© Copyright Notice

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law.

Effectiveness of Intrusion Detection Systems in High-speed Networks

Qinwen Hu
Department of Computer Science
The University of Auckland
New Zealand
qhu009@aucklanduni.ac.nz
Muhammad Rizwan Asghar
Department of Computer Science
The University of Auckland
New Zealand
r.asghar@auckland.ac.nz
Nevil Brownlee
Department of Computer Science
The University of Auckland
New Zealand
n.brownlee@auckland.ac.nz

Abstract: Network Intrusion Detection Systems (NIDSs) play a crucial role in detecting malicious activities within networks. Basically, a NIDS monitors network flows and compares them with a set of pre-defined suspicious patterns. To be effective, different intrusion detection algorithms and packet capturing methods have been implemented. With rapidly increasing network speeds, NIDSs face a challenging problem of monitoring large and diverse traffic volumes; in particular, high packet drop rates can have a significant impact on detection accuracy. In this work, we investigate three popular open-source NIDSs: Snort, Suricata, and Bro along with their comparative performance benchmarks. We investigate key factors (including system resource usage, packet processing speed and packet drop rate) that limit the applicability of NIDSs to large-scale networks. Moreover, we also analyse and compare the performance of NIDSs when configurations and traffic volumes are changed.

Introduction

Because of advances in Internet technologies, computer networks and ICT solutions bring convenience and efficiency in our daily life. The Internet's impact on society, economy and industry are rising rapidly. Despite significant benefits, these networks have raised numerous security issues. A Symantec[i] report, published in 2016, indicates that the number of zero-day vulnerabilities (Liu, Burmester, Wilder, Redwood, & Butler, 2014) increased 125% within a year (Symantec, 2016). At least 100 million fake technical support scams have been blocked by Symantec. An average of one million web attacks was blocked each day in 2015, an increase of 117% (more than double) compared with 2014 (Symantec, 2016). Therefore, it is important to find an effective way to protect our networks. There are off-the-shelf solutions in the market, such as firewalls

(Khoumsi, Krombi, & Erradi, 2014) that can filter inbound network traffic, antivirus software (Shukla, Singh, Shukla, & Tripathi, 2014) used to stop worms, and VPNs (Wright, 2000) that encrypt dataflow between communicating parties over the Internet. However, these mechanisms have limitations in detecting attacks within the internal network, while other mechanisms cannot inspect the contents of data packets. It is important to put in a second line of defence. To this end, NIDS can help network administrators to detect unwanted traffic passing through a network by providing three essential security functions: monitoring, detection, and responding to unauthorised activity by insiders and outsiders. Basically, NIDS monitors the traffic or events occurring in a computer system or network, then scans packets, identifying patterns that are matched in a pre-defined detection mechanism, such as signatures, rules, or scripts. Each detection mechanism is based on a known vulnerability, threat, or pre-attack activity. If NIDS detects any threats, then it generates an alarm message that could be inspected by the network administrator.

Nowadays, with increases in link capacity and speeds, NIDS faces the computational challenge of handling higher traffic volumes and running complex per-packet rules. Especially, when the network traffic reaches its peak, NIDS starts dropping packets before safely delivering them to its intrusion detection engine. This impedes its detection accuracy and leads to missed malicious activities, i.e., a potential increase in false negatives.

Several studies have investigated NIDS performance under different network environments, hardware, and platforms. For instance, in (Salah & Kahtani, 2010), Salah et al. compared Snort performance between Linux and Windows 2003 Server through a 1 Gb/s testing network. They found that Snort performs better in the Linux environment. Similarly, Alhomouda et al. (Alhomoud, Munir, Disso, Awan, & Al-Dhelaan, 2011) used different platforms (including ESXi Virtual Server, Linux 2.6, and FreeBSD) to measure the performance of Snort and Suricata. Their results suggested using Suricata on FreeBSD for monitoring unauthorised activities in high-speed networks. In another study, Paulauskas and Skudutis (Paulauskas & Skudutis, 2008) investigated Snort performance using specified hardware, logging techniques, and pattern matching algorithms. Their results showed that better hardware types and logging techniques could improve Snort's performance.

By reviewing existing studies, we discovered that the NIDS's CPU usage, memory usage, and packet drop rates could be affected by different configurations, i.e., detection algorithm, Data AcQuisition (DAQ) methods, amount of detection rules and flow types. In this paper, we evaluate the performance of three NIDSs on a 10 Gb/s network. In our experimentation, we explored key architectural considerations for open-source NIDSs as the network speed increases. We observed that the default configuration was not able to handle the traffic volume in our 10 Gb/s test network. Moreover, as the traffic volume rises and the traffic diversity increases in our University of Auckland campus, the number of dropped packets increases significantly when we deploy NIDSs with their default configurations. We applied various optimisation techniques and tested different packet detection algorithms (Garcia-Teodoro, Diaz-Verdejo, Maciá-Fernández, & Vázquez, 2009) and DAQ methods. As a result, all the NIDSs showed a significant performance improvement.

