International Journal of Control and Automation
Vol.6, No.5 (2013), pp.57-66
http://dx.doi.org/10.14257/ijca.2013.6.5.06

Genetic Algorithm Optimized Packet Filtering

Okta Nurika[1], Nordin Zakaria[2] and Low Tan Jung[3]

Department of Computer & Information Sciences
Universiti Teknologi PETRONAS,
Perak, Malaysia

[1]okta.rider@gmail.com, {[2]nordinzakaria, [3]lowtanjung}@petronas.com.my

Abstract

In this paper, we present a method to optimize packet filtering by genetic algorithm. Packet filtering in our work consists of packet capturing and firewall rules reordering. Genetic algorithm is used to automate rules reordering and the discovery of optimal combination of packet capture configuration, in the framework of PF_RING platform and rules ordering. Our method has been tested in different sizes of network traffic load. Genetic Algorithm evolves configuration based on the recorded throughput rates; the higher the throughput the better the solution. Results obtained indicate the effectiveness of the approach.

Keywords: *PF_RING; Packet Capture; Packet Filter; Genetic Algorithm; Optimization; Firewall*

1. Introduction

Packet capture is a method that enables users to capture and examine the contents of the data packets that go into a computer through network card [1]. Some purposes of packet capture includes troubleshooting network issues, examining network security, debugging protocol deployments, and studying the mechanism of network protocols [2].

PF_RING [3] is chosen as our platform of packet capture optimization and firewall rules reordering, which make up a packet filtering framework. PF_RING enables the development of network monitoring application and this can also contain firewall rule set. Applications that integrate PF_RING are for examples: Snort [4] and Suricata [5].

Our choice of PF_RING and genetic algorithm method are strengthened by previous works in Section II. Our methodology is explained in Section III. Section IV discusses about our experiments, results, and their analysis as well, where our optimization framework manages to optimize the overall packet filtering. Section V presents the conclusions of our work. Lastly, Section VI highlights the potential future works related to our research.

2. Related Works

2.1 Related Comparison Studies

We choose PF_RING as our packet filtering platform motivated by the results of numerous researches that show the reliability of PF_RING [3, 6, 7, 8, 9, 10]. Some works related to our packet filtering parameters are done by [7] who recommend very big buffer size for FreeBSD, and [11, 12, 13] who recommend device polling.

From the above works, we conclude that the configuration of packet capturing module has significant impact on the number of packets captured. This configuration setting should be

ISSN: 2005-4297 IJCA

adjusted according to the rate of incoming traffic. Different traffic load requires different treatment or setting.

2.2 Genetic Algorithm (GA)

GA is a search method to find near optimum solution, based on evolutionary natural selection process and genetics [14]. GA implements on population or group of chromosomes of potential solutions. GA shall generate the best combination of solution for specific problem. For every set of generation, it will create a new set of estimations by selecting chromosomes depending on their level of fitness, and then we can breed these chromosomes using genetic operators. This evolution is consequently expected to lead to better population.

The genetic operators are mentioned below according to description by [14, 15]:

1. **Reproduction**: It is an operator to make copies of better chromosomes in a new population.
2. **Crossover**: An operator to recombine two chromosomes to get a better breed.
3. **Mutation**: It adds new value to the chromosomes in the search process. It maintains the diversity of population and to dig deeper into the search space.
4. **Selection Method**: A method to assign more copies of solutions with better fitness values, in order to give them higher probability to survive.

2.3 Previous GA Implementations in Computer Networks

GA implementations in the area of computer networks have been done with positive results. This area includes packet filtering [15, 18], firewall rules accuracy [16], and packet classification [17]. Specifically, the need for GA in filter permutation optimization arises from the limitation of permutation optimization using adaptive pattern matching algorithm. This limitation occurs when the number of filters grows large (above 20) as stated by [19].

On the other hand, non-GA firewall rules optimization researches [20, 21, 22] use different methods to clean up firewall rules anomalies.

Our work notices an opportunity to optimize packet capture rate by combining firewall rules reordering with the optimization of kernel parameters. This will be discussed in the next section.

3. Proposed GA Optimized Packet Filtering Framework

3.1 Implementation

Our work fills up a research gap; GA has not been utilized to optimize both PF_RING kernel parameters and its firewall rules reordering, to our knowledge. This methodology is expected to boost the packet capture rate, compared to a system with firewall rules optimization but no kernel parameters optimized.

