**ECTI Transactions on Computer and Information Technology**
Journal homepage: https://ph01.tci-thaijo.org/index.php/ecticit/
Published by the ECTI Association, Thailand, ISSN: 2286-9131

# FFF: Fast Firewall Framework to Enhance Rule Verifying over High-speed Networks

Suchart Khummanee[1] Panida Songram [2] and Potchara Pruksasri[3]

## ABSTRACT

The current traffic trend on computer networks is growing exponentially, affecting network firewalls because they constantly have to filter out massive amounts of data. In this paper, we implement a firewall framework to improve traffic processing speed, named the Fast Firewall Framework (FFF or F$^3$). FFF can verify rules at Big-O(1) worst-case access time, and it also consumes a small amount of memory, which is only Big-O($n_{bit}$). To evaluate the firewalls' effectiveness, we benchmark the proposed firewall framework against the two fastest open source firewalls, IPSets and IPack. The experimental results show that FFF can execute rules faster than both firewalls and consumes less memory. In addition, the proposed firewall framework has a simple structure that makes it easier to implement.

*Article information:*

**Keywords:** Firewall rule verification, High-speed firewall, Fast packet matching

**Article history:**
Received: July 18, 2021
Revised: August 14, 2021
Accepted: September 24, 2021
Published: March 5, 2022 (Online)

**DOI:** 10.37936/ecti-cit.2022161.246990

## 1. INTRODUCTION

Nowadays, computer networks must be able to handle enormous amounts of information being communicated. Most of the information being communicated is urgent, such as online meetings, real-time streaming, online games, etc. The data must always be verified and filtered by a firewall to determine it is safe or not before sending it to users. Data communication is stable when the firewall can process data at high speed and continuity. Typically, firewalls are installed as a front-end barrier to protect users between private networks and the Internet by ensuring traffic is strictly following the rules. A rule is made up of a set of conditions for commanding the firewall software. The rule is basically comprised of two parts: a conditional and a decision (action) part. There are generally five parts of the condition: source IP address (SIP), destination IP address (DIP), source port (SP), destination port (DP), and protocol (Pro). If the data matches all condition fields in a firewall rule, the firewall must immediately decide between accept (allow) and deny (reject) in the decision-making section of the rule. Conventional firewall rules are executed in sequence from the first ($r_1$) to the last rule ($r_6$) as shown in Table 1. In example of the $r_1$ in Table 1, the firewall allows (action = *accept*) data communication from a group of 256 source devices (SIP = 192.168.1.0-255) to a group of 256 destination devices (DIP = 100.50.0.0-255) over a group of destination ports ranging from 80 - 90 (DP). If the firewall is unable to match the incoming data to rule $r_1$, then it is automatically shifted down to the next rule ($r_2$). In this example, $r_2$ rejects (*deny*) all HTTP connections (DP = 80) from a pool of thirty-one source IP addresses (192.168.1.20-50) to a hundred of the destination web servers (100.50.0.100-200). When the incoming data is not matched against any rules, it is implicitly dropped by the last firewall rule. For example, the last rule ($r_6$) in Table 1 drops every communication channel (DP and Pro = all) from any source IP address (SIP = all) to any destination IP address (DIP = all).

The time complexity to verify firewall rules, as described above, is equal to $O(n)$ (sequential execution), where $n$ is the number of firewall rules. A firewall that provides such a method of verification is known as a traditional firewall. Typical examples include Windows Defender Firewall [1], IPTables [2], Cisco Router Firewall [3], etc. As the population of Internet usage grows over time, firewalls inevitably need to process massive amounts of data. Therefore, firewall researchers have improved the speed of rule verification by revising the rule storage structure in a sequential manner to a tree structure, called the modern firewall. As a result, the speed of rule verification has

[1,2,3]The authors are with Department of Computer Science, Faculty of Informatics, Mahasarakham University, Mahasarakham, Thailand 44000, E-mail: suchart.k@msu.ac.th, panida.s@msu.ac.th and potchara.p@gmail.com

***Table 1:*** Example list of firewall rules.

| Rule | SIP | SP | DIP | DP | Pro | Action |
|---|---|---|---|---|---|---|
| $r_1$ | 192.168.1.0 - 255 | all | 100.50.0.0 - 255 | 80 - 90 | all | ***accept*** |
| $r_2$ | 192.168.1.20 - 50 | all | 100.50.0.100 - 200 | 80 | all | ***deny*** |
| $r_3$ | 192.168.1.50 - 200 | all | 100.50.0.100 - 200 | 50 - 80 | all | ***accept*** |
| $r_4$ | 192.168.1.20 - 150 | all | 100.50.0.100 - 255 | 85 - 100 | all | ***deny*** |
| $r_5$ | 172.16.1.0 - 255 | all | all | 21 - 22, 80, 443 | all | ***accept*** |
| $r_6$ | all | all | all | all | all | ***deny*** |

been increased from $O(n)$ to $O(log_2(n))$ as described in [4], [5], [6], [7] and [8]. However, nowadays, the volume of data communication is tremendous, and the vast majority of these communications are characterized by real-time or fast response communications. Therefore, firewalls need to be improved to speed up their rule-verifying in order to handle huge and uninterrupted data with $O(1)$ worst case access time. Such systems are called advanced firewalls. For example, IPSets [9] used hashing techniques, and IPack [11] applied indexing and a data packing approach to verify firewall rules. Although these advanced firewalls support high-speed rule verification, they still have several limitations: IPSets supports only small IP classes, consumes much memory as rules increase, and does not deal with rule conflicts, etc. IPack consumes less memory than IPSets, but there are known drawbacks to building structures for storing quite complex rules.

This research aims to simplify rule-storing structures, consume minimal memory, and eliminate rule conflicts, and with the speed of rule-validation maintained at $O(1)$. This paper is organized as follows: Section 2 explains in-depth how IPSets and IPack work. Section 3 articulates the contribution and proposed firewall design. Section 4 addresses the fast firewall framework. The performance evaluation is shown in Section 5. Lastly, Section 6 concludes this research.

## 2. HIGH-SPEED FIREWALL RULE VERIFICATION

This section describes the in-depth techniques of high-speed firewalls for rule verification:

### 2.1 IPSets

#### 2.1.1 Pros and Cons

IPSets [9] was developed to improve the speed of firewall rule verification with Big-O(1) access time based on IPTables [2] and Netfilter [10]. The highlight of IPSets is the ability to match any rule against any packet on both the incoming and outgoing interface at a constant speed using hashing techniques. However, even though IPSets is very efficient in rule verifying, there are some limitations that mean firewall administrators need to optimize rules before deploying them on the core firewall:

1. *Key generation*: Since IPSets implements hashing for the rule verification, it is always necessary to choose one of the key generation methods before using it, such as 'hash:ip', 'hash:net', 'hash:ip,port', 'hash:ip,port,ip', 'hash:net,port', etc. 'hash:net,port' is used to generate keys from a data set between 'net' and 'port', e.g. '192.168.1.0/24:80' has a range of keys (256 keys) equal to '1921681080', '1921681180', '1921681280', ..., '192168125480', '192168125580'. To make the key smaller, the dots (.) are removed from the IP address string, and these keys are always unique. The limitation with this feature is that firewall administrators have to choose which key generation method is appropriate for their organization, which requires a high level of experience with managing firewall rules.

