able to optimize the memory of PMAP by about 87.7%, and PHF by about 62.3% respectively, as shown in the column of packing results of Table 5. The Keys column in Table 5 is presented the number of hashed keys by the perfect hash function of PHF and MPHF. To process the number of the state paths at 10,000, PHF and MPHF must hash the number of keys ≈ 83.4 million.

5 Conclusions and Future Work

In this paper, we have proposed several techniques to improve the high-speed firewalls that can access the data in $O(1)$ worst-case access time, namely PMAP, SMPF, PHF and MPHF. They are as fast as IPSet [24] but different in the limitations as following:

Limitations: *IPSet* needs to be set up a group of rules in the network class C only before running, and each group is not bigger than 65,536 rules per a set. It does not support the IP network class A, but we can partition them to subnets before deploying to IPSet. It is not easy to understand, the administrator must have a lot of skills about the set of rules. *PMAP* consumes a lot of memory to build the rule data structures, it only supports a maximum number of 65,536 rules. *SMPF* supports the same number of firewall rules like PMAP, and the firewall structure is quite complicated. *PHF* encounters a lot of keys effecting the firewall's performance, it should solve this problem like IPSet. *MPHF* has drawbacks like PHF, but the big problem is the time to construct firewall rules data structures.

We have concluded the overall performance and limitations of high-speed firewalls in Table 6.

Table 6: The overall performance of high-speed firewalls

Name	Space	Time complexity		Structural
		Verify	Construct	
PMAP	Fair	O(1)	Fastest	More
SMPF	Best	O(1)	Fast	Most
PHF	Good	O(1)	Faster	Much
MPHF	Better	O(1)	Slow	Much
IPSET	Good	O(1)	Faster	Much

Space = Space complexity, Verify = Verification time, Construct = Construction time, and Structural = Structural complexity

References

[1] H. B. Acharya and M. G. Gouda, "Linear-time verification of firewalls," in *International Conference on Network Protocols (ICNP'09)*, pp. 133–140, Princeton, NJ, Oct 2009.

[2] M. Aspns, A. Signell, and J. Westerholm, *Efficient Assembly of Sparse Matrices Using Hashing*. Berlin: Springer Berlin Heidelberg, 2007.

[3] M. Bellion, "High performance packet classification (HIPAC)," 2005. (`http://www.hipac.org/`)

Table 4: Time complexity results of high-speed firewals

Path No.	Constructing time (sec)					Verifying time (sec)			
	DSD	PMAP	SMPF	PHF	MPHF	PMAP	SMPF	PHF	MPHF
1,000	1.60	0.59	41.12	20.37	243.51	0.031	0.031	0.032	0.015
2,000	6.51	1.17	88.27	39.27	628.73	0.031	0.015	0.032	0.015
3,000	14.84	1.81	141.74	58.38	1477.16	0.031	0.015	0.033	0.016
4,000	26.17	2.50	204.57	77.37	1763.36	0.015	0.031	0.032	0.031
5,000	50.95	3.37	282.79	100.28	3,983.51	0.031	0.031	0.034	0.034
10,000	188.83	38.29	630.90	506.51	N/A	0.124*	0.032	0.034	N/A

Note: Path No. = the number of firewall rule state paths, DSD = decision state diagram, PMAP = policy mapping, SMPF = sparse matrix packing, PHF = perfect hashing and MPHF = minimal perfect hashing firewall, N/A = consumes too much processing time, * = main + virtual memory access time

Table 5: Space complexity and optimized memory results of high-speed firewalls

Path No.	Constructing space (MB)				Packing result (%)			Keys (million)
	PMAP	SMPF	PHF	MPHF	SMPF	PHF	MPHF	PHF&MPHF
1,000	1,043.38	0.58	402.65	124.20	99.94	61.40	88.09	7,762,577
2,000	2,085.67	0.65	805.30	255.92	99.96	61.38	87.72	15,995,188
3,000	3,113.80	0.72	1,207.95	365.99	99.97	61.20	88.24	22,874,422
4,000	4,135.63	0.79	1,610.61	534.10	99.98	61.05	87.08	33,381,606
5,000	5,163.76	0.86	2,013.26	634.31	99.98	61.01	87.71	39,644,683
10,000	10,095.22	1.18	3,221.22	N/A	99.98	68.09	N/A	83,447,420

Note: Packing result = percentage of firewall rule compression from PMAP, Keys = the number of hashed keys, N/A = cannot build rule structures resulting from Table 4.