It is recommended to initiate GA implementation using existing GA library to save implementation time [14]. GAlib [23] is a recommendation by [14]. In our work, we utilize a minimized version of GAlib called GAlite [24].

Our optimization case is based on PF_RING parameters. PF_RING is a Linux kernel patch that accelerates packet capturing by creating a ring buffer inside the kernel to eliminate the need of copying packets from kernel to application space memory [25]. Therefore, an

application can directly access this PF_RING buffer to process the packets without copying those packets to its memory space.

Each combination of PF_RING parameters is a chromosome/individual. For each run, our GAlite application will generate values for these parameters, which then become the inputs to our PF_RING application. Our PF_RING application will run for 7 seconds, and then it generates a fitness value based on the total number of packets captured (throughput rate). This process repeats according to our GAlite application's algorithm and properties (number of generations, population size, *etc*.).

Finally, the best chromosomes from each generation will be selected, which are then compared to each other to select the overall best chromosome. The selected best chromosome is the near optimum combination of PF_RING configuration and its firewall rules ordering. The above mentioned process of our methodology is illustrated in the Figure 1 below:

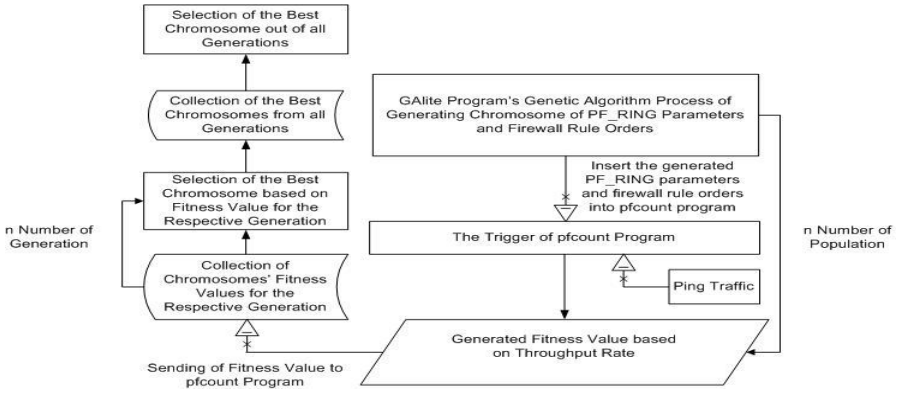

Figure 1. GA Optimized Packet Filtering Framework

3.2 Parameters to be optimized by GA

The permutations on PF_RING parameters which we have chosen, based on the official PF_RING manual by [26] are coded in 'pfcount'. These parameters are listed below:

1. **caplen or snaplen** = the size of packet capture length. The range is between 64 bytes to the size of Maximum Transmission Unit (MTU) of the layer 2 protocol. In our research, the protocol is Ethernet with its MTU is 1500 bytes. But in our algorithm, we make the maximum MTU to be 1536 to accommodate the increase of MTU every 64 bytes, and also simplify the rounding of result.
2. **active_wait** = ingress packet wait mode whether passive (poll) or active wait. When active wait takes place, every packet will activate kernel interrupt. While in passive wait, packets will be polled based on timer before kernel interrupt is called to process the packets. Integer 0 represents active wait and 1 represents passive mode as coded in PF_RING application.
3. **watermark** = packet poll watermark whether to make it low (reduce latency of poll but increase the number of poll calls) or make it high (increase the packet latency but reduce the number of poll calls). The range is 1 byte (check everytime for packet) until 50% of the maximum ring buffer size. In our research, the maximum ring buffer size is 4096, thus the maximum watermark value is 2048.
4. **cluster_type** = cluster type whether per-flow or round-robin. In per-flow cluster type,

each part of a packet transmission flow will go to the same cluster of buffer. While in round-robin cluster type, the chops of a packet transmission flow might go to the different cluster of buffer.