2. *Memory consumption*: As explained above, key generation is an important step in the IPSets hashing technique, where keys are formed by cartesian product ($\times$) of different groups of data, such as 'hash:net,port', which are concatenated between IP addresses within the subnet (net) and destination ports (port). For example, cartesian product outputs of the subnet number 192.168.1.0/24 (Class C = 256 IPs) and the destination port ranging from 80 - 89 (10 ports): '1921681080', '1921681180', '1921681280', '...', '192168125589' (256 IPs $\times$ 10 ports=2,560 keys). It is clear that an extensive range of data, when combined into keys, increases the number of keys as well. Another example is the number of IP addresses within Class B; for example, 172.16.1.0/16 is 65,536 IP addresses. When they are concatenated to the destination port in the range 80 - 84 (5 ports), they become keys: '1721681080', '1721681180', '1721681280', '...', '17216825525584' (65,536 $\times$ 5 = 327,680 keys). Usually, each key is stored in unique memory, so if each hashed key uses 128 bits of memory, then the amount of memory used to store the keys in this example is equal to (327,680 $\times$ 16 bytes)/1,000 = 5,242.88 KB $\simeq$ 5.2 MB (High memory consumption). For the last example, class A, the number of IP addresses of this class is $2^{24}$. Therefore, when using IPSets is not recommended to use larger classes such as Class B or A (Class C is recommended) to create firewall rules because it requires a lot of memory to store the keys. The memory allocation of the IPSets is defined as $2^n$.

Initially it reserved 2 MB of memory, but later the reserved memory was insufficient for usability, so the IPSets had to reserve the memory 4, 8, 16, . . ., $2^n$ MB, respectively.

3. *Rule anomaly verification*: IPSets only focuses on high-speed rule matching but does not address the detection of anomalous firewall rules, which are divided into six categories: shadowing, correlation, generalization, redundancy, irrelevancy anomaly [12], and semantics loss [13]. IPSets does not consider the overlap and conflict of the generated rules because they do not effect the speed of rule matching. However, it can cause wasteful memory usage, and some rules that are hashed and stored in memory may not be processed.

### 2.1.2 IPSets Module

IPSets is a complementary module for IPTables [2] and Netfilter [10] to speed up rule matching from O($n$) to O(1), shown in Figure 1. Assuming that a packet flows through the incoming interface of the firewall (NIC$_1$), Netfilter receives the packet and forwards it to IPTables to be matched with the specified firewall rules. However, IPTables' rule matching speed is only O($n$); thus, IPTables forwards the packets to be processed against IPSets instead, where it can speed up to O(1). The result returned from IPSets processing can only be accept or deny. The result received is forwarded to IPTables and Netfilters for further processing. For example, if the result is an accept, this packet is forwarded to the destination network over the outgoing interface (NIC$_2$). Otherwise, this packet is immediately discarded (dropped).

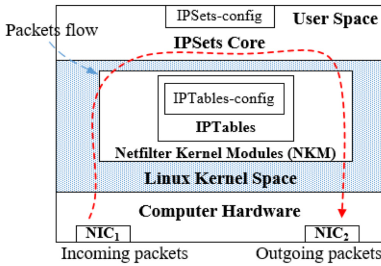


***Fig.1:*** *IPSets module over IPTables and Netfilter framework.*

### 2.1.3 IPSets Workflow

To understanding IPsets, this section provides an overview of its design and operating principles. As previously introduced, the perfect hashing technique is applied to the IPSets. Thus, in the first step of the IPSets process, the appropriate key generation scheme must be selected for their network. For this example, we chose to use the 'hash:net,port,net' format in conjunction with the rules in Table 1. IPSets can only use the required data fields to assemble keys, and the generated keys must also be able to match rules. Therefore, in this example, the IPSets uses the source IP (SIP), destination IP (DIP), and destination port fields to concatenate as the keys. The number of keys from Table 1 is 4,296,491,080 keys, except for the last rule because the number of keys exceeds the available memory. In the example of $r_1$ from Table 1, the number of SIP is 256 values (192.168.1.0-255), DP is 11 (80-90), and DIP is 256 (100.50.0.0-255). Therefore, the number of keys resulting from the Cartesian product between SIP, DP, and DIP of $r_1$ is 720,896, shown in Table 2.

***Table 2:*** The number of keys for each firewall rule.

| Rule | Number of keys generated by concatenating between SIP, DP and DIP ('hash:net,port,net') |
|---|---|
| 1 | 256 SIP * 11 DP * 256 DIP = 720,896 |
| 2 | 31 SIP * 1 DP * 101 DIP = 3,131 |
| 3 | 151 SIP * 31 DP * 101 DIP = 472,781 |
| 4 | 131 SIP * 16 DPs * 156 DIPs = 326,976 |
| 5 | 256 SIP * 4 DP * 4,294,967,296 DIP |
| 6 | 4,294,967,296 SIP * 65,536 DP * 4,294,967,296 DIP = 1,208,925,819,614,629,174,706,176 |

If one key requires 128 bits of memory space, the total amount of memory required is 68.47 GB (4,296,491,080 * 128). Firewalls currently in use do not have enough memory to support that. To address the problem of excessive memory usage, IPSets makes the data range of each field smaller. This is done by doing things such as setting the source IP ranges to class C and deactivating 'all' in the rules. Another technique is to reduce the data fields that are concatenated into keys, such as IP ('hash:ip') or Subnet field only ('hash:net'), or IP-port concatenation ('hash:ip,port'), etc. Next, the keys obtained from the first step are hashed and stored in baskets in order to group the duplicate keys, as shown in Figure 2.

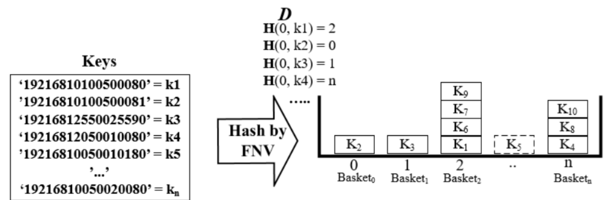


***Fig.2:*** *Hashing the keys to the basket.*

The FNV [14] algorithm is used for hashing keys, as it is fast and has low key collision rates. Let $d_i$ be a member of the function family ($D$), and $k_i$ be any key that will be hashed by $i \in 0, 1, \ldots, n$. After that, IPSets rehashes the redundant keys for the basket$_i$ again by changing the function family from 0 to 1, 2, . . ., $n$ until no more collisions occur, and then storing them in Table G and Table V [11], as shown in Figure 3.

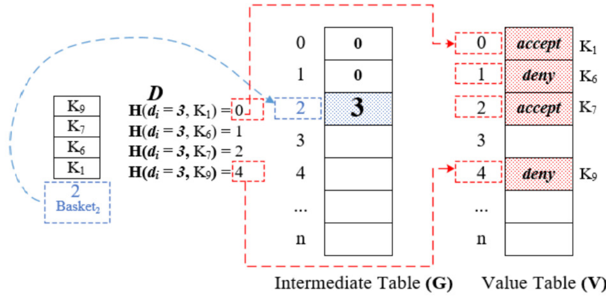


***Fig.3:*** *Rehash technique in case of keys collision.*

In basket$_2$ of Figure 3, there are four duplicate keys ($K_{1,6,7,9}$); the keys must be re-hashed by changing the $d_i$ value until no further key collision occurs in the basket. For example, given $d_i$ is equal to 3 and this $d_i$ does not cause the keys to be a collision, the G table stores 3 into the second memory address (Basket$_2$), and the V table holds the actual data hashed by 3 from the G table into memory with a unique address. In summary, the hashed or ready-to-use rules are stored in two tables: the G-table holds any hash-based function, and the V-table stores each rule's action.