In the rest of this paper, a brief overview of three popular NIDSs is given in Section II. Section III describes the testing environment, methodologies, and scenarios. In Section IV, the experimental results of using different configurations are discussed. Section V indicates the major impacts of using the default NIDS's configurations in high-speed networks. We also share our experiences of tuning NIDS configurations in high-speed networks for reducing packet loss rate and the system resource consumption. Section VI concludes this paper and provides research directions for future work.

Open-Source Intrusion Detection Tools

Many commercial and open-source NIDS tools are widely deployed in existing networks. Typically, organisations use one of them to protect their network. In this paper, we study three popular open-source NIDS tools: Snort[ii] , Bro[iii] , and Suricata[iv].

Like existing NIDSs, each of these open-source NIDSs provides three major functions. First, it captures the traffic passing into and out from a network. Second, it then rebuilds and decodes the traffic based on its flow types. Finally, it analyses packet contents based on pre-defined detection rules, which represent patterns from the past attacks.

However, each NIDS provider has implemented different detection algorithms and capture methods for achieving different design goals. For instance, Snort aims to provide a lightweight NIDS solution; therefore, it uses a signature-based detection with a single thread for minimising the hardware requirements and reducing the configuration complexity. In contrast, Suricata extends basic design ideas from Snort, but it introduces a multi-threaded structure to improve the detection performance. The following section briefly discusses the similarities and differences among three NIDSs.

Snort

Snort is a signature-based detection tool. It is an open source NIDS that offers both network intrusion detection and network intrusion mitigation. Basically, it comes with a set of relevant rules and features that help users to detect attacks and probes. For instance, based on the user's specification, Snort can detect buffer overflows, port scans, and web application attacks.

Bro

Bro is an open-source NIDS that passively monitors network traffic and looks for suspicious activities. Bro uses a specialised policy language that helps users to monitor and analyse attacks. For instance, Bro provides default policies for detecting SSH brute-forcing and validating Secure Sockets Layer (SSL) certificate chains. Unlike the current signature-based NIDSs, Bro provides more features for the detection of semantic misuse, anomaly detection, and for behaviour analysis.

Suricata

Suricata is another open-source NIDS tool. It shares some similarities with Snort and Bro. For instance, it inherits both pattern matching and script solutions to withstand attacks against it. It also introduces some new ideas and technologies to the NIDS solutions. That is, it offers a multi-threading solution to improve the flow processing speed.

Table I shows that all three intrusion detection systems have their merits. There is no ideal system. Instead, each system has its advantages and disadvantages.

Table I: AN OVERALL COMPARISON AMONG THREE NIDSS BASED ON THE DETECTION MECHANISM, PACKET CAPTURING METHOD, AND THE CONFIGURATION COMPLEXITY.

Parameter	**Snort**	**Suricata**	**Bro**
Packet detection	Pattern matching detection	Anomaly detection	Both
Packet capturing method	libpcap and AF Packet	libpcap and PF Ring (Bro-Clusters)	libpcap, AF Packet and PF Ring
Configuration complexity	Change the configuration file or the command line	Modify the code	Change the configuration file or the command

Our Methodology

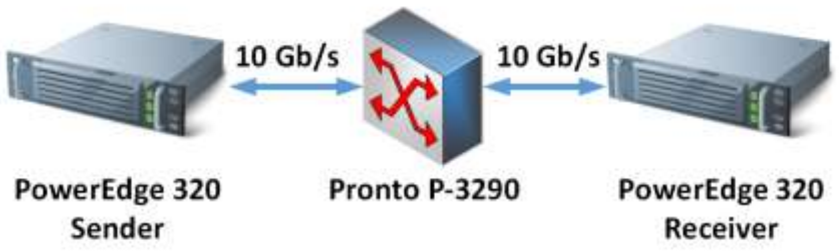

Fig. 1: Two test machines were configured for our 10 Gb/s experimental network A sender used GridFTP to deliver large quantities of data. Three IDSs were installed on the receiver side to measure NIDS performance while monitoring a 10 Gb/s data pipe.

In this section, the experimental environment and the proposed methodology are described. Test results will be discussed later. We evaluated the NIDS performance in our experimental and real networks. We used the former in order to generate a performance baseline for each NIDS. Fig. 1. shows the logical network diagram for this 10 Gb/s experimental testing environment. In this test network, the infrastructure consisted of two servers and one switch. More detailed information about the experimental environment can be found in Table II, which lists our hardware specifications.

