A Stateful Mechanism for the Tree-Rule Firewall

Thawatchai Chomsiri, Xiangjian He, Priyadarsi Nanda, Zhiyuan Tan

Center for Innovation in IT Services and Applications (iNEXT),
School of Computing and Communications, Faculty of Engineering and Information Technology
University of Technology, Sydney,
PO Box 123, Broadway 2007, Australia
Thawatchai.Chomsiri@student.uts.edu.au, {Xiangjian.He, Priyadarsi.Nanda, Zhiyuan.Tan}@uts.edu.au

Abstract— **In this paper, we propose a novel connection tracking mechanism for Tree-rule firewall which essentially organizes firewall rules in a designated Tree structure. A new firewall model based on the proposed connection tracking mechanism is then developed and extended from the basic model of Netfilter's ConnTrack module, which has been used by many early generation commercial and open source firewalls including IPTABLES, the most popular firewall. To reduce the consumption of memory space and processing time, our proposed model uses one node per connection instead of using two nodes as appeared in Netfilter model. This can reduce memory space and processing time. In addition, we introduce an extended hash table with more hashing bits in our firewall model in order to accommodate more concurrent connections. Moreover, our model also applies sophisticated techniques (such as using static information nodes, and avoiding timer objects and memory management tasks) to improve its processing speed. Finally, we implement this model on Linux Cent OS 6.3 and evaluate its speed. The experimental results show that our model performs more efficiently in comparison with the Netfilter/IPTABLES.**

Keywords— Firewall, Tree-rule firewall, Stateful firewall, Connection Tracking, Network Security

I. INTRODUCTION

The first generation of firewall was proposed in1990s [1] and applies packet filtering mechanism to detect unwanted packets based on packet header information (i.e., source IP address, destination IP address, and destination port) without considering their connection states. In the later development, a new type of firewall called 'Stateful firewall', which is the second generation of firewalls was suggested. This type of firewall considers connection states, such as; start of a new connection, being part of an existing connection, and not being part of any connection. The stateful firewalls operate faster and more secure in comparison with the packet filtering firewalls (stateless firewalls). Nowadays, almost all firewalls are stateful firewalls [2][3]. However, these traditional firewalls, including packet filtering and stateful firewalls, have a flat structure in terms of listing of firewall policy rules. These firewalls have to process packets by comparing packet header information against the listed rules sequentially. The process will be done if a match is found or the last rule is reached. This method gives a low performance compared to other searching algorithms. Apart from the speed problem, the traditional firewalls suffer rule conflicts, caused by shadowed rules and redundant rules [4][5][6][7], especially the shadowed rules. A shadowed rule is a rule which cannot be matched with any packets because all packets are already matched with other rules above it. This can cause a security problem due to some rules important in protecting against hackers and may be shadowed and not be used by firewall at all. Moreover, it is quite hard for firewall administrators to design listed rules for a traditional firewall and to protect a large network which requires a large number of rules without any rule conflicts.

In our recent studies [6][7], we proposed a new type of firewall to solve the aforementioned problems, and it achieves a high processing speed and fewer rule conflicts in large networks. In [6][7], we redesigned the firewall model to support two goals mentioned above. However, our redesigned model works only as a packet filtering firewall but not as a stateful firewall maintaining several connection states as suggested previously. In particular, the stateful (connection tracking) function includes an algorithm that can identify new and existing connections, as well as invalid ones which otherwise may raise a security concern. Thus, in this paper, we propose a novel connection tracking mechanism which will be used in the Tree-rule firewall [6][7].

The rest of this paper is organized as follows. Section II discusses the most relevant background and previous research works to this study. In particular, the problems with traditional List-rule firewalls are recapped in Section II-A, and the corresponding solutions suggested in the Tree-rule firewalls are reviewed in Section II-B. The design and analysis of a novel connection tracking mechanism are presented in Section III. The implementation of the proposed stateful mechanism and experimentations are detailed in Section IV. Finally, a conclusion is drawn in Section V along with potential future works.

II. BACKGROUND AND PREVIOUS WORKS

In this section, we present some of the problems with traditional firewalls and recap corresponding solutions used in our Tree-rule firewall model to avoid such problems. Below, the background information on packet filtering firewalls and stateful firewalls are first introduced.

A. *Traditional firewall (Listed-rule firewall)*

The traditional firewalls (Listed-rule firewall) filters packets by comparing their header information against a set of pre-defined firewall rules. This process operates rule by rule until a specific condition is reached. The rules in traditional firewalls are a list of condition statements followed by actions which are shown in Table 1 below.

Table. 1 An example of rule in Listed-rule firewall

Rule	Proto.	Source_IP	Destination_IP	Destination_Port	Action
1	TCP	192.168.1.1	54.251.1.1	80	ACCEPT
2	TCP	192.168.1.2	54.251.1.1	80	DENY
3	TCP	192.168.1.*	54.251.1.1	80	DENY
4	TCP	192.168.1.3	54.251.1.1	80	ACCEPT
5	TCP	192.168.2.*	54.251.2.3	80	DENY
6	TCP	192.168.2.4	54.251.2.*	80	DENY
7	TCP	192.168.3.*	54.251.3.5	80	ACCEPT
8	TCP	192.168.3.6	54.251.3.*	80	DENY
9	Any	Any	Any	Any	DENY

We have identified three key issues with the traditional firewalls in our previous studies [6][7]. They are

1. Security problem, caused by potential shadowed rules [6] and the change of meaning of the rule policy due to rule repositioning,

2. Functional speed problem, raised by shadowed rules [6], redundant rules [6] and sequential rule matching, and

3. Difficulty in rule design, in which one needs to carefully choose the proper positions for the firewall rules in order to avoid listing 'bigger rules'[7] before 'smaller rules'[7].