## 2.2 IPack

### 2.2.1 Pros and Cons

IPack [11] is designed to solve the problems of IPSets mentioned above. First, IPack solves the memory consumption problem of IPSets by mapping rules into a 3D array memory structure. Subsequently, it reduces idle or redundant data from the 3D array structure into a one-dimension array by using a packing technique. As a result, the amount of memory consumed is reduced from $O(2^n)$ to $O(n)$. IPack can support all kinds of rules, whether large or complex, such as rules built on a variety of conditions, or rules created from Class A or B, etc. IPSets requires high administrative skills to design rules appropriate for each network before deploying them, but IPack only requires the basic skills of a typical administrator. IPack rule-verifying speed is the same as IPSets, that is O(1). IPSets cannot detect anomalies and rule conflicts, but these features are included in IPack.

IPack is not perfect. The disadvantages of IPack are:

1. *Verification of rule anomalies*: Since IPack has to eliminate the anomalous rules before mapping them into a 3D array structure, the build time is slightly greater than IPSets.
2. *Establishment of the rules*: The establishment of IPack rules is highly complex and has several steps. Rules are challenging to develop, such as eliminating anomalies, mapping rules into a three-dimensional array, reducing the idle memory, and compressing data in a three-dimensional array to a one-dimensional array.
3. *Rule revisions*: As the rules change, IPack always needs to restart the process by eliminating the anomaly rules and then doing compression of the rules, which increases the overall rule processing time.

### 2.2.2 IPack Framework

IPack was completely redeveloped to fix the disadvantages of IPSets based on Netfilter Kernel Modules (NKM) over Linux Kernel version 2.6. The matching process between packets and rules is similar to IPSets. However, IPack has slightly fewer steps than IPsets, resulting in the rule execution speed being slightly higher than IPSets, as shown in Figure 4.

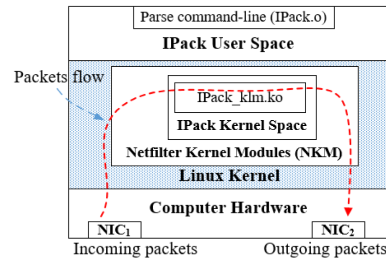


***Fig.4:*** *IPack over NKM and Linux Kernel.*

As shown in Figure 4, when packets have flowed from the Linux Kernel to NKM acting like a computer device driver, IPack executes them at the kernel level without forwarding packets to user space like IPSets. This reduces processing time.

### 2.2.3 IPack Workflow

There are three steps of IPack's workflow: First, IPack converts the rules into a tree structure called PSD [11] to eliminate rule conflicts. The rules appearing in the tree structure are guaranteed to having no rule conflicts. Second, IPack maps all the rules in the tree structure into three-dimensional arrays that store the data as a sparse matrix [15]. Finally, IPack performs compression of the unused rules in the three-dimensional arrays to generate one-dimensional arrays using the IP packing technique. As a result, the data in the one-dimensional arrays contains the rules without any conflicts, and the memory consumption is optimal, as shown in Figure 5. In conclusion, IPack improves several IPSets limitations. Namely, the memory consumption is decreased from $O(2^n)$ to $O(n)$, it has support for all types of rules, and admins can configure rules immediately and eliminate rule conflicts.

## 3. RESEARCH CONTRIBUTION AND FIREWALL DESIGN

Although IPack can improve upon most of the IPSets flaws, it still has two weaknesses: 1) it consumes too much time in the rule conflict elimination process, and 2) it consumes too much time to

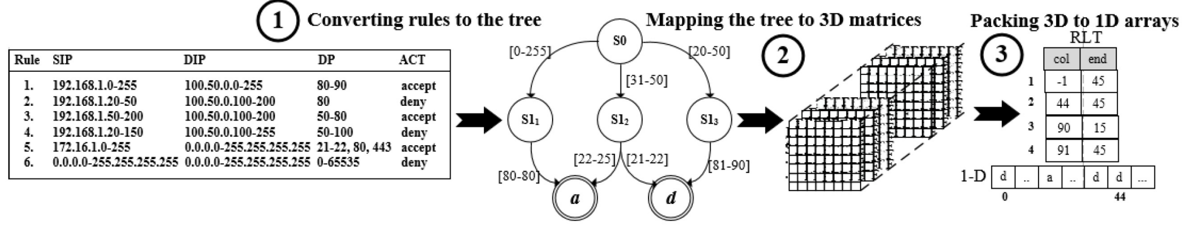


***Fig.5:*** *IPack's workflow.*

compress the data from three-dimensional to one-dimensional arrays. Therefore, this research has designed a novel firewall framework to reduce the time required, which increases the overall speed of rule matching.

## 3.1 Improving the Speed of Eliminating Rule Conflicts

As previously mentioned, IPack uses a tree structure to eliminate rule conflicts, and it takes $O(n^2)$ to establish the tree structures. This research designs a novel conflict resolution method, namely FIRST MATCH - FIRST EXECUTE (FMFE), which consumes much less time for eliminating rule conflicts. The principle of this method is that any packets can be matched (MATCH) against one of the rules arranged in order from top ($r_1$) to bottom ($r_n$), indicating that the rule is selected and executed (EXECUTE). For example, from Figure 6 converted from the rules in Table 1, it can be seen that $r_1$ completely obscures $r_2$, and $r_1$ still partially obscures $r_3, r_4$, and $r_6$. In fact, $r_2$ is never executed because no packets can match it ($r_1$ completely shadows $r_2$), which means $r_1$ is FMFE. Following the example, $r_4$ is partially obscured from $r_1$. Thus, $r_4$ can verify the packets that are not shadowed only, that is $r_1$ is partially FMFE of $r_4$. In the last rule ($r_6$) in Figure 6, its boundary is greater than any other rules. However, it is rarely matched against any packets because it is blocked by most of the other rules above ($r_1, ..., r_5$ are partially FMFE of $r_6$). Besides, conflicts in the original rules in Figure 6 remain. For example, $r_1$ correlates with $r_4$, but they have different decisions (Correlation anomaly [16]). In conclusion, the rules created later may be obscured by the rules created earlier.

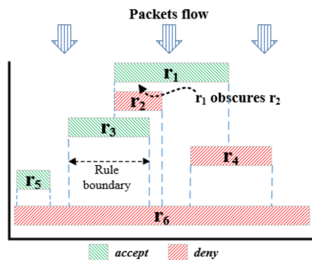


***Fig.6:*** *Controversial original rules.*

As mentioned earlier, with the concept of FMFE, rules obscured from previous rules (A rule on the top) are never executed. Therefore, they can be eliminated using the FMFE concept as shown in Figure 7. For example, $r_2$ is entirely obscured by $r_1$. Thus, $r_2$ can be eliminated from rules. In addition, $r_4$ is partially obscured by $r_1$, so the shadowed part of $r_4$ can be eliminated as well. Once all the rules have been processed following the FMFE technique, they do not have any conflicts, as shown in Figure 7. The FMFE algorithm is explained next.