Table II: AN OVERALL COMPARISON AMONG THREE NIDSS BASED ON THE DETECTION MECHANISM, PACKET CAPTURING METHOD, AND THE CONFIGURATION COMPLEXITY.

Role	**Model**	**CPU**	**Memory**	**NIC Data Transfer Rate**
Sender	Dell R320	Intel Xeon E5-2407 2.20GHz	16GB	Broadcom BCM57810 10 Gb/s
Receiver	Dell R320	Intel Xeon E5-2407 2.20GHz	16GB	Broadcom BCM57810 10 Gb/s
Switch	Pronto P-3290	MPC8541	2GB	Firebolt-3ASIC

We set up a port mirror in our campus boundary network, to monitor all the IPv4 and IPv6 traffic. Our campus network setup is shown in Fig. 2. As we can see, it has a 10 Gb/s Internet link, and our mirror host is a Dell R420 with 32 GB memory.

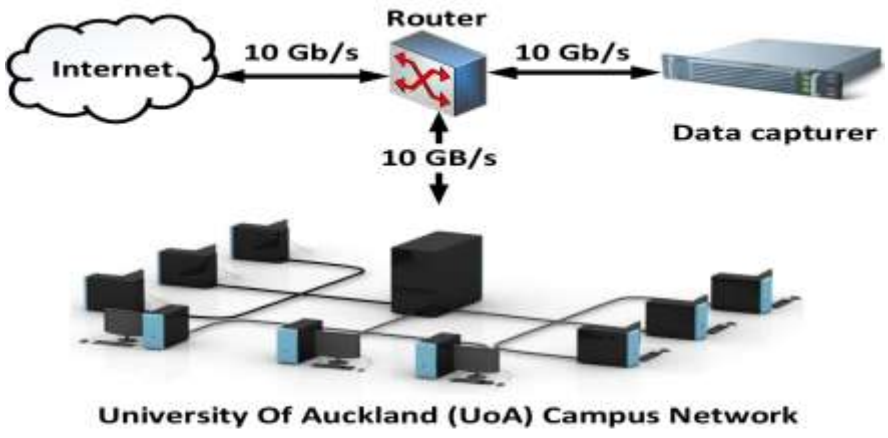

Fig. 2: Measuring CPU usage, memory usage and the packet drop rate on a Data Capture. Each test was run on a NIDS on the mirror host separately to monitor all incoming and outgoing traffic from the UoA campus network.

Methodologies

We have used two different methodologies. In our experimental environment, we used the active measurement strategy; we aim to deliver a large volume of data quickly and efficiently through high capacity links (10 Gb/s). However, the standard TCP or UDP implementation cannot be used for these links. Yu et al. (Yu, Brownlee, & Mahanti, 2014) mention some modified implementations that allow GridFTP, Fast Data Transfer (FDT), and UDP-based Data Transfer (UDT) to deliver a high volume of data through high-capacity links, and do a better job than the generic implementations do. The results of their experiment indicate that UDT had some utilisation problems when delivering large quantities of data through a 10 Gb/s network. The performance results for GridFTP and FDT were similar to each other, but when they used GridFTP with jumbo frames, it showed better performance for transferring high volumes of data in both congested and uncongested links. Based on the FTP protocol, GridFTP uses parallel data transfer, and improves the performance of high-speed data transfers between hosts over a long fat pipe network.

In our experimental environment, we did not launch any background traffic. Instead, the Sender (192.168.6.2) uses GridFTP to transfer large data volumes at 10 Gb/s to the Receiver (192.168.6.1). Yu et al.'s experiments demonstrated that GridFTP was able to handle about 20 flows running in parallel, but the number of dropped packets increased significantly beyond this limit. Therefore, in our experiments, we have evaluated a single flow test case, as well as the 20 flows scenario. In contrast, we used the passive measurement strategy to evaluate the NIDS performance in the UoA campus network link. There are two reasons for choosing the UoA network: we have a full view of incoming and outgoing packets, and the UoA network link provides a typical mix of flow types.

Experiment Scenarios

After analysing the NIDS design goals and architectures, we noticed that the NIDS performance could be affected by the following configurations: packet detection algorithms, DAQ methods and amount of detection rules loaded.