To overcome the aforementioned problems, we have proposed a 'Tree-Rule firewall' [7], where firewall rules are presented and operated in a tree data structure. A forwarding decision of an input packet follows the tree structure, so that the decision on the packet becomes faster.

B. *The Tree-rule firewall*

In this section, we recall the Tree-Rule firewall [7] and present its design in Figure 1. As discussed in [7], the advantages of the Tree-Rule firewall include:

- No shadowed rules,
- No redundant rules,
- No need of rule swapping because all rules will be sorted automatically,
- Ease of rule design with independent 'rule paths' [7], and
- High speed for packet decision.

With the Tree-rule firewall, network administrators can design a firewall rule easily using Graphical User Interface (GUI) that comes with the firewall. The firewall rules are organised in a tree shape as shown in Figure 1. They can define attributes for each column, and can add more attribute columns such as 'protocol type' and 'source port' columns. The attribute within the root node (the left column) can be Source IP address, Destination Port, or any attribute suitable to work with their networks. Ranges of numbers in each node will be sorted automatically without overlapping. Thus, administrators do not need to specify the position of a numbers so that this can avoid problems caused by rule swapping. The pointers between nodes can distinguish the 'rule paths' [7]. Therefore, no packet can match with two or more rule paths. This technique can avoid shadowing anomaly and redundancy anomaly.

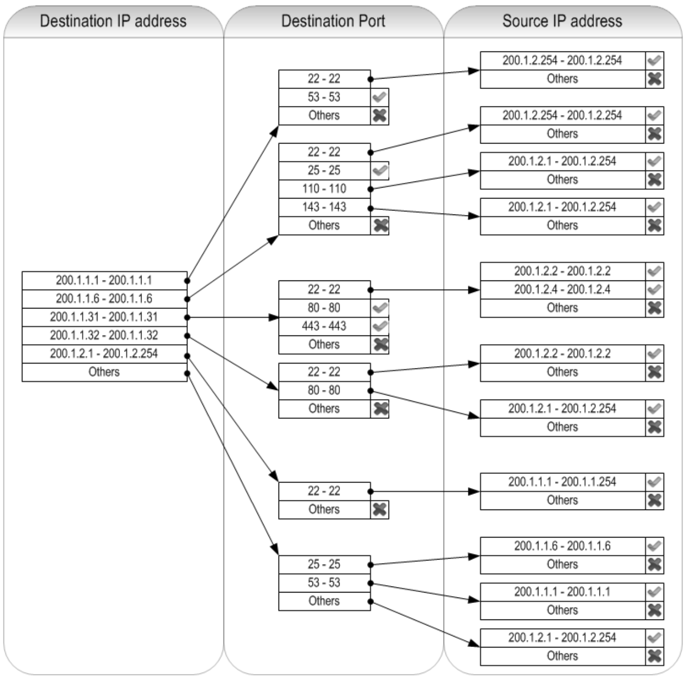

Fig 1. The design of Tree-rule firewall [7]

Apart from the users' view, the firewall also processes the rule on a tree structure as well. The firewall will take a corresponding attribute within packet header (i.e., Destination IP address) for searching in the root node first. Then, the firewall will verify packet's other attributes in order, by searching only on relevant nodes at the corresponding levels. As a result, the packet will be decided quickly with a specific action.

The next subsection will provide background knowledge on Packet filtering (Stateless) and Stateful firewalls.

C. *Packet filtering and Stateful firewalls*

Packet Filtering firewalls (Stateless firewalls) have been designed and created in the first decade of firewall age. These firewalls use an uncomplicated operation to examine incoming and outgoing packets. With this operation, each line of a set of firewall rules can regulate packet flow in one direction only [8]. In the context of Packet Filtering firewalls, Johansson and Riley [8] highlighted that a rule, which allows outbound access, requires a mirror rule [8] for allowing inbound access to permit replies to enter the network. Even though some Packet Filtering firewalls can add mirror rules automatically, most of these firewalls need firewall administrators to perform this task manually [9]. More importantly, Packet Filtering firewalls have a critical security issue because the mirror rule generated by the firewalls or even administrators can cause a big loophole, which allows attackers connecting into protected ports on protected computers of internal networks.

Similar to Packet Filtering firewalls, Stateful firewalls would examine packets' header (i.e., IP, TCP and UDP header). However, Stateful firewalls are little smarter because they understand TCP and UDP connection status by watching all connection states [8]. If clients behind a Stateful firewall create a new connection to communicate with web servers on the outside, the firewall will create inbound filters to allow relevant reply packets from the web server to the clients. The Stateful firewall will keep track states of network connection which travel across it. The firewall may check TCP sequence numbers for observing the connection status. Some examples of Stateful firewalls are Check Point Firewall-1 [10][11], Juniper Net Screen [12][13], and IPTABLES [14][15].