5. **poll_duration** = poll timeout in msec that takes care of data structures synchronization. According to [27], an acceptable clock interrupt delay for real time applications is around 100 ms to 200 ms. Additionally, [28] explains that the Linux system timer is programmed to generate a hardware interrupt 100 times in every second, which means that each hardware interrupt will take place every 10 ms. This concept has been affirmed by [29] that informs that in the Linux kernel, system timer ticks every 10 ms. Therefore, we conclude that when passive wait is chosen, the appropriate range of poll duration is 10 ms – 200 ms with escalation every 10 ms.
6. **Firewall rule set** = a set of firewall rules that in our case consists of 13 rules. Each rule will be given randomly generated priority value. This firewall rules ordering mechanism applies in condition where there are more rules to accept packets, than the rules to drop them, since the only fitness value used is throughput rate. It is also a good method to reorder the rules, if the network administrator is faced with loads of existing firewall rules. The rule index in the chromosome starts from 5 to 18.

3.3 PF_RING Application's Representation of Chromosome

Every chosen parameter is part of a chromosome which is indexed starting from zero (0). In other words, each parameter is a gene. In our GA implementation, every gene contains a float range value from 0-1 that will be randomly generated. Thus, specific formula for each parameter must be created to accommodate their real values from minimum to maximum. Below are the formulas based on C programming syntax, where gene(x) is the float random number generator ranging from 0-1:

1. **snaplen = (**gene(0) x 23) x 64 + 64
2. **active_wait =** (gene(1) > 0.5) ? 1 : 0
3. **watermark =** (gene(2) x 89) x 23 + 1
4. **cluster_type** = (gene(3) x > 0.5) ? "cluster_round_robin" : "cluster_per_flow"
5. **poll_duration** = (gene(4) x 19) x 10 + 10
6. **firewall rule [ ] priority =** gene(5)

The pseudo code of 'pfcount' below represents the mechanism of PF_RING itself.

```
Read snaplen input;
Read wait mode input {
        If (0)
                Active wait (Read poll duration = 0);
        If (1)
                Passive wait (Read poll duration from input; }
Read watermark input;
Read poll duration input {
        If (Active wait)
            Poll duration = 0;
        Else
            Poll duration = input;
Read cluster type input;
Read firewall rules order input;
PF_RING Socket Initialization;
```



```
While (true) {
          If (watermark = watermark input)
              Read incoming packets;
              Process incoming packets;
              Filter Incoming Packets;
              Count received packets;
              Generate throughput value;
          Else (poll duration times out)
              Read incoming packets;
              Process incoming packets;
              Filter Incoming Packets using Rules;
              Count received packets;
              Generate throughput value; }
```

The words in bold in the above pseudo code indicate PF_RING parameters to be optimized.

4. Experiment Results and Analysis

The experiment was conducted on a Virtual Box 4.1.6 with CPU: AMD Athlon Neo X2 Dual Core L335 1.60GHz (1 virtual CPU mode), AMD-V Hardware Virtualization with nested paging enabled, 512MB of RAM, Realtek PCIe GBE 1Gbps , Linux Kernel version 2.6.32, PF_RING 5.4.5, under Linux Debian 6.0.4.

Our packet capturing framework was measured against twenty-four levels of ping traffic load, according to the size of each packet injected (increase every 64 bytes), which are 64 up to 1536 bytes. These are considered low, medium, to high network traffic load.

The initial GA properties itself generates 50 of population size, 50 generations, 0.2 of mutation probability, 0.4 of crossover probability, and tournament selection method. In tournament selection, two chromosomes are picked up randomly, then their fitness values will be compared, and after that the winner will be chosen to be the next individual. This process repeats until the specified number of population is achieved [14].

The best chromosomes were determined from every traffic load. Below are the table and graphs showing the throughput progression for each load, and patterns of the selected PF_RING parameters.