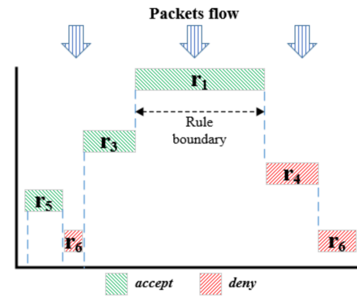


***Fig.7:*** *Conflict-free rules optimized by FMFE.*

1. *Preparing rules*: The first step is to prepare the rules before processing them with the SET operation by concatenating the key fields in the rules into strings, then converting them to integer numbers, and finally storing them in set variables. For example, for $r_1$ in Table 1, this algorithm selects the fields SIP=192.168.1.0-255, DP=80-90 and DIP=100.50.0.0-255 which are used to generate the keys in IPSets. Then it concatenates the selected fields to create strings such as '19216810801005000', '19216810801005001', '19216810801005002', '...', '192168125590100500254', '192168125590100500255'. The number of strings is equal to 720,896 (SIP = 256 * DP = 11 * DIP = 256). After that, it converts the strings into a set of integers: $r_1$ = [19216810801005000, 19216810801005001, 19216810801005002, ..., 192168125590100500254, 192168125590100500255]. This process continues until all rules have been converted. In practice, only two rules should be executed at a time to save processing memory.

2. *Removing the obscured part*: At this stage, the algorithm removes the obscured portion of the rules using the DIFFERENCE operator (**DIFF**). For example, $r_2$ differs $r_1$ is equal to the empty set ($\emptyset$) because $r_2$ is a subset of $r_1$ ($r_2 \subset r_1$) as shown in Figure 8. In another example, $r_3$ and $r_4$ are partially obscured by $r_1$, so $r_3$ DIFF $r_1$: [19216815050100500100, 19216815050100500101, ..., 192168120079100800200]) **DIFF** [19216810801005000, 19216810801005001, ..., 192168125590100500255] = [19216815050100500100, 19216815050100500101, ..., 192168120079100500200]. $r_4$ **DIFF** $r_1$ is equal to [192168120085100500100, 192168120085100500101, ..., 1921681150100100500255] **DIFF** [19216810801005000, 19216810801005001, ..., 192168125590100500255] = 19216812091100500100, 19216812091100500101, ..., 192.1681150100100500255. The above algorithm continues until all obscured rules have been removed, as shown in Figure 7. This step is considered completed. Rules which have passed this stage are deemed to be free from conflicts.

3. *Reconfigure rules*: Once all the obscured parts of the rules have been removed, the next step is to reconfigure the rules based on Figure 7, as shown in Table 3. In Table 3, $r_2$ from Table 1 is deleted (disappearing in Table 3) because it is inactive (never matched). Moreover, $r_2$ (move up from $r_3$ in Table 1 to $r_2$ in Table 3), and $r_3$ (move up from $r_4$) is smaller since it is partially cut off by $r_1$. The $r_4$ does not change because it is not a subset of any previous rules ($r_4 \not\subset r_1, r_2$ and $r_3$ or $r_1 \dot{\cup} r_1, r_2$ and $r_3$). The last rule ($r_6$) from Table 1 is removed because it drops all packets by default and consumes a lot of memory. Note: If the last rule is created as keys, the IPSets cannot execute it due to insufficient memory. Thus, in order to compare its performance with other firewalls, it is necessary to delete $r_6$ from the rules as shown in Table 3.

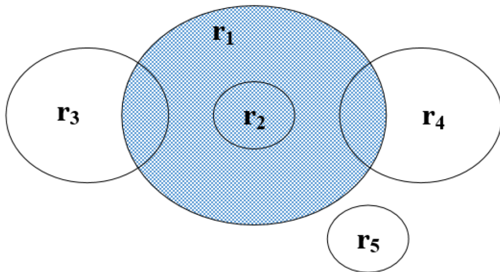


***Fig.8:*** *$r_2, r_3$ and $r_4$ DIFF $r_1$.*

## 3.2 Direct Memory Mapping

Conflict-eliminated rules are mapped directly into memory without converting to 3D and one-dimensional arrays like IPack, and without hashing the rule's keys to memory as IPSets. This new technique reduces both the time and complexity of setting rules, making its cost less than both existing techniques (IPSets & IPack). This is achieved with the following steps. The first step is to divide the memory into 256 fixed-size (blocks), each equal to $n$-bit size, where $n$ is the number of rules, as shown in Figure 9. For example, if the number of rules is four (in Table 3), the data stored in the memory of each position has four bits (0000). Each block of memory stores the sub-fields of each rule. For example, the $SIP_1$ block stores the first octet of SIP, and the $SIP_2$ block collects the second octet of SIP, respectively. According to $r_1$ in Table 3, SIP is 192.168.1.0-255. Therefore, the memory of $SIP_1$ block in position 192 is equal to 0001 (each bit represents a rule number). The $168^{th}$ position memory holds the $SIP_2$ block and contains 0001. The $1^{st}$ address memory of $SIP_3$ is 0001, and the memory addresses at 0-255 of the $SIP_4$ block are 0001. The DIP of $r_1$ (100.50.0.0-255) is mapped into memory in the same manner as SIP. The number of ports is 16 bits, divided into two equal blocks ($DP_1$ and $DP_2$), each equal to 8 bits ($2^8$ = 256 addresses). If the destination port number is 1234 ($00000100\ 11010010_2$), the position of $DP_1$ is 4 ($00000100_2$), and the position of $DP_2$ is 210 ($11010010_2$). In the case of $r_1$ in Table 3, the destination port numbers are 80-90; thus, they are mapped into the memory address 0 of $DP_1$ block and address position 80 to 90 of $DP_2$. The data 0001 is recorded in these addresses. The last two blocks are used to store rule actions, divided into two $n$-bit blocks. In the $r_1$ in Figure 9, the action is to accept, so the data 0001 is stored in the accept block (ACCEPT), and 0000 is recorded to the denied block (DENY). For other rules, we complete the same steps, all of which are entirely stored in direct memory, as shown in Figure 10.

## 3.3 Improving the Speed of Matching Rules

Let $p_i$ be any packet that flows into the firewall, where it can be matched against any rule, resulting in choosing an action of accepting or denying, called a successful matching. If such $p_i$ cannot be matched with any rule, it must always be dropped (called implicit denial). Assume $p_1$ has a source IP address (SIP) of 192.168.1.20, and it needs to access the destination website (DP=HTTP=80) with an IP number (DIP) of 100.50.0.100. The $p_1$ can be matched with a rule that has been mapped to the direct memory with the following steps: Given the variables X and Y to store values for calculations, initializing X = $1..1_n$ and Y = $0..0_n$, where $n$ is the position of the data bits, and the number of bits is equal to the number of rules (In this example, X=1111 and Y=0000). First, the firewall takes the first octet (192) of the source IP packet, pointing to the address of the $SIP_1$ block in memory, as shown in Figure 10. The result is equal to

***Table 3:*** Example of reconfigured rules.

| Rule | SIP | DIP | DP | Action |
|---|---|---|---|---|
| 1 | 192.168.1.0-255 | 100.50.0.0-255 | 80-90 | accept |
| 2 | 192.168.1.50-200 | 100.50.0.100-200 | 50-79 | accept |
| 3 | 192.168.1.20-150 | 100.50.0.100-255 | 91-100 | deny |
| 4 | 172.16.1.0-255 | all | 21-22,80,443 | accept |