In our study, three common NIDSs were tested in our experimental network and UoA campus network based on the following scenarios. First, we used the default NIDS configuration for finding the baseline of using NIDSs in the high-speed network, such as the packet drop rate, the CPU usage, and the memory usage. Second, we tested different detection algorithms. By default, the NIDSs come with different pattern matching algorithms. According to the specifications of both Snort and Suricata, different methods offer different performance effects on memory consumption, queue size, and CPU usage. We will describe some of the possibilities for improving NIDS performance by modifying or configuring the pattern matching algorithm. And third, we tested different network socket packet capturing mechanisms. Based on (SANS, 2016), a suitable capture mechanism can reduce the packet drop rate. In this test, we tried to understand which method is more suitable for our network environment based on the performance results. Finally, we optimised the number of rules loaded. The NIDSs come with a general configuration that enables all the rules or scripts. However, depending on the user environment, some rules will not be used. To some extent disabling the unnecessary rules may improve the performance results.

For each scenario, we have measured the following information: number of packets received, number of packets dropped, average CPU usage, and memory usage.

We have not considered hard disk read or write usage because, in our tests, we were only writing log information to the hard disk. We verified that the disk operation cost was very low and that our Hard Disk Drive (HDD) configuration was fast enough to handle these operations. Therefore, we decided that analysis of the HDD usage was not necessary.

Experiment Results

This section presents the results of our experiments. In each experiment, we considered performance factors including CPU usage, memory usage, received and dropped packets. The first experiment scenario defines a baseline for evaluating performance, where we use default NIDS configurations. For subsequent experiments, we will discuss changes made to the system. For each scenario, we launched an individual NIDS instance for one hour. In each experiment, we averaged all the measurements over 10 iterations.

- Experiment 1. Default OS & NIDS Configuration: In the first test case, all three intrusion detection systems were set up using their default configurations.

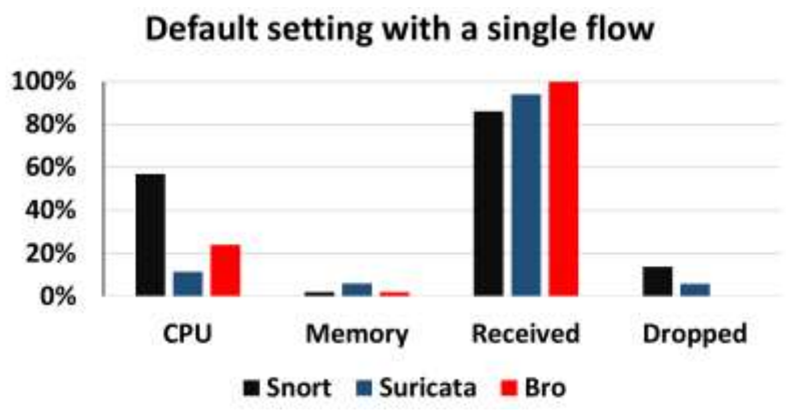

Fig. 3: Percentage of performance with default NIDS configuration with a single flow.

Fig. 3. shows the results of three NIDSs in their default configuration. In our experiment, we tested the three NIDSs separately to monitor the traffic on our high-speed experiment network (10 Gb/s) with a single flow. We observed different performance results. When it came to packet drop rate, Snort dropped 13.8% packets during the 1-hour testing. Suricata dropped 5.9% and Bro achieved the best drop rate (0.1%). In terms of CPU usage, Snort used more than 57% of CPU resources, Suricata launched seven threads in two different cores, the average CPU usage was 11.5%. Again, Bro used less CPU than Snort, it utilised 24% of CPU resources.

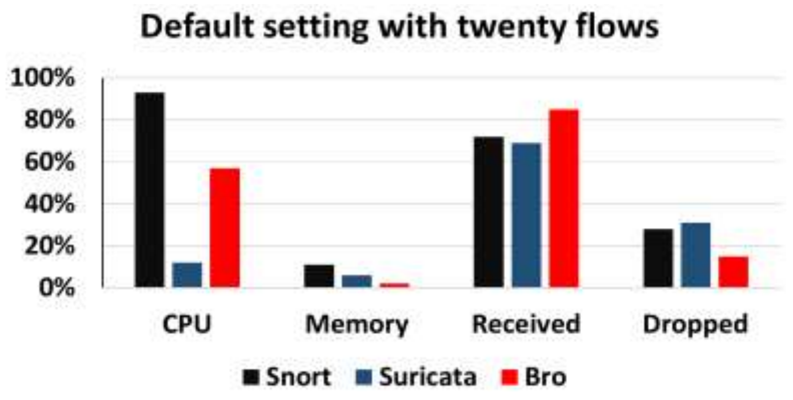

Fig. 4: Percentage of performance with default NIDS configuration with 20 flows.

Fig. 4. shows the results of three NIDSs in their default configurations for monitoring the traffic on our high-speed experimental network (10 Gb/s) with 20 flows. In this test case, CPU usage for Snort and Bro was increased. In particular, Snort used 93% of CPU resources. For the drop rate, all three NIDSs show the default configuration is less efficient for monitoring traffic on a high-speed network with multiple flows running in parallel. We notice that the number of flows can affect the performance of NIDSs. However, our test environment only generated TCP flows, so in an attempt to better understand how the performance can be affected by the mix of flows, we launched three NIDSs in our campus network link with their default configurations.