Table 1. The Chosen Best Chromosome Contents for Different Traffic Loads

No.	Packet Size (bytes)	Best Chromosome Throughput (bytes/7 sec)	Snaplen (bytes)	Active/ Passive Wait	Watermark Value (bytes)	Cluster Type	Poll Duration (ms)	Firewall Rules Order
1.	64	4435.92	192	Active wait	1703	Per flow	10	5-1-8-9-13-3-7-10-12-2-4-11-6
2.	128	6457.92	384	Active wait	2920	Per flow	350	11-8-7-3-12-2-6-13-1-9-5-4-10
3.	192	8640	1664	Active wait	990	Round robin	210	12-9-6-2-3-11-5-4-8-1-10-13-7
4.	256	10353.2	192	Active wait	1864	Per flow	90	12-10-1-5-3-13-11-4-2-9-7-8-6
5.	320	11975	512	Passive wait	2644	Per flow	210	5-8-7-3-9-4-1-6-2-13-12-10-11
6.	384	13465	1728	Active wait	1864	Per flow	170	13-1-7-3-8-12-2-6-5-9-4-10-11
7.	448	15139.5	384	Active wait	942	Per flow	100	3-1-12-10-7-13-2-6-9-11-8-4-5
8.	512	17259.4	1088	Active wait	114	Per flow	150	7-1-9-2-3-13-5-10-4-12-6-11-8
9.	576	19172.2	2304	Active wait	183	Per flow	30	4-6-1-10-13-9-2-8-5-3-12-11-7
10.	640	21226.7	2048	Active wait	714	Per flow	100	11-2-4-5-9-12-7-6-8-13-10-3-1
11.	704	22433.3	3840	Active	574	Round	90	10-6-13-9-5-4-

				wait		robin		7-11-12-8-1-3-2
12.	768	24713.9	640	Active wait	369	Per flow	110	12-10-11-13-5-9-7-4-6-2-1-8-3
13.	832	26786.2	2432	Active wait	1588	Round robin	230	6-12-11-1-4-10-8-5-2-7-13-3-9
14.	896	28549.3	2240	Active wait	921	Per flow	280	2-12-3-8-13-11-10-6-7-5-1-4-9
15.	960	31112	1088	Active wait	760	Per flow	110	3-1-10-6-9-2-11-12-13-4-8-5-7
16.	1024	33081	192	Passive wait	1816	Per flow	130	12-6-7-3-13-9-1-11-5-4-8-10-2
17.	1088	34655	1920	Active wait	1519	Per flow	150	2-9-1-13-11-5-12-3-4-7-10-6-8
18.	1152	36421	704	Passive wait	3426	Per flow	70	3-1-13-6-11-12-2-9-8-7-5-4-10
19.	1216	40131	2560	Active wait	829	Per flow	40	2-4-6-12-9-3-11-10-7-8-1-5-13
20.	1280	40461.1	2432	Passive wait	1103	Per flow	120	4-2-13-8-12-3-6-5-7-10-9-1-11
21.	1344	42172.2	576	Passive wait	3081	Per flow	10	9-10-13-11-4-2-3-5-6-8-1-7-12
22.	1408	46237.6	64	Passive wait	369	Round robin	90	6-8-10-9-1-4-11-12-5-7-13-3-2
23.	1472	47368.8	384	Active wait	1496	Per flow	200	9-8-6-7-11-10-5-2-12-3-1-4-13
24.	1536	50485.1	448	Active wait	1034	Round robin	80	11-12-10-6-3-9-8-2-7-4-5-1-13

