Acceleration of IPTABLES Linux Packet Filtering using GPGPU

Keyvan Karimi¹, Arash Ahmadi², Mahmood Ahmadi², and Bahram Bahrambeigy¹

¹ Department of Information Technology, Kermanshah Science and Research Branch, Islamic Azad University, Kermanshah, Iran
{kaiwan.karimi, bahramwhh}@gmail.com
² Engineering Faculty, Razi University, Kermanshah, Iran
{aahmadi, m.ahmadi}@razi.ac.ir

Abstract. Firewalls are a piece of software or hardware that control access to organization networks. Packet filtering is placed in the heart of firewalls. It is performed by comparing each data packet against a rule set. In the high bandwidth networks, filtering becomes a time consuming task. In this situation, the packet filtering firewall can reduce the overall throughput and become a bottleneck. To solve this problem a wide range of researches have been done to improve overall throughput of the packet filtering firewalls. In this paper, the first matching rule mechanism of Iptables is implemented in user-space by employing parallel processing capability of Graphics Processing Unit (GPU). The results show that CPU-GPU accelerated code brings significantly higher throughput over the CPU version of Iptables code. The overall throughput of packet filtering on GPU for 10,000 rules is about 400,000 Packets Per Second (PPS) which is 43 times faster than inefficient first matching rule algorithm of Iptables on CPU.

Keywords: GPU, Iptables, Packet filtering, Parallel processing.

1 Introduction

Packet filtering firewall takes its decision to accept or drop the packets only by investigating the header portion of each incoming or outgoing packet, which contains essential networking information including source IP address, destination IP address, protocol, port numbers, etc. Firewall performs filtering task by comparing the packets against rules of the filter set. Many organizations and corporations need to connect the internet to do their current activities. By growing the size of organizations and diversity of the users who connect to the Internet, along with new increasing services which are introduced by these organizations, it is an essential requirement to have more precise control on every aspects of the incoming and outgoing traffic. Accordingly, filtering rule sets are becoming more complicated in terms of interrelations and capacity, which results in a higher computational complexities. Modern firewalls use first-match semantics [3,4,5]. Today most of the firewalls use two mechanisms for packet matching: 1) linear search that implements the first match and 2) fast state
table lookup that uses hash table and search tree to bypass the rule set. Open source firewalls pf [6] and Iptables [7] use both aforementioned approaches. Linear search over a large rule set can become very inefficient and make the packet filtering a point of failure, because each packet needs to be checked against every rule until the first matching rule is found. The second method, on the other hand, is more suitable for long TCP connections in which fast state table lookup handles most of the packets. However, stateless connections such as UDP, ICMP traffic and short TCP flows, have to use the rule set and linear search similar to the first method.

In this paper, NFQUEUE target of Linux Iptables packet filter is used in order to send the packets to user-space and process them by parallel processing capability of Graphics Processing Unit (GPU). The independence nature of the packets is well suited for the parallel processing approaches. In this study, a fully functional packet filter is created by Linux Iptables packet filter, where the throughput of the proposed method is tested on a laboratory test-bed.

Our parallel design of Linux packet filtering can reach the rate of 400,000 PPS, with 10,000 rules. Iptables could only process about 2500 PPS with the same rule set [2] and on our test-bed PC, the sequential code, it could process about 9000 PPS but the authors used a slower machine than ours.

The rest of the paper is organized as follows: Section 2 is related works. Section 3 describes Linux Netfilter framework and Iptables tool to define chains of rules for packet matching. In section 4, we discuss a little about GPU and how GPU is used in high performance general-purpose applications. Section 5 describes our user-space packet-filtering approach which uses parallel computing capability of GPU to accelerate the code. In Section 6 results are presented and discussed. Section 7 is conclusion and future works.

2 Related Works

Rovniagin et.al. [2] proposed a geometric matching algorithm that its worst-case space complexity is $O(nd)$ where $n$ is the number of rules and $d$ is the number of fields to match. This algorithm is integrated into the Linux Iptables firewall codes and has been tested on the real traffic loads. The results show that this algorithm can filter over 30,000 PPS on a standard PC with 10,000 rules.

Employing GPU for packet filtering is introduced in [1] and the results is obtained in packet delay per number of firewalls and per number of rules. Unfortunately, the delay that API causes to exchange packets from kernel-space to user-space is not considered. It only calculates the raw elapsed time that packets spend to process on GPU but our approach is tested in a laboratory test-bed and by considering the time that packets spend to be received from network interface and to be processed on GPU then issue the verdicts. Another work is [8] that authors have used and employed parallel capability of GPU to process packets in user-space. It modifies user-level inter- face for packet I/O engine to remove high overhead caused by the I/O engine. In IP forwarding, only the destination IP address is checked but in packet filtering firewalls at least four to five fields have to be checked. Christiansen and Fleury [9] introduced
Interval Decision Diagrams for packet filtering using first order logic. They construct a logic formula by the set of rules based on the integer intervals that introduces the logarithmic search time but exponential build algorithm.