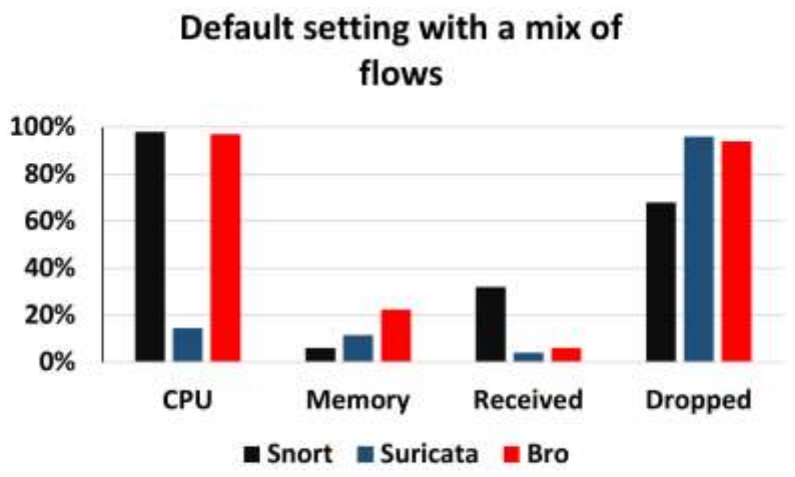

Fig. 5: Percentage of performance with default NIDS configuration with a mix of flows.

Fig. 5. clearly demonstrates that single-threaded NIDSs resulted in an overloaded CPU. As the transmission volume increases, the CPU load is also increased, both Snort and Bro result in nearly 100% CPU usage. Compared to the other two NIDSs, Suricata launched 37 threads across a quad-core system, the average CPU usage was 14.5%. When we tested these NIDSs in our campus network link with a mix of flows, the results show that the default configuration is less efficient if used in the high-speed network environment with multiple flows running in parallel.

- Experiment 2. Optimise each NIDS configuration by modifying the detection algorithm.

In the first test case, we demonstrated that the default configuration is not suitable for monitoring high-speed network traffic on the 10 Gb/s campus network link. The CPU usage and drop rate were very high. In this test, we focus on a simple change to the NIDS detection algorithm to achieve better results.

Detection Engine Pattern Matching Algorithm: We compared all available detection engine pattern matching mechanisms in Snort. The results show that 'ac-nq' offers high performance with a low memory consumption in all the test cases. We also evaluated other pattern matching algorithms mentioned in the Snort.conf file.

Fig. 6. demonstrates the performance results of using the 'ac-nq' pattern matching algorithm in Snort and the 'ac-gfbs' pattern matching algorithm in Suricata. We tested both algorithms in the following test cases: 1 flow, 20 flows, and the UoA campus link.

We evaluated all the pattern matching algorithms in Snort and Suricata. Some algorithms, although they generated similar performance results, required more CPU resources. A case in point is that both 'ac-bnfa' and 'ac-nq' in Snort captured 99.9% of packets; however, the former generated 92% CPU load while the latter only used 65% CPU to process 20 flows. The memory usage for Snort seems very constant for all scenarios, though we did not observe significant changes when we configured the different algorithms. The average memory consumption was 3% of the total memory size.

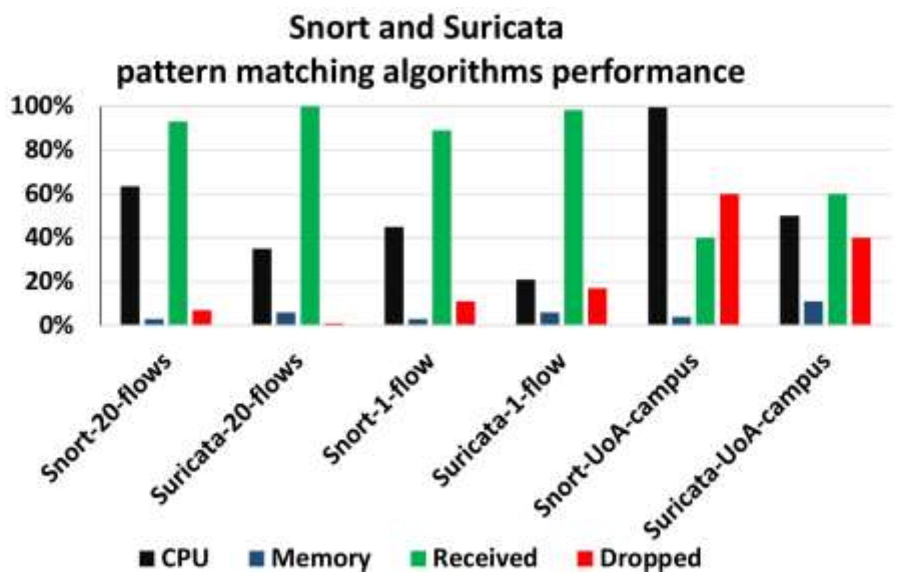

Fig. 6: Percentage of performance after optimisations of ac-nq in Snort and ac-gfbs in Suricata.