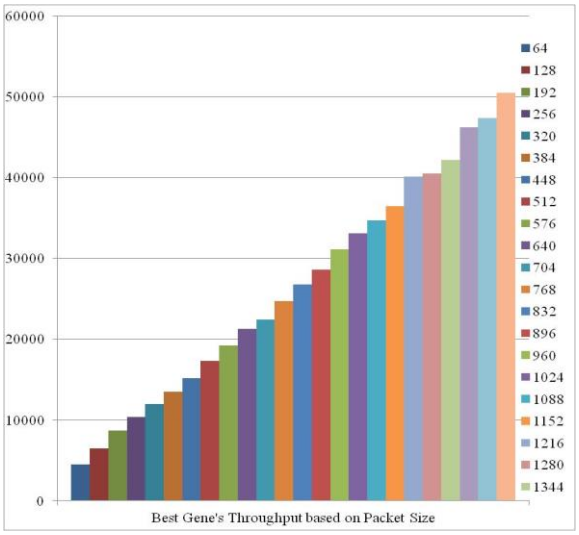

Figure 2. Throughput Progression

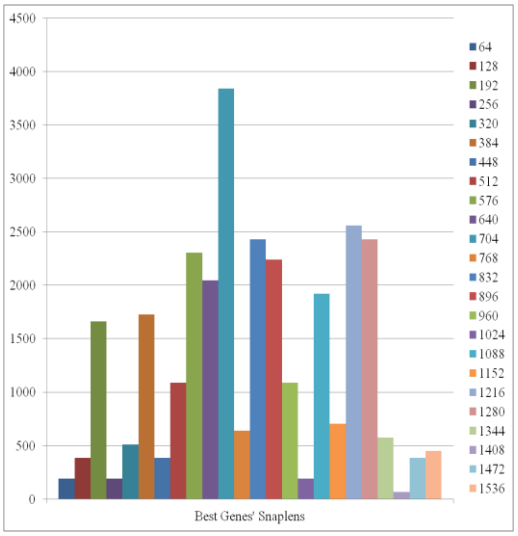

Figure 3. Snaplen Values

Figure 2 and Table 1 show the sequential throughput rate from ping traffic penetration, starting from 64 until 1536 bytes of packet size. The throughput rate progression is stable from 64 until 1152 bytes of packet size with moderately constant progression. But starting from 1216 until 1536 bytes of packet size, the throughput rate started to progress with more unstable varieties in progression. This shows that optimization is mostly needed within this range. The former stability affirms the research result by [3] that claims PF_RING performs consistent at low packet sizes, despite if we look at our Figure 3, 4, and 5 below; the snaplen size, watermark value, and poll duration for each packet size have high significant difference. However, they do not block the consistency of PF_RING.

Referring to Figure 3 and Table 1, it is viewed that 62.5% of all traffic loads or 15 out of 24 have snaplen sizes, which are bigger than their packet size. From here, we can conclude that that there is relation between the size of snaplen and the packet capture rate. The snaplen size must be bigger than the size of each packet of the input traffic, which is logical. This practice will prevent segmentation that takes CPU cycle, which eventually makes the network analysis process even longer, since the fragmented segments have to be reassembled at the receiving application.

The most significant improvements are seen at ping traffics with 768, 960, 1216, 1408, and 1536 bytes of packet size. These particular significant improvements of PF_RING have strengthened the claim by [3] and [7], who noticed that capturing a 1GE stream with big sized packet was much easier than capturing it with small sized packet (64 bytes).

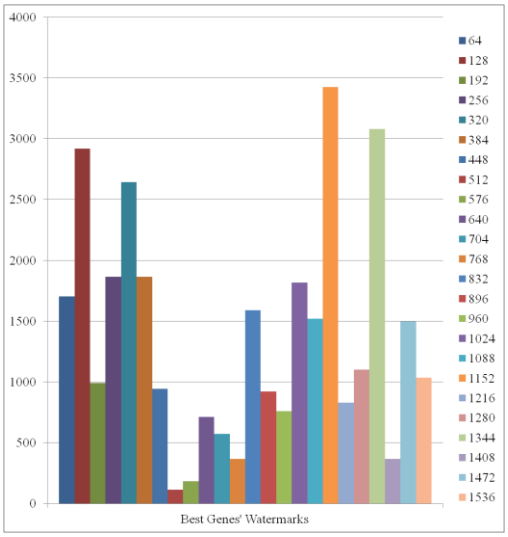

Figure 4. Watermark Values

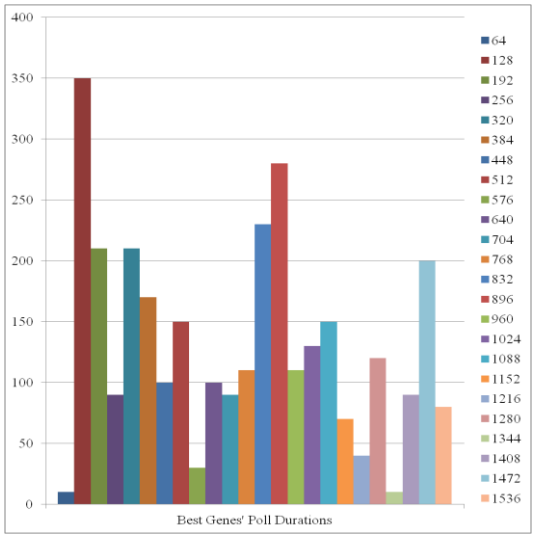

Figure 5. Poll Duration Values

Figure 4 shows the watermark values for all traffic loads. The highest watermark value is 3426 bytes and the lowest is 114 bytes. The 3426 value actually exceeds the specified limit that is 2048 bytes, which was caused by the 7 significant digits of C++ float data type. Analysis from this figure generates 17 out of 24 or 70.83% of all traffic loads have watermark values, which are less than half of the highest recorded watermark value. This reflects that the recommended general watermark value is at most half of the specified maximum limit, although PF_RING will still perform well if it exceeds. The optimal value of watermark will make the network application not to wait too long before receiving the packets.