3 Linux Netfilter/IPTables

Netfilter refers to a framework within the Linux kernel that can be used to hook functions into the networking stack. On the other hand, Iptables uses the Netfilter to hook functions designed to perform operations on packets (such as filtering) into the networking stack [7]. You can think of Netfilter as a framework that Iptables builds firewall functionality on it. The kernel module named ip_tables which is a component of Netfilter provides table-based system for defining firewall rules that can filter or transform packets. The table can be administered using the Iptables user-space tool. System administrator can define tables containing chains of rules for the treatment of packets. Iptables searches chains of rules for matching with each packet sequentially. There are five predefined chains that a table may not contain all of them: PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING. Iptables has the following built-in tables: Filter, NAT, Mangle and Raw. The Filter table is used for filtering purposes and consists of three chains: INPUT, OUTPUT and FORWARD. Fig. 1 shows a sample Iptables rule set that is defined in the Filter table. This sample rule set consists of six rules that the top rule in each chain has the maximum priority and it will be processed first. Every rule in filter table has four main sections:

- Table, that the rule should be added to
- Chain, that the rule should be inserted to
- Match, the criteria
- Target, which specifies the action on matched packets

![Fig. 1. Excerpts from Iptables firewall configuration file, showing three chains and six rules](image)

In the Filter table of Iptables, each chain has a default action that when the packets did not match to any rule, it will be executed on packets. The default action is DROP or ACCEPT.
4 Overview on GPGPU

General-Purpose computing on Graphics Processing Unit (GPGPU) refers to the use of GPU processing capability for general purpose processing. The vast number of processing cores available in current generation of GPU architectures is well suited for parallel computing purposes. Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model created by NVIDIA Corporation and implemented by the GPUs that it produces. CUDA gives the developers the ability to use latest NVIDIA GPUs for computation like CPUs. Unlike CPUs, the GPUs have a parallel throughput architecture that create and execute many concurrent threads. CUDA programs consist of some phases that are executed on both CPU and GPU sides. The phases that have little or no parallelism are implemented on host code (on CPU) and the phases that have rich amount of parallelism are implemented on device code (on GPU). A simple code execution of CUDA program is illustrated in Fig. 2. CUDA-accelerated libraries, compiler directives, and extensions to industry-standard programming languages, including C, C++, are prepared for the CUDA platform accessible. Therefore, C/C++ programmers can use ‘CUDA C/C++’ that is compiled with ‘nvcc’ command to accelerate their applications on GPU [10].

![Fig. 2. Execution of a CUDA program](image)

In contrast to general-purpose processors that have to satisfy requirements from legacy operating systems, applications and I/O devices that make memory bandwidth more difficult to be increased, simpler memory models and fewer legacy constraints of GPUs have made it easier to achieve higher memory bandwidths. The more recent NVIDIA GT500 chip supports about 327.7 GB/s.

5 Implementation

5.1 IPTables/Libnetfilter_queue Packet Filtering

In addition to Iptables, Netfilter project also provides a set of libraries that can be used to perform different task in user-space [7]. One of these libraries is Libnetfilter_queue that is used for packet filtering in user-space. The target NFQUEUE in
Iptables rules is used to queue the packets to user-space. Receiving queued packets from kernel Nfnetlink_queue subsystem and then issuing verdicts to the kernel Nfnetlink_queue subsystem is performed by Libnetfilter_queue. For example, the following rule will ask for a decision to the listening user-space program for TCP packets going to the box:

```
iptables -A INPUT -p tcp -j NFQUEUE --queue-num 0
```

In user-space, our program uses the Libnetfilter_queue to connect to queue 0 and receive the messages from kernel. It should be noted that:

- A fixed length queue is implemented as a linked list of packets
- An integer value is integrated to each packet as an index
- The packets are released when the user-space program issues the verdict to the corresponding index value

We read multiple packets by the size of our program buffer, then the packet buffer is sent into the GPU to be processed in parallel in order to specify the action that should be applied on each packet by issuing the verdicts. Issuing the verdicts by Libnetfilter_queue for packets can be performed in any order, for example receiving packets in 1, 2, 3 order and issue them in 2, 3, 1 order. Packet flow in Netfilter is depicted in Fig. 3 and we can find the position of our packet-filtering program which is the abstract view of our test-bed. Moreover, It shows that how Iptables and Libnetfilter_queue cooperate to perform packet filtering in user-space.

![Packet Flow in Netfilter and the position of our CPU-GPU user-space program](image)

We create multiple threads in user-space on CPU to distribute traffic among them and then each thread sends its packet buffer to GPU. The synchronization between CPU threads is performed by a Boolean variable to avoid sending data packets on GPU at
the same time. In this situation multiple threads process different connections. Fig. 4 shows the multi threads program model. To distribute data packets to different CPU threads we have to define a rule with –queue-balance target which sends different connection to different queues to exploit processing power of all the cores of our CPU.

Fig. 4. Exploit multiple cores and threads of CPU

5.2 GPU Programming Model

Considering independent nature of each packet in network communications, it can be concluded that parallel processing of multiple packets can benefit from high parallel processing capability of GPU by using CUDA parallel computing platform. Therefore, we receive the packets and queue them in our packet array data structure. Acceleration of packet processing is performed by calling ‘buffer size’ multiply ‘rule set size’ number of threads in the GPU. Using this configuration, multiple packets about buffer size are processed once.