Similar results were observed for Suricata, the default algorithm ('ac-gfbs') dropped 5% packets when monitoring a single flow. In contrast, 'ac-gfbs' offers good performance with moderate CPU usage and low memory; the number of dropped packets reduced to 4.9% for the same test case. Considering that other detection algorithms offered a similar drop rate with much higher memory consumption (3-5 GiB), it motivated us to choose a low memory usage algorithm.

As we can see above, our different network environments yield different performance results. Configuration choices directly affected the packet drop rate, users can configure different algorithms based on their network environment requirements, such as hardware, bandwidth, and the expected mix of flows.

It is important to note that, unlike Snort and Suricata, Bro does not allow the user to modify the pattern matching algorithm. In addition, when we used the optimised pattern matching algorithm in the University of Auckland campus network link, Snort reduced its drop rate by 30% and Suricata increased its receive rate by 57% compared to that with the default pattern matching algorithm. However, the drop rate was still higher than the receive rate, which affects the detection result. In the next experiment, we tried to evaluate the AF Packet DAQ methods with the different pattern matching algorithms.

- Experiment 3. Modify DAQ mechanisms to improve capturing performance.

AF PACKET is the Linux native network socket. It functions similarly to the memory mapped libpcap DAQ (Deri, 2005), but no external libraries are required. Based on our hardware requirements, we have configured the DAQ buffer memory to 1 GiB in order to explore the similarities and differences with libpcap. For improving the processing time and reducing CPU usage, Suricata introduced the Zero Copy mode for sharing a packet's memory address with the capture process, so the process can read packets from their original memory address instead of receiving them from the kernel.

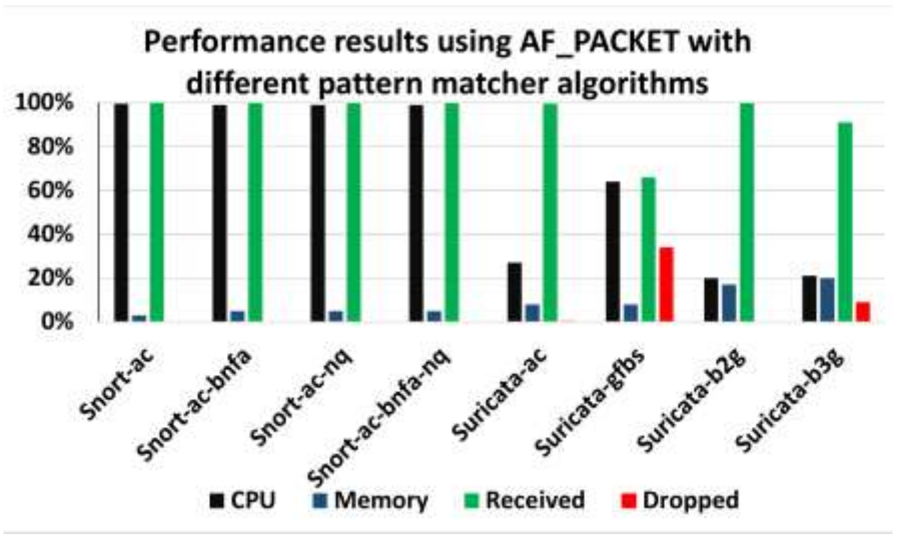

Fig. 7: Percentage of performance utilised after using AFPacket DAQ method in Snort and Suricata.

Bro does not support AF PACKET capturing mode, so we only built test cases for Snort and Suricata for comparing the results to previous libpacp results. In this experiment, we changed the data capture method to AF PACKET, and evaluated all pattern matching algorithms provided by Snort and Suricata. Fig. 7. illustrates the performance results for Snort and Suricata when used in our 10 Gb/s campus network link.

Both NIDSs have significantly improved results compared to the default DAQ (libpcap packet capture engines). With AF PACKET, most pattern matching algorithms were dropping less than 0.3% packets overall; however, the 'gfbs' and 'b3g' algorithms in Suricata did not manage to achieve the best results using AF PACKET; the former dropped 34% packets and the latter dropped 9%. The CPU usage remained similar to the previous results; Snort's CPU usage is affected by the number of mix flows. The average CPU usage for Suricata is 20% except for the 'gfbs' algorithm (64%), we performed the 'gfbs' algorithm in the UoA campus network link more than 10 times, but the results remained the same.