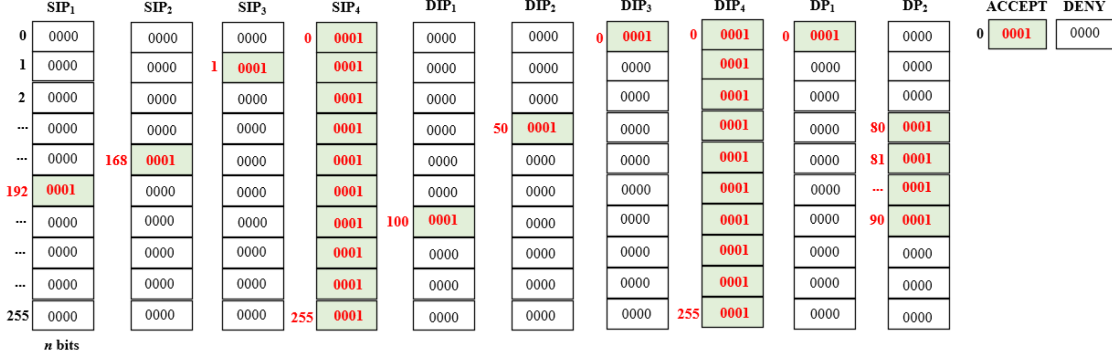


***Fig.9:*** *Mapping $r_1$ to the direct memory.*

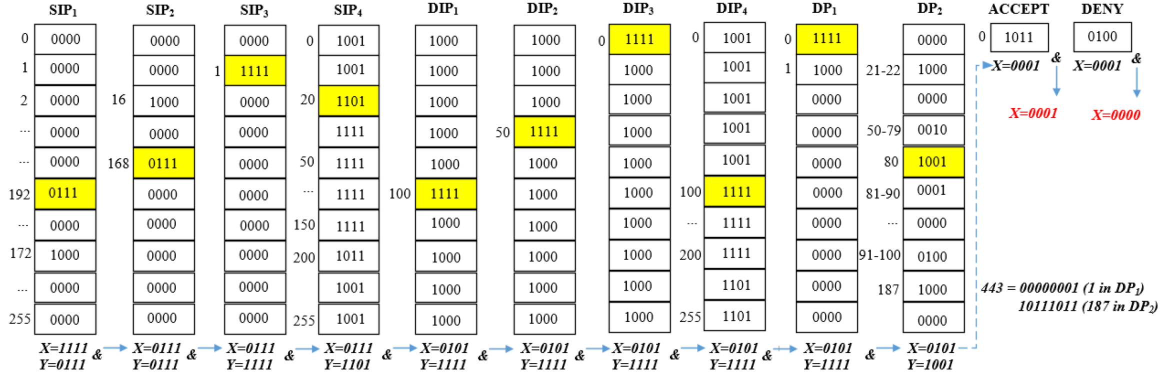


***Fig.10:*** *Mapping all rules to the direct memory, and how to match any rule against any packet.*

0111 and is stored in the variable Y (Y=0111). The X and Y values are fed to the AND operator (X=1111 AND Y=0111). The calculated result is 0111 and is stored in the variable X (X ← X AND Y). Then the firewall reads the second octet of the packet's source IP (168) and fetches the data from location 168 of the $SIP_2$ block in memory. The result is 0111 and stored in the variable Y. The algorithm will process as in the previous step by ANDing the values in X (0111) and Y (0111). The result obtained from the calculation is stored in X (0111) again. The other field calculations in the rule are the same as the previous steps. The final output stored in X is calculated with the value contained in the ACCEPT and DENY variables as shown in Figure 10 (Right-hand side). The results indicate what the firewall should do with the packet $p_1$, either passing or dropping it. In this example, X is equal to 0001, and ACCEPT and DENY variables are equal to 1011 and 0100, respectively. Therefore, the resulting calculation of X and ACCEPT (X ← X AND ACCEPT) is 0001, and X and DENY (X ← X AND DENY) is 0000. The results show that $p_1$ can continue to pass through to another network (X AND ACCEPT $\neq$ 0). The method of matching a packet against the rules is shown in Algorithm 3.3

**Algorithm 1** : Matching firewall rules

1: **Input**: $n$, $sip_{1,2,3,4}$, $dip_{1,2,3,4}$, $dp_{1,2}$, $\mathrm{SIP}_{1,2,3,4}$, $\mathrm{DIP}_{1,2,3,4}$, $\mathrm{DP}_{1,2}$, ACCEPT, DENY
2: **Output**: $\{0,1\}$
3: X ← $11...1_n$, Y ← $00...0_n$
4: Y ← SIP$[sip_1]_1$, X ← X AND Y
5: Y ← SIP$[sip_2]_2$, X ← X AND Y
6: Y ← SIP$[sip_3]_3$, X ← X AND Y
7: Y ← SIP$[sip_4]_4$, X ← X AND Y
8: Y ← DIP$[dip_1]_1$, X ← X AND Y
9: Y ← DIP$[dip_2]_2$, X ← X AND Y
10: Y ← DIP$[dip_3]_3$, X ← X AND Y
11: Y ← DIP$[dip_4]_4$, X ← X AND Y
12: Y ← DP$[dp_1]_1$, X ← X AND Y
13: Y ← DP$[dp_2]_2$, X ← X AND Y
14: **if** X AND ACCEPT $\neq 0$ **then**
15: Print "ACCEPT MATCH", Return 1
16: **else if** X AND DENY $\neq 0$ **then**
17: Print "DENY MATCH", Return 0
18: **else**
19: Print "MISMATCH"
20: **end if**

To find out which rule number matches a packet ($p_1$), consider where the data bit appears as 1 in X. For this example, the rule that matches against $p_1$ is

rule number 1 (X=0001). If a packet matches rule number 2, the result is 0010. If it matches rule number 3 (X=0100), it is 0100 (DENY=0100, X AND DENY = 0100).

The algorithm used for searching for the rule number is shown in Algorithm 3.3 This direct bit-level memory matching is similar to a bloom filter [17], but it has an advantage over the bloom filter. That is, it can point to the number of rules.

**Algorithm 2** : Searching the rule number

```
1:  Input: x
2:  Output: n
3:  n ← 0, len ← len(x)
4:  while x[n] ≠ 1 do
5:      n = n + 1
6:      if n == len then
7:          Print "Can't find the rule"
8:          Return NULL
9:      end if
10:     Print "Discovered the rule number: " n
11:     Return n
12: end while
```

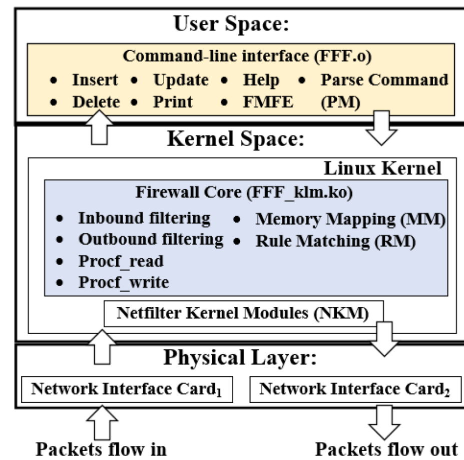


***Fig.11:*** *FFF ($F^3$): Fast Firewall Framework.*

## 4. FAST FIREWALL FRAMEWORK

As the proposed firewall modifies several functional structures such as rule design, rule creation, conflict elimination, and rule verification (rule matching), this research to create FFF ($F^3$) has to redesign the firewall from the kernel to the user interface, as shown in Figure 11. The details of FFF development are presented next:

### 4.1 Implementing the Firewall Core

A firewall core is a critical software system that drives and controls all the firewall functionality. Its tasks include transmitting and dropping data with the kernel-level operating system, verifying rule anomalies and conflicts, matching rules against packets, and packet analysis, etc. For this paper, the firewall core was developed using the C/C++ language (GCC 4.4.7) and GNU Make 3.8 on 64-bit Linux kernel version 2.6, namely FFF_klm.ko, as shown in Figure 11 (in Kernel Space). The FFF_klm.ko kernel module is responsible for filtering incoming packets that flow from the incoming network card (Network Interface $Card_1$), Netfilter Kernel Modules (NKM), and the built-in software package called Inbound filtering. To process packets with the NKM, FFF_klm.ko is always executed via the built-in Procf_read and Procf_write function. Once the packets have been filtered, they are verified against the firewall rules configured from the command-line (in the user space) by Rule Matching (RM). If such packets can be matched to any of the rules, they must be either forwarded to the destination network via the outgoing network card ($Card_2$) or blocked. On the other hand, if they match no rules, they must be implicitly thrown away.