Every thread has two features: thread index and block index, thread index and block index can be three-dimensional in which the dimensions are x, y and z. In our implementation, we have used the x dimension of block index to specify each packet and combination of the y dimension of block index and x dimension of thread index to specify each rule. In this implementation, each thread checks a packet with a rule for matching. We have a buffer of packets and another array (rule_matched array) corresponding to this buffer array along with rule array. The rule array is sent in the beginning of our program into the GPU memory. The rule_matched array has the same size of packet buffer and each element of this array corresponds to the each packet in the packet buffer that specifies the first rule packet matched with. Each thread when finds a matching rule with a packet, writes its thread number to the corresponding element in the rule_matched array. Because the packets may matched more than one rule in firewall rule set and threads run concurrently, we have to apply a mechanism to solve this problem. For this reason, we have used CUDA ‘atomic-Min’ Atomic Operations to solve the problem. While a thread wants to write to that memory location, it locks the memory location and writes its rule number. Therefore, the first rule that has the maximum priority in firewall rule matching concept is written to the rule_matched array. Fig. 5 shows parallel model that is used to rule matching on GPU.
Fig. 5. Processing packets on GPU and return the verdict result to Host

The following equations show the relation between packets and rules with thread and block indexes:

\[ \text{Rule\_number} = \text{ThreadIndex.x} + \text{BlockIndex.y} \times \text{BlockDim.x} \quad (1) \]
\[ \text{Packet\_number} = \text{blockIndex.x} \quad (2) \]

Every thread that runs the code to find a matching rule with the packets uses these two values to point the buffer array and rule array.

5.3 Test-bed

We ran our user-space packet-filtering program on a commodity PC that has a 2.5 GHz Intel Core i5 CPU with a 1 Gbps Ethernet interface. The packet generator is a PC with a single 1 Gbps Ethernet interface which is named by Ubuntu1 and the firewall is Ubuntu2 PC which is shown in Fig. 6. Both PCs ran Ubuntu Linux 11.10. The two PCs are connected together by a crossover Ethernet cable according to Fig. 6. We have configured our test-bed to generate random source and destination IP addresses, source and destination ports by flooding the packets. All the packets that traffic generator generates are 80 bytes TCP packets with no TCP flags set. On the firewall PC that the program were running, we have configured the program to finish when it filters a specific amount of packets and then it prints the time elapsed. When we received the packets, and sent them into the GPU for processing then we simply set the verdicts to drop all the packets. We used Ostinato open source package in Ubuntu, which is a cross-platform network packet crafter/traffic generator to generate traffic. By this tool (Ostinato) we can create and customize different traffics and run all the traffics at the same time to any interface.
6 Results

We ran our test-bed for different rule set size of 100, 1000, 5000 and 10,000 rules. The rule sets are real rule set and the buffer size is 1000 packets. We have calculated the results in PPS and packet delay which is the time between receiving packets from network interface and process them on GPU then issue the verdicts. As it is mentioned before, the buffer size is 1000 packets while the rule set size is variable, the result throughput and packet delay are shown in Fig. 7. The x axis shows number of rules in the filter set files and the y axis shows the throughput and packet delay of the Iptables code and CPU-GPU versions code in user-space. For each state, we ran the program more than 30 times and the average value of them is calculated. The results are shown in Fig. 7 that we have at least 2.3 times throughput speedup and we get about 43 times speedup for 10,000 rules set size in the GPU over the CPU version. It is obvious that the throughput for 10,000 rules in sequential code is about 9000 PPS while the throughput for our GPU accelerated code is about 400,000 PPS for the same rule set size.

![Fig. 7. Throughput and packet delay for different rule set size on CPU and GPU](image)

The throughput of Libnetfilter_queue API that includes receiving the packets and issuing the verdicts for 80 Bytes packet sizes can reach up to 600,000 PPS without considering other processing overheads. Therefore, the throughput of CPU-GPU packet-processing code is limited up to 600,000 PPS. The program gains 400,000 PPS...
throughput on CPU-GPU for 10,000 rules that is anyway 43 times faster than CPU code of Iptables in kernel.

7 Conclusion and Future Works

In our work while the Iptables preserves the statefull fast packet matching we have improved the naive slow linear matching for large rule set and it is well suited for heavy loaded networks. Because the Libnetfilter_queue library uses Netlink sockets for inter-process communications (IPC) between the kernel and user-space therefore Netlink sockets are not suitable for high performance packet processing. Therefore, our user-space packet filtering throughput is limited up to 600,000 PPS. Consequently, the total bandwidth for 80 Bytes packet size is 256 Mbps while the corresponding Iptables linear search algorithm can only reach up to 5.8 Mbps bandwidth. The integration of high performance packet I/O with accelerated GPU packet processing can be the future work for removing packet I/O inefficiency. Employing a better packet-matching algorithm on GPU can make more parallelism acceleration to eliminate CPU processing power inefficiency for rich features and free open-source packet filter firewalls like Iptables.

References