For a single flow as well as for 20 flows, the results remained similar to the UoA testing results.

We also evaluated the PFRING[v] capture method for Bro and Suricata. PFRING provides a circular (ring) buffer and the applications read packets from this buffer. It accelerates the packet capture operation by distributing packets to multiple application processes simultaneously. More detailed explanations can be found from the PFRING project homepage (Ring). Suricata was able to use PFRING and generate similar performance results to AFPACKET. Unfortunately, we were not able to get PFRING working as a DAQ for Bro. For some reason, DAQ would not load the PFRING capture library. As mentioned in existing studies (SANS, 2016), (MetaFlows), using PFRING gave the best performance in terms of CPU usage and packet drop rate.

- Experiment 4. Evaluate each NIDS accuracy by launching some existing attacks.

In this experiment, we try to understand how well each intrusion detection system can detect a variety of malicious packets sent to it. We evaluate the alarm accuracy of each tool by comparing the false alarms and accurate alerts generated by each IDS. In order to simulate the test scenarios, we installed the Kali Linux penetration testing platform on the sender end, which will launch penetration testing scripts and send malicious packets to a receiver (192.168.6.1). In parallel, we use GridFTP in the Sender (192.168.6.2) to transfer big data volumes at 10 Gb/s to the Receiver (192.168.6.1)

- *Port scanning* is a very traditional attack. This is often used by attackers to discover which hosts are opening ports, in order to find out whether a host is running some vulnerable service. Every port has three different statuses: it can be an opening port, a closing port or a filtering port. To discover an opening port, an attacker sends a request to a target system. That request contains a specified destination port. If a relevant response comes back from the target system, it indicates that the port is opened. Otherwise, if the attacker receives a reject packet, the attacker can guess that the port is closed. Firewall rules prohibit packets from passing through a filtered port; in such a case, the source host may not get back anything at all. We used Nmap2 from the Kali Linux penetration testing platform to launch a TCP SYN scanning attack. It sends a SYN packet to the target host and waits for a response. A SYN/ACK packet can help an attacker by indicating which port is listening. If an attacker receives an RST packet or no response is received, the attacker marks the port as either a non-listening or a filtered port. As a result, it discovered all the open ports on our target host and detected which operating system was running. The experimental results show that we sent 4244 SYN requests to probe different ports on our receiver host. Three IDS tools were able to recognise those UDP requests in our high-speed network with the optimised setting and no packet drops were observed in this test.
- *Fragmentation attack* means that attackers send fragmented malicious packets to bypass the checks carried out by firewalls. For instance, if an attacker wants to start a telnet session to a target machine, but port 23 is blocked by a packet filtering firewall, the attacker can easily pass the firewall by sending a first packet that sets the fragment offset to 0, the DF flag to 0, the MF flag to 1 and the destination port to 25, and a second packet that contains the fragment offset 1 and the destination port 23. When the target host reassembles both packets, the second packet will overwrite everything in the first packet. The firewall will not put the second packet into the ruleset, because the offset in the second packet is greater than 0, which means this is a fragment of another packet (the first packet in this case). In such a case the malicious packet can be delivered to the target host without the firewall checking. We used Scapy from the Kali Linux penetration testing platform, it generates customized TCP packets and sends to our receiver host. Similar to the previous attack, we ran the 10 Gb/s background traffic in parallel. In this test, we observed that Snort, Suricata, and Bro provided a detailed view of the logs showing how many fragmented packets were received. From the result, we found that as the MTU size increased, the number of fragmented packets decreased.

  To summarise experiment four, in which we used three IDS tools running their default rule sets, we find that all IDSs generate alerts on known malicious traffic, such as the TCP SYN scanning attack or the fragmentation attack. All in all, we delivered traffic with a high transmission rate and sent malicious traffic at the same time. Snort, Bro, and Suricata are able to detect and generate alerts for most attacks with the optimised configuration.

Discussion

Our results indicate that deploying a suitable NIDS in networks requires a lot of research and repeated evaluations. After analysing and comparing the results from all experiments, we find:

- *Dropped Packets*. In terms of dropped packets, Suricata and Snort achieved the desired result after optimising the configuration and the capture method, 0.3% dropped packets were detected for our campus network link. Unfortunately, some of Suricata's pattern matching algorithms still dropped packets in the University of Auckland campus network link. For instance, the 'gfbs' pattern matching algorithm in Suricata dropped 34% incoming traffic.
- *CPU Usage*. When we discuss CPU usage, lower results are better. It seems that in the mixed flow environment, the single thread NIDS generated a high CPU load on a single core processor, while Suricata reduced the load on a single CPU by utilising all the processor cores. The average CPU usage for Suricata is 23%.
- *Memory Usage*. NIDSs seem to use only a small amount of memory resources. Note that our different network environments did not have any significant effect on memory consumption. The memory for three NIDSs is affected by the different searching algorithms, the DAQ method and the number of detection mechanisms loaded.