The IPTABLES is one of the most popular open source Stateful firewalls. There are many sources of documents which can guide one to understand the mechanism inside the Stateful firewalls. The article entitled "Netfilter's Connection Tracking System" by Pablo Neira Ayuso [16] gives useful information about a stateful mechanism within the Netfilter used by IPTABLES. This article reveals that the Netfilter uses a hashing algorithm to digest important packet header information (i.e., source IP address, destination IP address, source port, and destination port), and then push the hashed result into a hash table. To prevent hashing collision, the Netfilter uses buckets of linked lists to carry the hashed results. If a new connection is created, a firewall will compute a hashing result and find a location for it. If a reply packet reaches back to the firewall, the firewall will verify whether the packet belongs to an existing connection or not. Finally, an entry of a hash will be destroyed when the relevant connection is disconnected or expired. Thus, the first packet of connection will be examined against the firewall's rule list while other packets will be examined against the hash tables and buckets.

Considering the merits of stateful firewall models in configuration and security, we intend to enhance our Tree-based firewall via an integration of the mechanism of stateful firewall models into its system design. The details of the new stateful Tree-based firewall will be presented in the next section. The connection tracking model in our new stateful Tree-based firewall outperforms the one of Netfilter.

III. Design and Analysis

Our stateful Tree-rule firewall is designed based on the connection tracking model of Netfilter [16][17][18][19] with improvements. Similar to the connection tracking model of Netfilter, a 'Hashing Table' is involved in the connection tracking model in our stateful Tree-rule firewall. Comparatively, this newly designed model consumes less memory space (RAM) and operates in a faster manner. The details of this new model is discussed in Sections III-A to III-F.

A. *Using one node per connection*

Netfilter uses two nodes per connection for storing packet information (i.e., Source IP address, Destination IP address, Source Port, Destination Port, and Protocol Type), which is called 'tuple'. The first node (node 'x') as shown in Figure 2 (a) is used for recording information of packets transmitting from the 'original direction' (i.e., packets coming from the point that started the connection), while the second node (node 'y') is used for the 'reply direction' (i.e., reply packets going to the point that started the connection). In the Netfilter's model, the five important attributes of packet information are stored into nodes and are digested (hashed) using hashing function to obtain entry numbers (i.e., 'a' and 'b' in Figure 2 (a)). These numbers then indicate in which buckets the nodes are located. Thus, the two nodes related to the same connections are located on different entries. For example, Entry1 and Entry2 are entry numbers for the two nodes related to the same connection. They are calculated using the functions as shown below.

Entry1 = Hash(Source_IP, Destination_IP, Source_Port, Destination_Port, Protocol_Type, Key),

and

Entry2 = Hash(Destination_IP, Source _IP, Destination_Port, Source_Port, Protocol_Type, Key),

where 'Key' is a secret number generated by the firewall randomly.

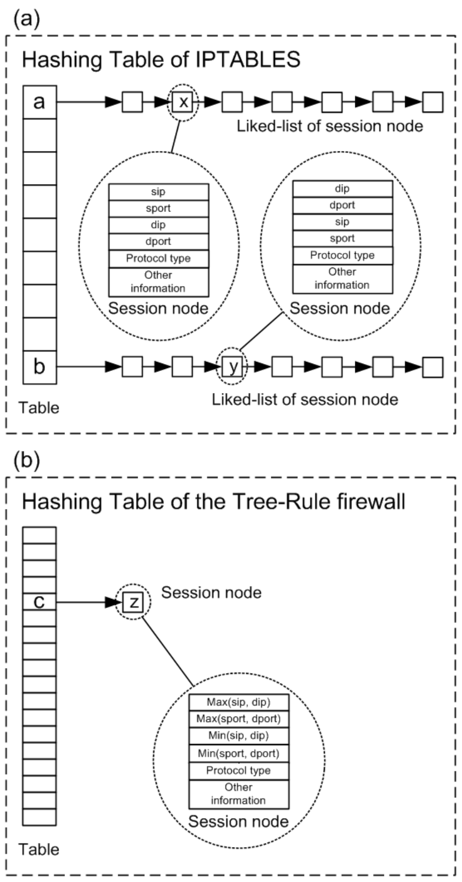

Fig 2. Hashing Tables of IPTABLES and the Tree-rule firewall

Differently, in our model, only a single node is used per connection using Max() and Min() functions. Consequently, we need only one entry number (i.e., 'c' in the Figure 2 (b)) which is obtained using the function below:

Entry = Hash(Max(Source _IP, Destination_IP), Min(Source_IP, Destination_IP), Max(Source_Port, Destination_Port), Min(Source_Port, Destination_Port), Protocol_Type, Key).

Note: Jenkins' hash (jhash) is used in both the connection tracking model of Netfilter and that of our stateful Tree-rule firewall. This is due to the reason that jhash provides fast operation and is capable of distributing its 32 output bits randomly.

By using our proposed scheme, approximately 50 per cent of the memory space can be saved. This allows our firewall to increase the number of concurrent connections up to two times more. In the Netfilter's model, the first packet in a new connection, as shown in Figure 3, needs to be calculated twice. The first calculation is for the 'original direction' node and the second calculation is for the 'reply direction' node (as shown in Figure 2 (a). Moreover, it has to allocate memory two times for the two nodes as well. In contrast, our model needs only one hashing calculation because we use only one node per connection. With the same reason, to terminate a connection needs two operations to delete the two nodes from memory, while our model takes only one node away. Thus, our model can reduce the processing time significantly.