### 4.2 Implementing the User Interface

Any decision of the firewall core is executed via the rules that the user configures in the User Space with the command-line user interface (FFF.o). The syntax of the rules of this research is similar to IPTables [2]. Therefore, administrators can configure them immediately without any learning difficulties. The grammatical form of the rules is shown in Table 4. According to the first example rule, "fff –out –srcip 192.168.1.0 –mask 255.255.255.0 –destip any –destport 80 –action accept" means to allow (*accept*) the network 192.168.1.0 (256 IP addresses) to access any web servers (HTTP = port 80), and it applies to the outbound interface (–out). When the grammar of the rules is applied to the rules in Table 3, the results are shown in Table 5. The rules can be reconfigured at any time with Insert, Delete, and Update commands. Before the created rules are forwarded to the firewall core to be executed on-demand, they must always be parsed for grammatical integrity first with Parse Command (PM) in userspace, as shown in Figure 11. When the interpretation of the rules does not make any mistakes, they are always checked for conflicts by the FMFE method. After all conflicting rules are resolved, they are passed to the firewall core by the Procf_read and Procf_write to map the rules into the direct memory using the Memory Mapping approach (MM), which is the last step of FFF. Then it is ready to run.

***Table 4:*** *The FFF command-line syntax.*

| Command | Description | Example of how to use |
|---|---|---|
| fff | The firewall name | fff |
| in, out | Inbound, outbound interface | –in<br>–out |
| srcip | Source IP address | –srcip 192.168.1.0<br>–srcip any |
| mask | Subnet mark of IP address | –mask 255.255.255.0 |
| destip | Detination IP address | –destip 200.0.0.0<br>–destip any |
| srcport | Source Port | –srcport 1234<br>–srcport 100-200<br>–srcport 1234, 1350 |
| destport | Destination Port | –destport any<br>–destport 443, 8080 |
| proto | Protocol | –proto tcp<br>–proto udp<br>–proto all |
| action | Decision of rule | –action accept<br>–action deny |
| delete | Delete any rule | –delete 5 |
| print | Print all rules | –print |
| help | Help command | –help |
| exit, quit | Exit program | –exit, –quit |

## 5. PERFORMANCE EVALUATION

There are three metrics used for evaluating firewall effectiveness in this paper: throughput, time, and space complexity, which are detailed below.

### 5.1 Throughput

The evaluated firewalls were deployed on a production network, as shown in Figure 12. The client-side network connectivity starts from a client to the Internet via ISP (3BB) using VDSL technology, which has a 100/50 Mbps (download/upload stream). IPSets, IPack, and FFF are installed in the font of the VDSL router to evaluate their effectiveness using the IPERF client software [18]. The sever-side network starts from the Internet to the server at Mahasarakham University via UNINET (ISP). This server has software to estimate the packet throughput, known as the IPERF server. The criteria for evaluating firewalls consist of several factors: TCP and UDP packet transfer rates, and the number of rules. The TCP packet evaluation criteria consist of the window size as 16, 32, and 64 KB, respectively. The time interval is equal to 1 sec, and the concurrent connection limit is one channel. The data types used for the transfer test are packets and binaries. The UDP bandwidth used to evaluate the firewalls is 100 Mbps. Rule sets with 100, 500, 1,000, 2,000, 3,000, 4,000, 5,000, 10,000 rules were used.

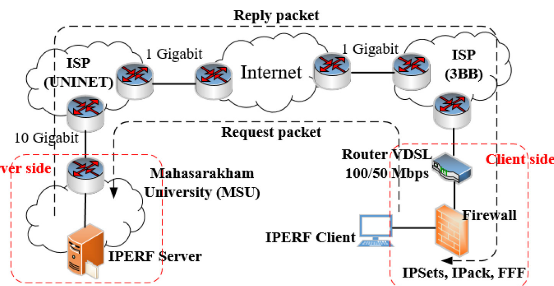


***Fig.12:*** *Network topology for evaluating IPSets/IPack/FFF.*

For firewalls' TCP throughput running on the real network connection, Figure 13, 14, and 15 show the throughput of IPSets, IPack, and FFF packet transfer using different TCP window sizes (WS), i.e., 16, 32, and 64 KB, respectively. The number of firewall rules changes ranging from 100 to 10,000. The average throughput of each firewall to transfer packets set the window size as 16 KB is 912.75 (IPSets), 923.87 (IPack), and 924.71 KB/sec (FFF), respectively. The results show that all firewalls have similar throughput, where packet transfer rates differ no more than 1.2 KB/sec. By increasing the size of windows from 16 to 32 KB, the overall throughput of firewalls approximately increases 1.75 times on average (from 0.9 to 1.6 MB/sec). Likewise, when the window size is set to 64 KB, all firewalls can transfer packets better, increasing from 1.6 to 2.1 MB/sec. In the case of evaluating the throughput of TCP by communicating over the binary data, shown in Figure 16, 17, and 18, the size of the windows is increased from 16, 32, and 64 KB, respectively, similar to the TCP packets. The results from our overall evaluation show that FFF has a slightly higher throughput than the other two firewalls, and the speed of rule matching tends to be relatively stable. The maximum speed of binary data transfer is approximately 18 Mbps on average.

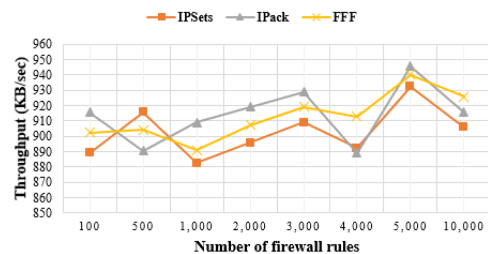


***Fig.13:*** *TCP throughput (packet) with WS=16K.*

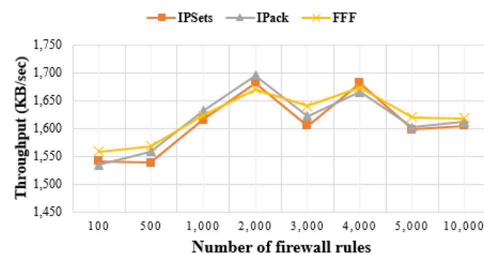


***Fig.14:*** *TCP throughput (packet) with WS=32K.*

***Table 5:*** *Examples of FFF firewall rule syntax.*

| Rule | Rules built according to syntax | Meaning |
|---|---|---|
| **1** | fff –in —-srcip 192.168.1.0 –mask 255.255.255.0 –destip 100.50.0.0/24 –destport 80-90 –action accept | Allows 256 source IPs to connect to 256 destination IPs which are open with port number 80-90. |
| **2** | fff –in —-srcip 192.168.1.50-200 –destip 100.50.0.100-200 –destport 50-79 –action accept | Allows 151 source IPs to connect to 101 destination IPs which are open with port number 50-79. |
| **3** | fff –out —-srcip 192.168.1.20-150 –destip 100.50.0.100-255 –destport 91-100 –action deny | 131 source IPs are not allowed to connect to 156 destination IPs that are open with ports numbered 91-100. |
| **4** | fff –in —-srcip 172.16.1.0 –mask 255.255.255.0 –destip all –destport 21-22, 80, 443 –action accept | Allows 256 source IPs to connect to all destination IPs which are open with port numbers 21-22, 80, 443. |

