vpFirewall: Virtual Parallel Firewall

Kerim Gokarslan
CPSC 424/524: Parallel Programming Techniques (Fall 2018)

December 19, 2018

1 Introduction

A Firewall is one of the essential components of a network system, which implements the security policy of the system in different network layers. Although the first firewalls are more hardware oriented, nowadays, most of the firewalls are virtualized and implemented in software level. This project aims to do research the current implementation of different firewalls in Linux systems and proposes a user-layer multithreaded virtual firewall implementation to leverage parallel processing. We implemented a state-of-art user-space firewall running on Linux using pthread with a fixed-size thread pool. This firewall, vpFirewall, consists of two layers: a Linux kernel plugin based on NetFilter[1], which basically queues every ingress and egress IP packets and a user-space program to decide on the verdict for each packet. The paper is organized as follows. Section 2 describes the design of the firewall and implementation details. Section 3 describes our evaluation methodology and results. Section 4 presents our conclusion. Finally, Section 5 discusses future work.

2 Design and Implementation

vpFirewall has two main parts: The kernel plugin and the user-space program. The kernel plugin is mainly responsible for queuing incoming or outgoing packets using NetFilter, where the user-space program reads the packets from the queue and decide on verdict using preloaded firewall rules.

NetFilter, gcc and NetFilter Queue are required to compile and run vpFirewall. NetFilter comes by default on most of the Linux distributions having kernel version newer than 2.3. The following command is an example of installation of packets required for running vpFirewall using yum packet manager.

```
sudo yum -y install gcc libnetfilter_queue libnetfilter_queue-devel kernel-devel
```

vpFirewall can be run on any Linux distribution supporting NetFilter, and we mainly tested it on CentOS 7.

[1] `https://www.netfilter.org/`

2.1 The Kernel Plugin

The kernel plugin is responsible for queuing ingress or egress IP packets using NetFilter. It is implemented in **vp_firewall_kernel.c** and it can be compiled and loaded as follows:

```
make vp_firewall_kernel && insmod vp_firewall_kernel.ko
```

Once it is loaded, the user space program needs to run immediately, since this plugin queues every packet coming to or outgoing from the host. In the evaluation part, we simply allowed SSH packets (i.e. TCP packets coming to port 22 or going from port 22) for testing purposes. The kernel plugin can be removed by running the following command:

```
rmmod vp_firewall_kernel
```

2.2 The User-Space Program

The main functionality of vpFirewall is implemented within the user-space program. The main program is implemented in **vp_firewall**, which simply connects to the NetFilter queue, and registers the packet handlers. **vp_firewall_pthread** is one of the packet handlers, which based on pthread structure and utilizes a fixed-size thread pool. The NetFilter Queue handler runs on the main thread and whenever there is a packet on NetFilter Queue, it puts it to the thread-safe queue we implemented on **packet_queue.c**, where each thread waits on that queue. We used pthread mutex and pthread signal & wait for adding a packet to the queue or removing a packet from the queue.

We also implemented a simple firewall rule mechanism similar to **firewalld**, and the firewall rules needs to be loaded when the user-space program is started, and the default path for the rules is **vp_firewall.conf**, and the code in **vp_firewall_load** is responsible for parsing the rules. vpFirewall supports the following firewall rules:

1. -A

   A option specifies the direction of the packets and it is mandatory. It can be **INPUT**, **OUTPUT** and **FORWARD**.

2. -p

   -p option specifies the transport layer protocol. It can be **tcp**, **udp** or **icmp**.

3. -d or –dest and -s or –source

   -d option specifies the destination IP address and it supports CIDR notation. -s option, on the other hand, specifies the source IP address supporting CIDR.

4. –dport or –destination-port and –sport or –source-destination

   –dport specifies the destination port and –sport specifies the source port.

5. -m

   -m specifies matching protocol. For simplicity, we only implemented -m MAC.

6. –mac-source and –mac-destination

   –mac-source specifies the source MAC address and –mac-destination specifies the destination MAC address. Note that if -m mac does not specified the source and destination will not be used.

7. -j

   -j specifies the action to be taken for the matching packets with the rule. There are three options **ACCEPT**, **DROP** or **REJECT**. NetFilter takes the similar action for dropping and rejecting, so the **DROP** or **REJECT** options corresponds to the same action.

The following is an example configuration file having four different firewall rules. Note that, vpFirewall picks the first matching rule, that is, the order of the rules specify the priority of the rules.

```
−A OUTPUT −p tcp −d 192.168.100.0/24 −−dport 22 −j ACCEPT
−A INPUT −p tcp −m tcp −−dport 8000 −j DROP
−A INPUT −p tcp −m tcp −−dport 8080 −j ACCEPT
−A INPUT −p tcp −−destination−port 22 −m mac −−mac−source 00:0F:EA:91:04:07
    ↪ −j ACCEPT
```

The user-space program can be compiled and running by executing the following command:

```
make vp_firewall && ./vp−firewall
```

3 Evaluation

We evaluate vpFirewall on a virtual machine on Google Cloud having 24 vCPUs (Intel Haswell), 64 GB RAM and CentOS 7 with Linux Kernel 3.10. We also provide a VirtualBox image for testing proposes which can be accessed at `http://cloud.com`.[2] We used iPerf3[3] (with TCP packets) for testing throughput of the system.