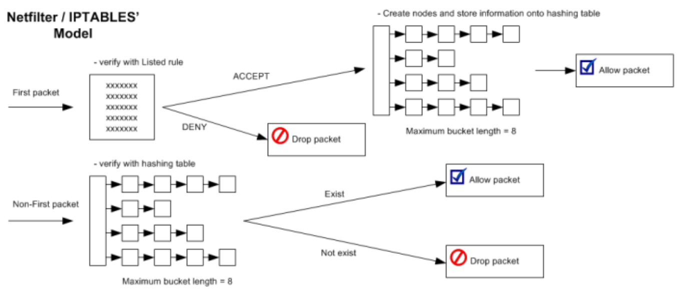
Netfilter / IPTABLES' Model
- Create nodes and store information onto hashing table
- verify with Listed rule
First packet
xxxxxxx
ACCEPT
DENY
Drop packet
Maximum bucket length = 8
Allow packet
- verify with hashing table
Non-First packet
Exist
Allow packet
Not exist
Drop packet
Maximum bucket length = 8

Fig 3. Steps of Netfilter's connection tracking [16]

Additionally, in our model, the number of connections handled by our stateful Tree-rule firewall can reach the maximum value. This is different from the Netfilter's model, which has a limitation on the use of pair nodes. Given a pair nodes in the Netfilter's model with a new connection (as shown in Figure 3), after calculating Entry1 and Entry2 the Netfilter looks for spaces to place the two nodes by checking bucket lengths. If the relevant buckets are close to be full or already full, then the length of the two buckets can be categorized into three cases (A, B, and C):

A. 7 and 7

B. 8 and 8

C. 7 and 8

Note:

- The maximum bucket length of Netfilter's model is 8, and
- Assuming that Jenkins' hash is used in Netfilter's model and our model to provide random output bits.

In case A, luckily, there are memory spaces for the two nodes. If the two nodes are allocated to the tails of the buckets, the lengths of the buckets will now both be 8s exactly. In case B, unluckily, the two relevant buckets are already full. No more nodes can be appended to the tails of these buckets. In case C, there is only a space for one of the pairs but not enough for two. In this case, Netfilter's model are not able to use memory space efficiently because maximum bucket length in average, for this case, is 7.5 but not 8.

One may argue that our model has to calculate Max and Min for the Source IP address, Destination IP address, Source Port, and Destination Port, for every packet. As such, these operations may consume additional CPU load. To respond to this argument, we have created a small program for measuring time consumption of Jenkins' hash function and the Max/Min functions. It is found that the computation of Jenkins' hash applied in Netfilter's model takes approximately 1,000 times longer than those of the Max and Min functions. Our model requires hashing calculation once only and four times of Max and Min calculations, while Netfilter's model makes hashing calculation twice and no Max and Min calculations. Overall, our model achieves a better performance in terms of speed.

B. Expanding Hashing Table size vertically

In this section, we propose to improve Netfilter's model by expanding the size of Hashing table vertically (as shown in Figure 4). This method can reduce hashing collisions and further improve the speed of operation. We decrease 'maximum bucket length' and then increase the number of entries in Hashing table with the same memory space. For example, if we double the size of Hashing table vertically, we need to decrease maximum bucket length by half of the size (i.e., from 8 to be 4). Hence, by increasing the size of Hashing table by two times, the collision will be decreased approximately by 50%. This percentage is reasonable for decreasing maximum bucket length to half. In our model, in fact, we use one node per bucket and expand the Hashing table vertically to 8 times or more. However, the size of the hashing table can be expanded to 16, 32, 64 times or more because our model requires smaller node size than that used in Netfilter's model.

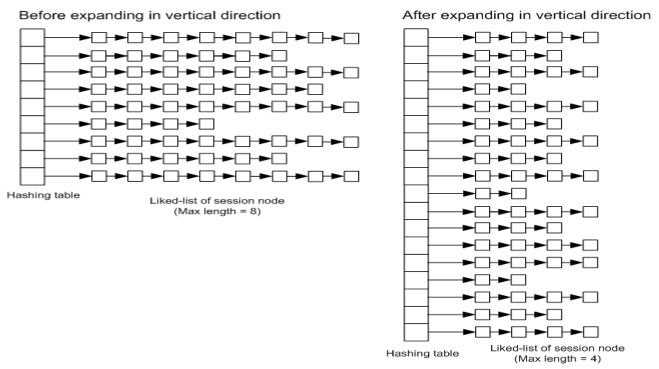
Before expanding in vertical direction
Hashing table
Liked-list of session node (Max length = 8)
After expanding in vertical direction
Hashing table
Liked-list of session node (Max length = 4)

Fig 4. Expanding hashing tables in vertical direction

It is worth emphasizing that a collision may still occur because there will be a chance to have two different packets associated with the same hashed output. However, expanding the size of the Hashing table in vertical direction as proposed in this section can reduce probability of collisions generated by a hashing function. This method can handle more concurrent connections and allows shorter bucket lengths for the same memory space, and the same chance of collisions. Consequently, time consumption for a sequential search within buckets will be decreased automatically.

Considering systems A and B which have different sizes of Hashing tables (the size of A is bigger than the size of B). If distribution of nodes of the two systems is random, and increasing rates of nodes in the two systems are the same, we have found that the bucket length in A will be shorter than the bucket length in B. Consequently, searching for nodes in system A will be faster than B . Thus, we can conclude that expanding size of Hashing table vertically can reduce searching time within the buckets.

Although expanding the size of a Hashing table vertically requires more spaces for the pointer of each expanded entry because each entry must point to its first node, these pointers require little memory space compared to the size of nodes.