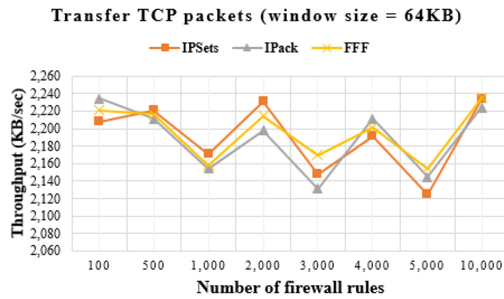


***Fig.15:*** *TCP throughput (packet) with WS=64K.*

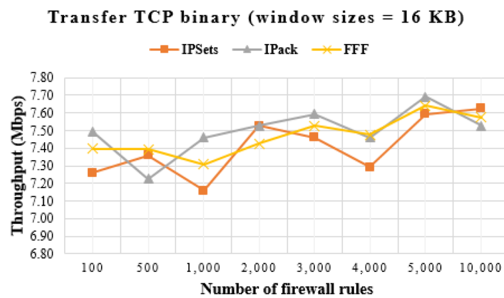


***Fig.16:*** *TCP throughput (binary) with WS=16K.*

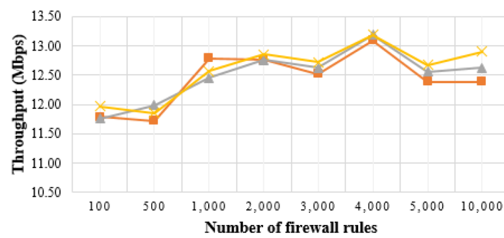


***Fig.17:*** *TCP throughput (binary) with WS=32K.*

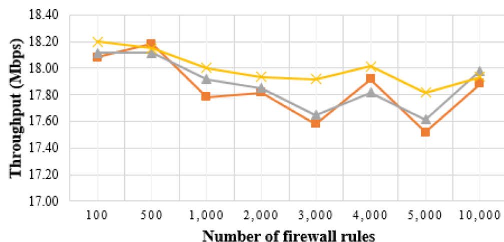


***Fig.18:*** *TCP throughput (binary) with WS=64K.*

Since UDP has less overhead associated with connections, error checks, and the retransmission of missing data. Therefore, it can transfer large volumes of data quickly. Besides, it is very suitable for testing real-time or high-performance applications like high-speed firewalls. According to the UDP throughput shown in Figure 19, the firewalls are rated for throughput by packet transfer with a bandwidth size of 100 Mbps. The results clearly show that the firewalls can transfer UDP packets about 15 times more quickly than TCP packets. FFF has the best performance, with an average maximum throughput of 37.4 Mbps. Likewise, when testing all firewalls against the binary data transfer method like in Figure 20, the throughput is very high (Approx. 95.4 Mbps), almost the same as the maximum configured bandwidth. As mentioned above, it shows that communication in the binary format over the UDP is the fastest, and the fastest firewall capable of verifying the binary transfer in this situation is FFF.

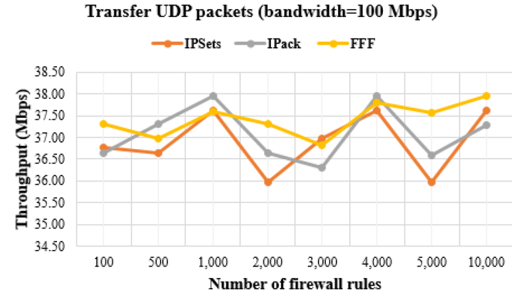


***Fig.19:*** *UDP throughput (packet).*

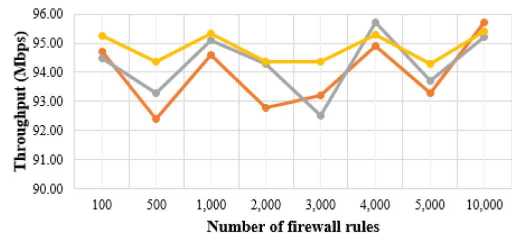


***Fig.20:*** *UDP throughput (binary).*

## 5.2 Time Complexity

There are two types of time estimates for high-speed firewalls: rule establishment time and rule verification time. IPSets takes time to generate keys from rules and hashes them into memory without col-

lision. Therefore, the rule establishment time is the key generation time plus hashing time, and that is O($n^2$) [11]. Likewise, with IPack, it takes time to eliminate conflicts, time for mapping the rules tree to three-dimensional arrays, and time to pack three-dimensional arrays to one dimension arrays. Therefore, the total time of IPack to establish the rules is O($n^2$) [11], the same as for IPSets. FFF needs less time than both firewalls. FFF requires time to eliminate the rule conflicts and some time to map rules into the direct memory, so the total time it takes to implement the rules is O($n$). Additionally, it has a less complex data structure for storing rules than IPack, similar to IPsets. The time to verify the rules (matching rules) for all three firewalls is O(1) because IPSets uses the hashing technique, and IPack and FFF use indexing methods to point directly to the memory where the rules are stored.

### 5.3 Space Complexity

The space complexity is proportional to the main memory used to store rules while the firewall is running. IPSets consumes memory for rules (hashed keys) in two tables: Table G and V. Because of the large number of keys generated by the rules, the IPSets maker requires increasing the memory space in cases of insufficient storage until it is equal to O($2^n$). For example, in Table 3, from $r_1$ to $r_3$, when they are made as keys of type 'hash:net,port,net', the total number of keys is 1,336,335. If each key uses 32-bits of data, then the amount of memory consumed is 10.69 MB.

$$IPSets = \frac{1,336,335(keys) * 32(bits) * 2(tables)}{8 * 10^6} = 10.69\ MB$$

Assume that IPSets is assigned the default memory value of 8 MB. Then, the next memory allocation which result in insufficient memory is $n = 4$ ($2^4 = 16$ MB). The memory consumption in the case of IPack is constant because the arrays have a fixed size, and the formula is 65,536 + (2,322 * $n$), where $n$ is the number of rules. In Table 3, the total memory used to store IPack rules is 0.299 MB. Thus, IPack memory space consumption is O($n$).

$$IPack = \frac{65,536 * 32(bits) + 2,322 * 4(n) * 32(bits)}{8 * 10^6} = 0.299\ MB$$

Finally, FFF uses ten blocks of memory to store rules, as shown in Figure 10. Each block is equal to 256 positions, and each position is $n$ bits, where $n$ is the number of rules. As an example from Table 3, there are four rules. Therefore, FFF consumes little memory to store the available rules:

$$FFF = \frac{10 * 256 * 4(n)}{8 * 10^6} = 0.00512\ MB$$

The memory usage increases depending on the rate of change of the $n$ value. One rule is equal to one bit. The space complexity of FFF is O($n$) where $n$ is the number of bits. Assuming the tested rules range from 1 to 10000, the memory used for each location is 32 bits, and the number of keys generated from each IPSets rule is a constant of 4096 keys (calculated from the average). As a result, the memory required to store all rules is shown in Figure 21.