Figure 5 displays the poll duration for every traffic load. We make the analysis based on three categories. First is for traffic loads that have poll duration equal or below half of highest specified duration (200 ms), second is for traffic loads that exceed half of highest specified duration (200 ms), and the last one is for traffic loads that equal or exceed highest specified duration (200 ms). The resulted categorization is listed in table below.

Table 2. Resulted Poll Duration Categories

Poll Duration Category	Number of Traffic Load
Duration < = 100ms	11
100ms < Duration < 200ms	7
Duration > = 200ms	6

Referring to Table 2, there are eleven traffic loads (45.83%) in the first category, which are 64, 256, 448, 576, 640, 704, 1152, 1216, 1344, 1408, and 1536 bytes. The second category has seven traffic loads (29.17%), each 384, 512, 768, 960, 1024, 1088, and 1280 bytes. Finally the third category includes 6 traffic loads (25%), namely 128, 192, 320, 832, 896, and 1472. Within the all categories, the traffic loads vary from low/small, moderate/medium, up to high. Additionally, the first category dominates with the most number of traffic loads. From this analysis, we interpret that it is generally suggested to deploy poll duration that is ranging from 10 until 100 ms, for the mentioned traffic load range (64-1536 bytes).

Table 3. Portion of Wait Mode Utilization

Wait Mode	No. of Traffic Loads
Active Wait	18
Passive Wait	6

Table 4. Cluster Mode Utilization

Cluster Mode	No. of Traffic Loads
Per Flow	19
Round Robin	5

Looking at Table 3, we can state that 75% or 18 out of 24 traffic loads were treated with Active Wait (no device polling). Based on Table 1, for traffic loads of 64-512 bytes, there is only one traffic load (320 bytes) or 12.5% of this range that uses Passive Wait (device polling), it means that within this range, the use of Active Wait (kernel interrupt for every packet) costs cheaper than device polling. Next, for traffic loads of 576-1024 bytes, there is also only one traffic load (1024 bytes) or 12.5% of this range using Passive Wait, which says that within range of traffic load, Active Wait deliberately performs better. And at last is the analysis of Wait Mode for traffic load ranging from 1088 to 1536 bytes of packet size, where there are four traffic loads (1152, 1280, 1344, and 1408 bytes) or 50% of this range that activate Passive Wait. It concludes that for this higher load of traffic range, the utilization of device polling delivers high impact.

Table 4 illustrates the Cluster Mode utilized by all traffic loads. There are 19 traffic loads (79.17%) utilizing Per Flow cluster, while the rest (5 traffic loads) or (20.83%) use Round Robin cluster. Table 1 tells that Round Robin cluster only applies on traffic load 192, 704, 832, 1408, and 1536 bytes. On the other side, Per Flow cluster applies on majority of traffic loads ranging from small, medium, until high traffic loads. We can make an explanation that Per Flow cluster gives better packet capturing rate in general, compared to Round Robin mode. This is suitable when only one network application takes place, where there is no sharing of packets among multiple applications, therefore the single application will only look on its own buffer, and it will get all parts of a transmission flow without the need to reassemble if parts are separated.

For further analysis, Table 1 displays the contents of the selected best chromosome (PF_RING configuration and its firewall rules ordering) for 24 different traffic loads (64-1536 bytes). There are 13 rules excluding the default ‘deny’ rule at the end of the set, which is not included in reordering process. Analysis from the firewall rules ordering column of Table 1, we notice that 14 out of 24 chromosomes (58.33%) put rule no. 13, which has ‘accept’ action at the front line of the rule set. The chromosomes with this characteristic are for traffic load 64, 256, 384, 448, 512, 576, 704, 768, 896, 1024, 1088, 1152, 1280, and 1344 bytes of each packet size. The traffic loads with this anomaly range from small up to high traffic loads. Thus, we can conclude that our GA approach works well to order or reorder the firewall rules, where the rule set tends to have more rules to accept packets than to drop them, especially when the new administrator is facing existing firewall rules, where new insertions and reordering are highly needed.