In the Netfilter's model, one node requires 350 bytes [20], while one pointer requires only 8 bytes. In our model, one node requires 40 bytes while one pointer requires only 8 bytes. The number of bytes used for each node has been reduced by 8.75 times.

C. *Using one node per bucket*

It is apparently that a short bucket length requires less time for a sequential search in comparison with a long bucket length (as shown in Figure 4). Thus, we decide to use one node per bucket in our model (as shown in Figure 2 (b)) instead of using 8 nodes per bucket as presented in the Netfilter's model. However, to mitigate the collision problem, we will expand the size of Hashing table vertically by at least eight times.

With this scheme, we can avoid the sequential search within buckets, and provide a better speed performance. This scheme also results in less memory consumption because no left and right pointers are required to be maintained in a node to work as doubly linked list (i.e., in the Netfilter's model [16]). Consequently, the node size can be reduced. Moreover, this decrease of complexity provides an easy way to implement or create a efficient firewall.

Some users may have negative thinking about the collision problem on our model (maximum node per bucket = 1) because they may believe that collisions can be mitigated using only buckets. In fact, we have already mitigated the collision problem by expanding the size of Hashing table vertically. Reducing bucket length from 8 to 1 can be compensated by extending the Hashing table by 8 times. Moreover, we can extend it up to 64 times with the same amount of memory space used in Netfilter because our nodes are smaller than Netfilter's nodes (our node size = 40 bytes, Netfilter's node size = 350 bytes).

D. *Verifying non-first packets using Tree Rule before Hashing Table*

Generally, in firewalls including Netfilter, the 'rule set' created by firewall administrators will be used only for the first packet of a connection (as shown in Figure 3.). The first packet of a connection is verified with the 'rule set'. If the result is ‘ACCEPT’, then the firewall will create corresponding nodes within the 'Hashing table'. Then, the subsequent packets, namely the second packet, third packet and so on, will be verified with the entries in the 'Hashing table' instead of the 'rule set'. Therefore, it can be seen that verifying packets using Hashing table will be done more often than verifying packets using the rule set. From our study, we have found that hashing calculation takes approximately 1,000 times in comparing two numbers used in a verifying rule set or Tree rule. This is because hashing function is very complex and involves a complicated computing procedure. In contrast, verifying packets using rule set does not take much time because it requires only comparison between a pair of numbers.

However, there are some attack packets that are deliberately designed as non-first packets and injected into host connections. An good example of such type of attack is the 'Reset Flood Attack' [21][22][23], where packets are generated by worms randomly and contain no SYN flag. These attack packets will be verified by using the Hashing Table. If these packets are verified by using rule set (or Tree rule) and are dropped, it can reduce CPU time because verifying packet with rule set is similar to comparison between numbers which is faster than using hashing function. We can drop packets which are denied by rule set (or Tree rule) without using the Hashing Table again because if the first packets are denied by rule set, packets' information will have no chance to appear in the Hashing Table. In this method, we will consider packets from both directions.

Therefore, in our model, we will verify non-first packets using Tree Rule first for both directions (as shown in Figure 5). If at least one of results of this test is ACCEPT, we will then verify using the Hashing Table.

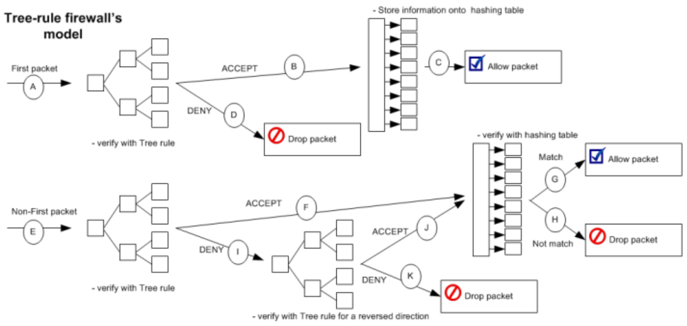

Fig 5. Verifying non-first packets using Tree Rule before Hashing Table

As such, the proposed firewall can operate faster in the case of attack using random packets because approximately 50% of randomly generated packets will be the non-first packets which will be denied by firewall rule as it can be seen in the bottom flow of packets in Figure 5 (i.e., the path E→I→K). In this case, by eliminating the unnecessary hashing calculation, the overhead of our proposed stateful Tree-rule firewall in packet filtering is further reduced.

In the case of attack where the number of non-first packets is greater than the number of first packets, if the verified result of a Tree rule is 'ACCEPT' (i.e., the path E→F or the path E→I→J shown in Figure 5), the proposed firewall will need only little extra time to verify packets using Tree rules before verifying packets using the Hashing table. However, the overall time consumption is only slightly increased because verifying packets using Tree rules takes very little time and can be ignored compared to verifying packets using Hashing table.