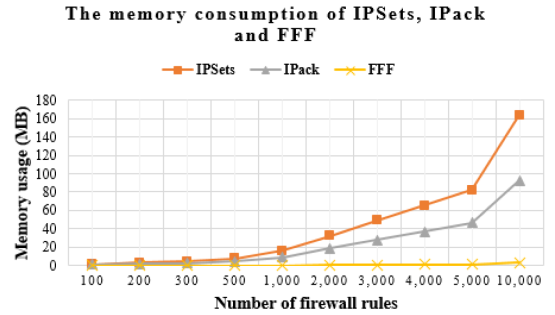


***Fig.21:*** *The memory consumption trend of each firewall.*

## 6. CONCLUSION

This research has successfully designed and developed a high-speed firewall framework named FFF to address previously published high-speed firewall drawbacks in IPSets and IPack. Each firewall evaluated in this research takes similar time to verify packets against the rules and matching is constant, that is O(1). In addition, the time to establish the rules depends on the structural complexity of each firewall, with the most complex being the IPack and IPSets, respectively, which both take O($n^2$). In contrast, FFF takes less time to establish rules than both of the other firewalls, and is O($n$). The memory consumption of each firewall is different depending on the number of rules and the memory structure used to store rules. IPSets consumes the highest memory because the number of keys generated by rules is high. On the other hand, FFF allocates the least memory because its memory structure is bit-type data. Finally, the summary of time, space complexity, and other features for the firewalls is shown in Table 6.

***Table 6:*** *Summary of IPSets, IPack and FFF performance evaluation.*

| Firewall feature list | Evaluation results | | |
|---|---|---|---|
| | **IPSets** | **IPack** | **FFF** |
| Time for building structures | $O(n^2)$ | $O(n^2)$ | $O(n)$ |
| Time for rule verifying | O(1) | O(1) | O(1) |
| Time to look up the rule no. | N/A | O(1) | $O(n)$ |
| Memory consumption | $O(2^n)$ | $O(n)$ | $O(n_{bit})$ |
| IP subnet support (A - C) | C only | All | All |
| Basic operations | Match only | All | All |
| Checking for rule conflicts | No | Yes | Yes |
| Eliminating rule conflicts | No | Yes | Yes |
| Skills for handling rules | High | Low | Low |
| Technique for matching rules | Hashing | Indexing | Indexing |
| Data structure complexity | Low | High | Low |

**Remarks**: the basic operations: matching, forwarding and dropping the packets; N/A: IPTables operates this function instead.

**Note:** Examples of FFF source files can be downloaded from https://github.com/Suchart-k/FFF.

## ACKNOWLEDGEMENTS

This Research was Financially Supported By Faculty of Informatics, Mahasarakham University Grant year 2022.

## References

[1] Microsoft 2021: Windows Defender Firewall with Advanced Security. https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-firewall/windows-firewall-with-advanced-security (2021). Accessed 23 Apr 2021.

[2] Diekmann, C., Hupel, L., Michaelis, J., Haslbeck, M., Carle, G.: Verified iptables Firewall Analysis and Verification. Journal of Automated Reasoning 61(1), 191-242 (2018). doi:10.1007/s10817-017-9445-1.

[3] Richard Deal, Cisco Router Firewall Security, Cisco Press, 2004.

[4] H. Hamed, A. El-Atawy and E. Al-Shaer, "On Dynamic Optimization of Packet Matching in High-Speed Firewalls," in *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1817-1830, Oct. 2006, doi: 10.1109/JSAC.2006.877140.

[5] S. Khummanee, A. Khumseela and S. Puangpronpitag, "Towards a new design of firewall: Anomaly elimination and fast verifying of firewall rules," *The 2013 10th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2013, pp. 93-98, doi: 10.1109/JCSSE.2013.6567326.

[6] T. Chomsiri, X. He, P. Nanda and Z. Tan, "Hybrid Tree-Rule Firewall for High Speed Data Transmission," in *IEEE Transactions on Cloud Computing*, vol. 8, no. 4, pp. 1237-1249, 1 Oct.-Dec. 2020, doi: 10.1109/TCC.2016.2554548.

[7] D. Rovniagin and A. Wool, "The geometric efficient matching algorithm for firewalls," *2004 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, 2004, pp. 153-156, doi: 10.1109/EEEI.2004.1361112.

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

[9] IPSets. (2021, 10 Feb). IP set features. [Online]. Available: https://ipset.netfilter.org/features.html.

[10] J. Kadlecsik and G. Psztor, Netfilter performance testing, Netfiler Research Report 2004, December, 2020.

[11] Suchart Khummanee, "IP Packing Technique for High-speed Firewall Rule Verification," *Journal of Internet Technology*, vol. 20, no. 6 , pp. 1737-1751, Nov. 2019, doi: 10.3966/160792642019102006006.

[12] E. S. Al-Shaer and H. H. Hamed, "Modeling and Management of Firewall Policies," in *IEEE Transactions on Network and Service Management*, vol. 1, no. 1, pp. 2-10, April 2004, doi: 10.1109/TNSM.2004.4623689.

[13] Khummanee S. (2019) The Semantics Loss Tracker of Firewall Rules. In: Unger H., Sodsee S., Meesad P. (eds) Recent Advances in Information and Communication Technology 2018. IC2IT 2018. Advances in Intelligent Systems and Computing, vol 769. Springer, Cham. https://doi.org/10.1007/978-3-319-93692-5_22.

[14] L. C. Noll, The core of the FNV hash, FNV Research Report 2013, April, 2021.

[15] R. Shahnaz, A. Usman and I. R. Chughtai, "Review of Storage Techniques for Sparse Matrices," *2005 Pakistan Section Multitopic Conference*, 2005, pp. 1-7, doi: 10.1109/INMIC.2005.334453.

[16] S. Khummanee, P. Chomphuwiset, P. Pruksasri "Decision Making System for Improving Firewall Rule Anomaly Based on Evidence and Behavior," *Advances in Science, Technology and Engineering Systems Journal*, vol. 5, no. 4, pp. 505-515 (2020), doi: 10.25046/aj050460.

[17] H. Byun and H. Lim, "Functional bloom filter, better than hash tables," *2018 International Conference on Electronics, Information, and Communication (ICEIC)*, 2018, pp. 1-3, doi: 10.23919/ELINFOCOM.2018.8330628.

[18] B. A. Jon Dugan and E. Seth, IPERF - the ultimate speed test tool for TCP, UDP and SCTP, IPERF testing report 2017, June, 2020.

**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, agricultural robotics, and Internet of things (IoT).

**Panida Songram** received a Ph.D. degree in Computer Science from the King Mongkut's Institute of Technology Ladkrabang. She is currently an Assistant Professor at the Faculty of Informatics, Mahasarakham University (MSU), Thailand. Her research focuses on data mining, algorithms, classification and applications.

**Potchara Pruksasri** has obtained both B.Sc. and M.Sc. of Computer Science at Khon Kaen University Thailand in 2000 and 2005 respectively. In 2005, he has been employed as a lecturer at Mahasarakham University, Thailand. He is currently working on his Ph.D. research at Computer Science Department, Faculty of Informatics, Mahasarakham University. His research focuses on information security and access control of the supply chain information system to secure data exchange of global supply chains.