Conclusions and Recommendations

In this paper, we provided an overview of three popular open-source NIDS tools along with their comparative performance benchmarks. We evaluated the feasibility of using each of three NIDS tools in a high-speed network. We also proposed possible performance improvement by tuning the following: the pattern matching algorithms, the packet capture methods and the network environment. In our network environment, Suricata with its multi-threaded architecture works better than Snort and Bro. In order to reuse the existing Snort or Bro solutions in the high-speed network, we found some alternative configurations for improving the performance of Snort and Bro. Abaid et al. (Abaid, Rezvani, & Jha, 2014), for example, introduced the use of an SDN splitting incoming flows and distributing traffic load across a range of Bro instances. In such system, each Bro instance only processes a small amount of traffic. Subsequently, we evaluated the different algorithms and methods for Snort, and observed a significant performance improvement after optimising Snort's configuration in the high-speed network.

For future, developing a generic high-speed packet capturing method would an interesting line of work. This new method could ideally be used in all the platforms, thus requiring fewer configurations. Another research direction would be an investigation of detection accuracy and response time of NIDS in the presence of different types of network attacks. Finally, we would like to study how an open-source IDS should be designed to more readily facilitate new emerging IPv6 attacks.

Acknowledgements

This work was conducted as part of a University of Auckland Ph.D. research project. We thank Dr. Se-Young Yu for assistance with the GridFTP setup, and Dr. Tim Chown for comments that helped us in improving the quality of our work. We would also like to show our gratitude to the ITS security team from the University of Auckland that provided us with some insights, and their expertise that assisted us in conducting our research

References

Abaid, Z., Rezvani, M., & Jha, S. (2014). MalwareMonitor: an SDN-based framework for securing large networks. Paper presented at the Proceedings of the 2014 CoNEXT on Student Workshop.

Alhomoud, A., Munir, R., Disso, J. P., Awan, I., & Al-Dhelaan, A. (2011). Performance evaluation study of intrusion detection systems. Procedia Computer Science, 5, 173-180.

Deri, L. (2005). nCap: Wire-speed packet capture and transmission. Paper presented at the End-to-End Monitoring Techniques and Services, 2005. Workshop on.

Garcia-Teodoro, P., Diaz-Verdejo, J., Maciá-Fernández, G., & Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. Computers & Security, 28(1), 18-28.

Khoumsi, A., Krombi, W., & Erradi, M. (2014). A formal approach to verify completeness and detect anomalies in firewall security policies. Paper presented at the International Symposium on Foundations and Practice of Security.

Liu, X., Burmester, M., Wilder, F., Redwood, W. O., & Butler, J. (2014). Zero-Day Vulnerabilities. Coast Guard Journal of Safety & Security at Sea, Proceedings of the Marine Safety & Security Council, 71(4).

MetaFlows. Open Source IDS Multiprocessing With PF RING. Retrieved from https://www.metaflows.com/features/pf_ring

Paulauskas, N., & Skudutis, J. (2008). Investigation of the intrusion detection system" Snort" performance. Elektronika ir elektrotechnika, 15-18.

Ring, P. PF Ring Introduction. Retrieved from http://www.ntop.org/products/packet-capture/pf_ring

Salah, K., & Kahtani, A. (2010). Performance evaluation comparison of Snort NIDS under Linux and Windows Server. Journal of Network and Computer Applications, 33(1), 6-15.

SANS. (2016). Open Source IDS High Performance Shootout. Retrieved from https://www.sans.org/reading-room/whitepapers/intrusion/opensource-ids-high-performance-shootout-35772

Shukla, J., Singh, G., Shukla, P., & Tripathi, A. (2014). Modeling and analysis of the effects of antivirus software on an infected computer network. Applied Mathematics and Computation, 227, 11-18.

Symantec. (2016). Internet security threat report. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf

Wright, M. A. (2000). Virtual Private Network Security. Network Security, 2000(7), 11-14.

Yu, S.-y., Brownlee, N., & Mahanti, A. (2014). Performance and Fairness Issues in Big Data Transfers. Paper presented at the Proceedings of the 2014 CoNEXT on Student Workshop.

Endnotes

i https://www.symantec.com

ii https://www.snort.org

iii https://www.bro.org

iv https://suricata-ids.org

v https://www.ntop.org/products/packet-capture/pf_ring