At last, we compare the packet capture rates between our method that adds kernel parameters optimization to the firewall rules, against the mere firewall rules without kernel parameters optimization. The results are described in the table below.

Table 5. Packet Filtering Rates Comparison between Two Methods

Traffic Load	Packet Filtering Rate of System with Mere Firewall Rules (bytes/7 sec)	Packet Filtering Rate of System with Firewall Rules added with Kernel Parameters Optimization (bytes/7 sec)	Increment of Packet Filtering Rate (%)
64	3764.76	4435.92	17.83
128	5701.08	6457.92	13.27
192	7700	8640	12.21
256	9533.64	10353.2	8.60
320	11650	11975	2.79
384	13374.9	13465	0.67
448	15075.1	15139.5	0.43
512	17248.6	17259.4	0.06
576	19118.7	19172.2	0.28
640	21149.4	21226.7	0.37
704	22386.5	22433.3	0.21
768	24587.4	24713.9	0.51
832	26628.8	26786.2	0.59
896	28410.1	28549.3	0.49
960	30804.7	31112	1
1024	32936.1	33081	0.44
1088	34586.3	34655	0.2
1152	35417.8	36421	2.83
1216	39791.2	40131	0.85
1280	40323.1	40461.1	0.34
1344	42116.1	42172.2	0.13
1408	44419.8	46237.6	4.09
1472	46481.4	47368.8	1.91
1536	50055.6	50485.1	0.86

From Table 5, we can refer that the packet rates of the system with kernel parameters optimization are always improved, compared to a system with mere firewall rules without kernel parameters optimization. Overall, our methodology is able to improve the packet capture rates within traffic loads, which range from 64 until 1536 bytes as tested.

5. Conclusions

The packet capture kernel parameters optimization and firewall rules reordering, can be accomplished at the same time in one GA optimization framework. Our GAlite program performs the combinatorial process based on the PF_RING kernel parameters and its firewall rules. The results show that the proposed method is suitable for different levels of traffic load, since different traffic load requires particular packet capture setting of kernel parameters. However, the firewall rules ordering does not depend on the size of traffic load. Proper rules ordering maximizes the throughput rate regardless of the size of traffic load. Our framework can assist an administrator to set optimal packet capture configuration when specific network transmission is desired, especially when the traffic input rate can be estimated.

6. Future Work

In the future, our framework can be applied in other prospective open source packet capturing module. Thus, it can become a generic packet capture optimization method. Energy measurement can also be added to develop a green packet capturing framework based on the power consumption for specific configuration of packet capture parameters.

Acknowledgements

This work has been done using the facilities at High Performance Computing (HPC) Service Centre of Universiti Teknologi PETRONAS (UTP).