[4] F. C. Botelho and N. Ziviani, "External perfect hashing for very large key sets," in *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management (CIKM'07)*, pp. 653–662, Lisboa, Portugal, Nov 2007.

[5] M. D. Brain and A. L. Tharp, "Perfect hashing using sparse matrix packing," *Information Systems*, vol. 15, no. 3, pp. 281–290, 1990.

[6] C. C. Chang, J. H. Jiang, and T. S. Chen, *Rehash method: A compression technique of sparse binary-matrices*. National Chung Cheng University, Chiayi, Taiwan: Technical Report of the Department of Computer Science and Information Engineering, 1996.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms Third Edition*. Massachusetts Institute of Technology: The MIT Press, 2009.

[8] S. Debasis, *Classic Data Structures*. Delhi, India: PHI Learning; 2nd edition edition, 2009.

[9] A. Farzaneh, H. Kheiri, and M. A. Shahmersi, "Fundamental study perfect hashing," *Theoretical Computer Science*, vol. 182, no. 1, pp. 1–143, 1997.

[10] A. Farzaneh, H. Kheiri, and M. A. Shahmersi, "An efficient storage format for large sparse matrices," *Commun.Fac.Sci.Univ.Ank.Series A1*, vol. 58, no. 2, pp. 1–10, 2009.

[11] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud, "Practical minimal perfect hash functions for large databases," *Communications of the ACM*, vol. 35, no. 1, pp. 105–121, 1992.

[12] M. G. Gouda, A. X. Liu, and M. Jafry, "Verification of distributed firewalls," in *IEEE Global Communications Conference (GLOBECOM'08)*, pp. 1–5, New Orleans, LO, Nov-Dec 2008.

[13] 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.

[14] X. He, T. Chomsiri, P. Nanda, and Z. Tan, "Improving cloud network security using the tree-rule firewall," *Future Generation Computer Systems*, vol. 30, pp. 116–126, 2013.

[15] T. Heinz, "Hipac high performance packet classification for netfilter," 2004. (`http://www.net.t-labs.tu-berlin.de/papers/H-HiPAC-04.pdf`)

[16] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*. Potomac, Maryland: Computer Science Press, 1976.

[17] J. H. Jiang, C. C. Chang, and T. S. Chen, "A compact sparse matrix representation using random hash functions," *Data and Knowledge Engineering*, vol. 32, no. 1, pp. 29–49, 2000.

[18] 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, Maha Sarakham, May 2013.

[19] S. Khummanee and K. Tientanopajai, "The policy mapping algorithm for high-speed firewall policy ver-

ifying," *International Journal of Network Security*, vol. 18, no. 3, pp. 433–444, 2016.

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

[21] A. X. Liu and M. G. Gouda, "Structured firewall design," *Computer Networks Journal*, vol. 51, no. 4, pp. 1106–1120, 2007.

[22] 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.

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

[24] Netfilter, "IPSET," 2015. (`http://ipset.netfilter.org/index.html`)

[25] Landon Curt Noll, "Fnv hash," 2015. (`http://isthe.com/chongo/tech/comp/fnv/`) 2015.

[26] S. Pissanetsky, *Sparse Matrix Technology*. San Diego, CA: Academic Press Inc, 1984.

[27] M. Roberts, "Sparse matrix compression formats," 2015. (`http://www.cs.colostate.edu/mcrob/toolbox/cpp/sparseMatrix/sparse-matrix-compression.html`)

[28] D. Rovniagin and A. Wool, "The geometric efficient matching algorithm for firewalls," *IEEE Transactions on Dependable And Secure Computing*, vol. 8, no. 1, pp. 147–159, 2011.

[29] R. Russell, "IPTables," 2015. (`http://ipset.netfilter.org/iptables.man.html`) 2015.

[30] C. A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis Third Edition*. Blacksburg, USA: Virginia Tech, 2010.

[31] J. M. Stewart, *Network Security, Firewalls, And Vpns*. Burlington: Jones and Bartlett Learning, 2014.

[32] J. van Leeuwen, *Handbook of Theoretical Computer Science*. Cambridge MA: The MIT Press, 1990.

[33] Wikipedia, "Sparse matrix," 2015. (`https://en.wikipedia.org/wiki/Sparsematrix`)

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 lecturer 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.