E. Use of Static Node and Label to identify free nodes

In the Netfilter's model, if a new connection is created, firewall must prepare memory spaces for relevant nodes (i.e., using the 'kmalloc()' function in C language). Also, if the connection is terminated, the firewall must return these memory spaces back to OS (i.e., using the 'kfree()' function). These operations involve time factor performing these operations. Considering a normal cycle for the HTTP request and reply, which use TCP connection with approximately five relevant packets (in average), firewall called the 'kmalloc()' function and 'kfree()' function twice generating a total of four memory related operations which is high compared to the number of relevant packets (five packets). To resolve this problem, we develop our model using a static memory. The node which is used for recording packet information should be a static node and will be created when the firewall is loaded and executed. If the connection is created, packet information will be stored in the node, and the 'Label' of this node will be marked to be 'Unavailable'. If the connection is terminated, the Label of this node will be marked to be 'Available' without destroying the node from memory. If the next connection which has the same entry number is created, the firewall can use this node immediately (without requesting further memory space from OS for the new node) by overwriting information of the new packets to this area of memory, and then change the Label to be 'Unavailable'. With this method, verifying packets can be done easily by considering Label and packet information within the node. In implementing the above scheme, the variable type for the Label is a one byte variable (actually it requires only one bit '0' if the node is 'Available' and '1' if the node is 'Unavailable').

In the context of computer programming, each function including memory management function affects run time. With our scheme, the firewall does not need to request and return memory space from and to the OS for nodes (i.e., calling kmalloc() and kfree()). As a result, the proposed firewall can operate faster.

Using our scheme, firewall administrators must prepare a certain (fix) but a very small size of memory for the firewall software, and this memory space cannot be shared with other processes although the firewall is working with only few packets.

Based on our calculation, a 768 MB RAM can handle 16.7 million connections. This provides approximately 46 bytes per connection for our firewall model. According to the information presented on the Netfilter's website [15], the default value for maximum connection of IPTABLES is 8192, which requires 128 MB of RAM. Therefore, it requests 16,384 bytes per connection for Netfilter's model. Hence, our scheme can carry a total of 16,384 / 46 = 356 times more connections compared to the Netfilter's model.

F. Using the Label for Time Out instead of Timer Object

In the Netfilter's model, if a connection is terminated by FIN or RST flags, firewalls will create a 'Timer Object' to schedule and specify the time when the relevant nodes will be deleted. For example, the Timer Object will delete a node in the next 30 seconds since the FIN flag was detected. This operation uses event driven techniques. Creating, deleting and managing Timer Objects consumes OS resources and CPU times. These resources and time consumptions are significant when compared with a short connection (i.e., normal use of web browsing / HTTP protocol). Thus, our model removes this drawback by using a Label for expired nodes instead of using the Timer Objects. With this method, if a firewall find FIN or RST flags, it will calculate expiration time for the node and store it into the Label. A variable type for this Label is the 'struct timespec' (in C language) which requires 16 bytes, and can record date and time in seconds, and even nanoseconds. To verify the node, the firewall will check the timer's Label in the node before using information from the node. If the current time is new and greater than the time stored in the Label, it means that this node is expired (the node does not exist). In contrast, if the current time is less than the time stored in the Label, it means that this node is still active, and the firewall can use information from this node.

Firewalls can work faster because they do not necessarily manage timer objects such as calling the 'new_timer()' and 'delete_timer()' functions. Moreover, this scheme can save OS resources and easy to implement.

IV. Firewall Implementation and Experimental analysis

We implement the Tree-rule firewall using C language on Linux Cent OS 6.3, and conduct experiments on real network environments. In our previous research [6][7], it has been shown that the Tree-rule firewall provides higher security in comparison with traditional firewalls. In this paper, however, we focus on speed issue along with our stateful (connection tracking) mechanism. We emphasize on the following four cases of packets in TCP connections (as shown in Figure 6.):

- Case #1: The fist packets (SYN packets) accepted by the firewall,
- Case #2: The fist packets (SYN packets) denied by the firewall,
- Case #3: The non-fist packets (Non-SYN packets) accepted by the firewall, and
- Case #4: The non-fist packets (Non-SYN packets) denied by the firewall.

Case #4 can then be further divided into two sub-cases, Case #4a and Case #4b for the following:

- Case #4a: The non-fist packets which will be denied by the firewall and verified with hashing table, and
- Case #4b: The non-fist packets which will be denied by the firewall but not be verified with hashing table.

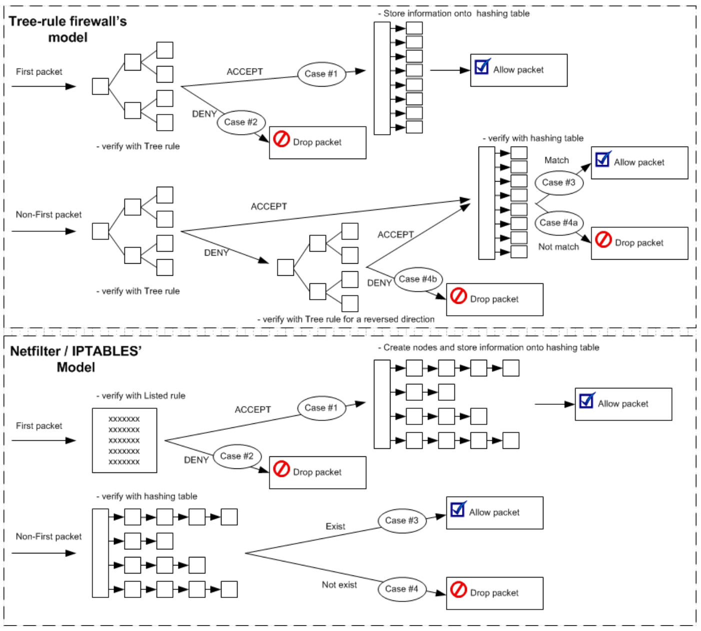

Fig. 6. The four cases which was evaluated on speed issue

On client machines, apart from a normal internet browsing, we use BackTrack 5 R3 live boot flash to generate packets and use 'hping3' command to generate the first packet of connection with '-S' parameter. The detailed command is given below.

```
# hping3 192.168.22.2 -a 192.168.11.2 -p 333 -S -s +1 -d 1440 -i u1000
```

This command can be used for Cases #1 and #2 because the '-s +1' parameters will generate packets of which their source ports will be starting from '1', and increased by one, for each packet. In addition, we use '-d 1440' to specify packet size of 1440 bytes (not including layer-2 header and tail size) because this number is a regular size for the HTTP protocol. With this tool, we can create and send packets as much as possible using the '--flood' parameter, and can specify time interval for each packets using the '-i' parameter. For generating packets related to Cases #3 and #4, we will use this command without '-S' parameter to turn off the SYN flag.