References

[1] SonicWALL, SonicOS Enhanced 4.0: Packet Capture, **(2012)**.
[2] Wireshark, "Packet Capture Purposes", http://www.wireshark.org/docs/wsug_html_chunked/ ChapterIntroduction.html#ChIntroPurposes, **(2012)**.
[3] L. Deri, "Improving Passive Packet Capture: Beyond Device Polling", Proceedings of the 2004 4th International System Administration and Network Engineering Conference, **(2004)**, pp. 1-10.
[4] L. Ricciulli and T. Covel, "Inline Snort multiprocessing with PF_RING", Snort Setup Guides, **(2011)**.
[5] Suricata, Suricata with PF_RING, https://redmine.openinfosecfoundation.org/projects/suricata/wiki/ Installation_with_PF_RING, **(2012)**.
[6] L. Braun, A. Didebulidze, N. Kammenhuber and G. Carle, "Comparing and Improving Current Packet Capturing Solutions based on Commodity Hardware", Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, **(2010)**, pp. 206-217.
[7] F. Schneider, J. Wallerich and A. Feldmann, "Packet capture in 10-gigabit ethernet environments using contemporary commodity hardware", Proceedings of the 8th International Conference on Passive and Active Network Measurement, **(2007)**.
[8] M. Dashtbozorgi and M. A. Azgomi, "A high-performance and scalable multi-core aware software solution for network monitoring", The Journal of Supercomputing, vol. 59, no. 2, **(2010)**, pp. 720-743.
[9] L. Deri and F. Fusco, "Exploiting commodity multi-core systems for network traffic analysis", **(2009)**.
[10] G. A. Cascallana and E. M. Lizarrondo, "Collecting packet traces at high speed", Proceedings of Workshop on Monitoring, Attack Detection and Mitigation (MonAM) 2006, **(2006)**.
[11] J. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel", ACM Transactions on Computer Systems, vol. 15, no. 3, **(1997)**, pp. pp. 217–252.
[12] I. Kim, J. Moon and H. Y. Yeom, "Timer-based interrupt mitigation for high performance packet processing", Proceedings of 5th Int'l Conference on High-Performance Computing in the Asia-Pacific Region, **(2001)**.
[13] L. Rizzo, "Device Polling Support for FreeBSD", Proceedings of BSDCon Europe Conference, **(2001)**.
[14] K. Sastry, D. Goldberg and G. Kendall, in Search Methodologies, Edited E. K. Burke and G. Kendall, Springer US, Los Angeles **(2005)**, pp. 97-125.
[15] A. Shrivastava and S. Hardikar, "Performance Evaluation of BPNN and Genetic Algorithm", VSRD International Journal of CS & IT, vol. 2, no. 7, **(2012)**, pp. 621-628.
[16] E. El-Alfy, "A Heuristic Approach for Firewall Policy Optimization", The 9th International Conference on Advanced Communication Technology, vol. 3, **(2007)**, pp. 1782 – 1787.
[17] A. T. Nottingham and B. Irwin, "gPF: A GPU Accelerated Packet Classification Tool", Southern Africa Telecommunication Networks and Applications Conference, **(2009)**.
[18] A. Nottingham and B. Irwin, "Investigating the Effect of Genetic Algorithms on Filter Optimisation within Fast Packet Classifiers", Proceedings of the ISSA 2009 Conference, **(2009)**, pp. 99-116.
[19] A. S. Tongaonkar, "Fast Pattern-Matching Techniques for Packet Filtering", Stony Brook Unive. Tech., **(2004)**.
[20] J. G. Alfaro, N. Boulahia-Cuppens and F. Cuppens, "Complete analysis of configuration rules to guarantee reliable network security policies", Int'l Journal of Information Security, vol. 7, no. 2, **(2007)**, pp. 103-122.
[21] E. S. Al-Shaer and H. H. Hamed, "Discovery of policy anomalies in distributed firewalls", Proceedings of the 23rd Annual Joint Conf. of the IEEE Comp. and Communications Societies, vol. 4, **(2004)**, pp. 2605-2616.
[22] E. S. Al-Shaer and H. H. Hamed, "Design and Implementation of Firewall Policy Advisor Tools", Technical Report CTI-techrep0801, **(2002)**, pp. 1-21.
[23] M. Wall, "GAlib: A C++ Library of Genetic Algorithm Components", Massachusetts Inst. of Tech., **(1997)**.
[24] N. Zakaria, "GAlite: A Small C++ GA Library with Just the Minimal Functionality", http://code.google.com/p/galite/, **(2012)**.
[25] NTOP, "Why TNAPI (Threaded NAPI)?", http://www.ntop.org/products/pf_ring/tnapi/, **(2012)**.
[26] NTOP, PF_RING User Guide, Linux High Speed Packet Capture, **(2012)**.
[27] C. Dovrolis, B. Thayer and P. Ramanathan, "HIP: Hybrid Interrupt-Polling for the Network Interface", Proceedings of 2001 ACM SIGOPS Operating Systems Review, vol. 35, no. 4, **(2001)**, pp. 50-60.
[28] J. Mohr, "Linux Hardware Interrupt", http://www.linux-tutorial.info/modules.php?name=MContent &pageid=86, **(2012)**.
[29] K. Wehrle, F. Pahlke, H. Ritter, D. Muller and M. Bechler, in The Linux® Networking Architecture: Design and Implementation of Network Protocols in the Linux Kernel, Pearson Prentice Hall, New Jersey, **(2005)**, pp. 35-39.