We tested vpFirewall with 1, 10, 25, 100, 500 and 1000 firewall rules and thread pool of size 1, 4, 8, 16, 24, 32 and 64. Figure 1 shows each throughput result for the combination of firewall rules and number of threads. We clearly show that using multithreaded firewall increases the throughput of the firewall comparing to single threaded version up to 20%. Moreover, multithreaded vpFirewall has better results when there are more rules in the firewall, as it is shown in Figure 1. Another important result of this implementation is the fact that increasing the number of threads when there are few threads existing has more impact on throughput than increasing number of threads when there are more than 8 threads.

[2]Please send me an email if you want to test Google Cloud VM instead of VirtualBox.
[3]https://iperf.fr/

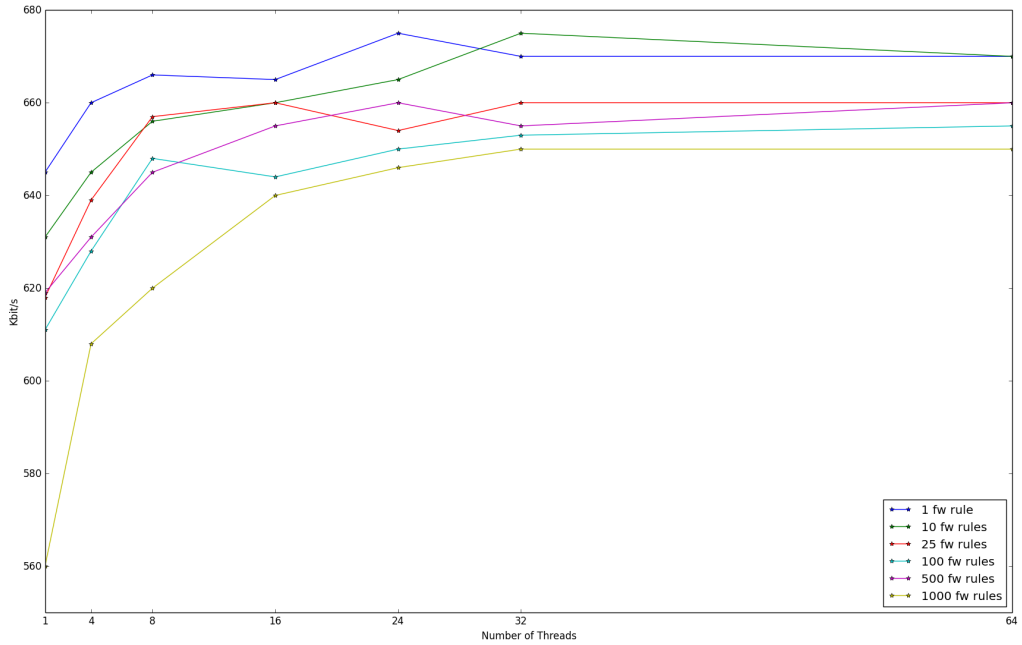

Figure 1: Throughputs for different firewall configurations and number of threads.

We also implemented the thread-per-packet strategy, however, it is not stable especially when the network is busy. As we have shown in the thread-pool implementation, using a few threads for vpFirewall (e.g. 4 threads) has better results and less overload on the system.

4 Conclusion

vpFirewall is a state-of-art multithreaded virtual firewall, which aims to show that multithreading increases the network throughput of the system. We show that multithreading can increase the performance up to 20%, and we believe that this can be improved much higher levels with different implementation techniques, such as not using NetFilter Queue or implementing a kernel plugin based firewall. We also concluded that using a small-sized thread pool enough for a good performance, for example, a thread pool of size 4, since it has the best price over performance ratio, where the price is allocating threads to vpFirewall and performance is the throughput of vpFirewall.

5 Future Work

As it is described in the previous sections, vpFirewall mainly focuses on showing the fact that a multithreaded virtual firewall has advantages over a single-thread virtual firewall. Although we used NetFilter and implemented a user-space firewall, a multithreaded kernel plugin or a multithreaded user-space program without using NetFilter can be implemented

to increase overall throughput. Moreover, different parallel programming tools like OpenMP or MPI can also be used for parallelism, instead of pthreads. Even, GPUs can be used for such purposes, for example, NVIDIA GPUDirect[4] is a good candidate for implementing a virtual firewall, since it can connect to Network Interface Card (NIC) directly from GPU.

[4] `https://developer.nvidia.com/gpudirect`