We compare the processing speed of the connection tracking module in our stateful Tree-rule firewall with that of the connection tracking module in Netfilter. Figure 6 shows the four cases on the Tree-rule and IPTABLES/Netfilter. We also measure the processing time on packet decision process for one packet. We start to record packet arrival time on the first line of the packet hooking function (i.e., the function 'hook_func()' of the link [24]). We then record the end time before the line 'return NF_ACCEPT' and 'return NF_DROP' for all cases. Programmed codes between start time and end time for each case, give different time consumptions because some cases need to compute hashing function while other cases do not. The recorded time in nanosecond will be written on the file '/var/log/messages' using the 'PRINTK' function. Likewise, we also measure computation time of packets on Netfilter/IPTABLES' module by using this technique. We modify the file '/kernel/net/netfilter/nf_conntrack_core.c' of the Netfilter's ConnTrack module, and recompile it (including relevant files) to create the new 'nf_conntrack.ko' file. We reload the modified modules to operate with IPTABLES. Information about starting time and ending time for the four cases of our prototype software and Netfilter/IPTABLES are collected and their average is computed. We test them on real network environment with more than 50,000 packets before taking average for every case. Our experimentations use approximately 50 client PCs in an university network LAB. These computers (firewall and clients) use Intel Core i5 with 2.30 GHz of CPU, and 4GB of RAM. We test with 50 rules for IPTABLES and approximately 50 rule paths for Tree-rule firewall. The experimental results are presented in Table 2 and Figure 7.

Table 2. Time consumption of packet decision on Tree-rule firewall and Netfilter / IPTABLES

Firewall	**Processing Time for one packet (nanosecond)**				
	Case 1	**Case 2**	**Case 3 (average)**	**Case 4a (average)**	**Case 4b**
Tree-Rule firewall	481.13	164.44	497.69	455.12	239.94
Netfilter/IPTABLES	3,722.82	1,259.94	3,319.55	3,193.41	3,193.41

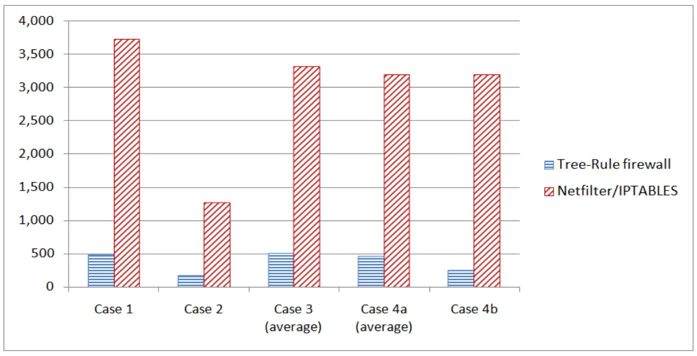

Fig. 7. Representation of time consumption on Tree-rule firewall and Netfilter/IPTABLES

The experimental results of the Tree-rule firewall shown in Table 2 and Figure 7 reveal that Case #3 takes little more time than Case #1, because Case #3 needs to verify packet header with a tree rule for one or two times before verifying the packet information with the hashing table. This slight time difference is due to the fact that verifying packet with the tree rule is an easy job in comparison with calculating a hashing function. Case #2 and Case #4b of the Tree-rule firewall take less time compared to other cases because they do not need to perform the hashing function. Theoretically, Cases #3 and #4 of Netfilter/IPTABLES should take equal time because they use same algorithms. However, in practical, they may encounter different time of computations. The largest time used by Netfilter/IPTABLES is the time in Case #1, where the firewall has to perform hashing task two times (for two directions), and also requests memory spaces from OS to create two nodes for storing connection information for both original and reply directions.

Apart from time consumption on packet decision, we also measure memory consumption of both the Tree-rule firewall and Netfilter/IPTABLES. Theoretically, Netfilter require 350 bytes per connection [20], while our model (Tree-rule firewall) requires only 40 bytes per connection. The Tree-rule firewall uses less memory because it does not use timer objects and memory spaces for a reply-direction nodes. However, our experimental results show that Netfilter takes approximately 400-430 bytes per connection, whereas our model takes approximately 40-60 bytes per connection. These numbers are based on 10,000 concurrent connections with normal traffic load on real network.

V. CONCLUSION AND FUTURE WORKS

In this paper, we propose a stateful mechanism which will be used on the Tree-rule firewall. We design and develop our model from the basic connection tracking model of Netfilter. Our proposed connection tracking model requires low memory space, while it can handle more concurrent connections by using one node per connection. Our scheme extends the hashing table vertically and uses one-node bucket length. Moreover, this model has a low time consumption, which benefits from the avoidance of hashing computation by considering the Tree rule first. If packets are not corresponding to the Tree rule (for both directions), the firewall will not perform hashing calculation. We also get rid of the timer objects used for closing connections. Instead, we use the Label which can tell the firewall which connections are expired by reading only 16 bytes of time information from the memory. Moreover, this model does not necessarily create new nodes to store packets' information because we use static memory for all nodes created after the firewall is loaded into the memory and executed. With this scheme, the firewall only checks whether the relevant node is free or not by using the marked Label. Our experimentation is conducted on real network environments, and the results show that our stateful Tree-rule firewall operates faster than Netfilter/IPTABLES. Our stateful Tree-rule firewall also requires less memory owing to the fact that the size of a node has been reduced and the number of nodes used per connection has been decreased from two to one. In future, we plan to consider FIN and RST packets and their effects on the Tree-Rule firewall implementation. Our stateful Tree-Rule firewall will be developed and tested for Network Address Translation (NAT), IPv6 and Virtual Private Network (VPN) in future.

REFERENCES

[1] W. Cheswick, S. Bellovin, A. Rubin, Firewalls and Internet Security: repelling the wily hacker, Addison-Wesley Professional, 2003.

[2] J. Garcia-Alfaro, F. Cuppens, N. Cuppens-Boulahia, S. Martinez, J. Cabot, Management of stateful firewall misconfiguration. Computers & Security, Elsevier, 39 (2013) 64-85.

[3] S. Paul, R. Jain, M. Samaka, J. Pan, Application delivery in multi-cloud environments using software defined networking. Computer Networks, Elsevier, (2014) In Press.

[4] E. Al-Shaer, H. Hamed, Firewall policy advisor for anomaly detection and rule editing, in: Proceedings of the IEEE/IFIP Integrated Management, IM, 2003, pp. 17–30.

[5] A. Ashfaq, S. Rizvi, M. Javed, S. Khayam, M. Q. Ali, E. Al-Shaer, Information theoretic feature space slicing for statistical anomaly detection, Journal of Network and Computer Applications, Elsevier, 41 (2014) 473–487.

[6] T. Chomsiri, X. He, P. Nanda, Limitation of listed-rule firewall and the design of tree-rule firewall, in: Proceedings of the 5th International Conference on Internet and Distributed Computing Systems, China, 2012, pp. 275–287.

[7] X. He, T. Chomsiri, P. Nanda, Z. Tan, Improving cloud network security using the Tree-Rule firewall, Future Generation Computer Systems, Elsevier, 30 (2014) 116-126.

[8] J. Johansson, S. Riley, Protect Your Windows Network: From Perimeter to Data (Microsoft Technology), Addison-Wesley Professional, 2005.

[9] What are the risks associated with relying on IPSec IP Filtering?, 2014, http://security.stackexchange.com/questions/3909/what-are-the-risks-associated-with-relying-on-ipsec-ip-filtering.

[10] F. Cuppens, N. Cuppens-Boulahia, J. Garcia-Alfaro, T. Moataz, X. Rimasson, Handling stateful firewall anomalies, Information Security and Privacy Research, Springer Berlin Heidelberg, (2012) 174-186

[11] Administration Guide - Check Point, 2007, http://updates.checkpoint.com/ID/CheckPoint_UTM-1_AdminGuide.pdf.

[12] S. Kumar, R. Gade, Experimental Evaluation of Juniper Network's Netscreen-5GT Security Device against Layer4 Flood Attacks, Journal of Information Security, Springer Berlin Heidelberg, 2 (2011) 50.

[13] NetScreen-Security Manager: Configuring Firewall/VPN Devices Guide, 2007, http://www.juniper.net/techpubs/software/management/security-manager/nsm2007_1/nsm2007_1_device_config.pdf.

[14] R. Rosen, Netfilter, Linux Kernel Networking, Apress, (2014) 247-278.

[15] The netfilter.org project, 2014, http://www.netfilter.org/.

[16] P. Ayuso, Netfilter's Connection Tracking System, LOGIN;, The USENIX magazine, 32 (2006) 34-39.

[17] The netfilter.org project, 2014, http://www.netfilter.org/.

[18] Q. M. AL-Musawi , MITIGATING DoS/DDoS ATTACKS USING IPTABLES, International Journal Of Engineering & Technology, IJENS Publishers, 3 (2012) 12.

[19] Q. X. Wu, The Research and Application of Firewall based on Netfilter. Physics Procedia, Elsevier, 25 (2012) 1231-1235.

[20] Netfilter/IPTABLES FAQ: Problems at runtime, 2014, http://www.netfilter.org/documentation/FAQ/netfilter-faq-3.html.

[21] A. Anand, B. Patel, An Overview on Intrusion Detection System and Types of Attacks It Can Detect Considering Different Protocols, International Journal of Advanced Research in Computer Science and Software Engineering, IJARCSSE, 8 (2012) 94-98.

[22] Malicious Packets - Packet-Craft.net, 2014, http://www.packet-craft.net/Malicious/.

[23] A. Srivastava, B. B. Gupta, A. Tyagi, A. Sharma, A. Mishra, A recent survey on DDoS attacks and defense mechanisms. In Advances in Parallel Distributed Computing, Springer, (2011) 570-580.

[24] P. Kiddie, Creating a simple 'hello world' Netfilter model, 2009, http://www.paulkiddie.com/2009/10/creating-a-simple-hello-world-netfilter-model.