Analysing Performance Issues of Open-source Intrusion Detection Systems in High-speed Networks

Qinwen Hu[a], Se-Young Yu[b], Muhammad Rizwan Asghar[a]

[a]*School of Computer Science, The University of Auckland, New Zealand*
[b]*International Center for Advanced Internet Research, Northwestern University, USA*

Abstract

Driven by the growing data transfer needs, industry and research institutions are deploying 100 Gb/s networks. As such high-speed networks become prevalent, these also introduce significant technical challenges. In particular, an Intrusion Detection System (IDS) cannot process network activities at such a high rate when monitoring large and diverse traffic volumes, thus resulting in packet drops. Unfortunately, the high packet drop rate has a significant impact on detection accuracy. In this work, we investigate two popular open-source IDSs: Snort and Suricata along with their comparative performance benchmarks to better understand drop rates and detection accuracy in 100 Gb/s networks. More specifically, we study vital factors (including system resource usage, packet processing speed, packet drop rate, and detection accuracy) that limit the applicability of IDSs to high-speed networks. Furthermore, we provide a comprehensive analysis to show the performance impact on IDSs by using different configurations, traffic volumes and different flows. Finally, we identify challenges of using open-source IDSs in high-speed networks and provide suggestions to help network administrators to address identified issues and give some recommendations for developing new IDSs that can be used for high-speed networks.

1. Introduction

Intrusion Detection Systems (IDSs) have played a significant role in detecting malicious activities in a network and the hosts connected to it. IDSs such as Snort, Bro, and Suricata, are used for identifying potential attacks on today's networks; however, there are performance limitations of IDSs with currently available high-speed networks. There have been several studies [1, 2, 3, 4] that focus on two main aspects of IDS performance: the first one is to find and reduce factors that affect IDS performance; the other one is to improve the overall IDS performance.

Some studies [1, 2, 3] find that IDS performance can be influenced by various factors such as IDS configuration, the number of network flows[1], and flow durations. For instance, Salah *et al.* [1] and Alhomouda *et al.* [2] discovered that different Operating Systems (OSs) and platforms could impact IDS performance. Salah *et al.* [1] found that Snort performs better in the Linux environment for handling 1 Gb/s traffic. Alhomouda *et al.* [2] measured the performance while using Suricata on FreeBSD for monitoring unauthorised activities with a revvast volume of background traffic. Hu *et al.* [3] explored that IDS performance can be affected by different flows with different durations.

Both Snort and Suricata use a regular expression to match attacker's patterns in network traffic. However, with the large traffic volume, matching packet's data using regular expressions consumes a significant amount of system resources and becomes performance bottleneck during the packet detection procedure. Antonatos *et al.* [5] discovered that in Snort, string pattern matching consumes 40-70% of the total processing time. For this reason, some existing studies [6, 7, 8] suggest improving the regular expression matching architecture in order to improve IDS performance. Yang *et al.* [7] proposed a novel Deterministic Finite Automata (DFA) accelerated architecture that improves the throughput of DFA while managing memory efficiently. Their solution leverages three Field Programmable Gate Arrays (FPGA)-based algorithms: Simple State Merge Tree (SSMT), Distribute Data in Round-Robin(DDRR), and Multi-path Speculation, which make the serial DFA matching can be parallelised and pipelined. They tested this architecture in different production environments. Their results show that this new design improves the processing speed by 108 times.

Hu *et al.* [3] highlighted the challenges of using the default IDS packet capturing mechanism and packet detection mechanism in a high-speed network. For instance, they observed less than 10% packet drop rate when they used a default IDS configuration for a 1 Gb/s single flow test network. However, the default configuration gives an abysmal performance (80% packet drop rate, 99% CPU usage) when processing multiple flows on a 2 Gb/s network. After modifying the packet capturing mechanism and the packet detection mechanism, their result shows a significant improvement. The packet drop rate reduced to 1% and the CPU usage to 11.5%. Campbell and Lee [4] introduced

Email addresses: qhu009@aucklanduni.ac.nz (Qinwen Hu), young.yu@northwestern.edu (Se-Young Yu), r.asghar@auckland.ac.nz (Muhammad Rizwan Asghar)

[1] A network flow (*a.k.a.* flow hereafter) is a group of packets having the same (i) source and destination IP addresses, (ii) port numbers, and (iii) the protocol.

a hardware-based solution to reduce the packet detection volume for each IDS instance. Their solution is a hybrid approach that uses a set of Bro police scripts on a load-balancing device to interact with the Bro instances using predefined Application Programming Interfaces (APIs). Their solution reduced the packet detection volume for each IDS instance, while maintaining IDS accuracy at the high-level of effectiveness.

By reviewing existing studies [7, 3, 4, 2], we discovered that CPU usage, memory usage, and packet drop rates of an IDS could be affected by different environments, *i.e.,* packet detection mechanisms, packet capturing mechanisms, the number of flows, and hardware specifications. Many studies [7, 3, 4, 9] have been conducted to improve IDS performance on high-speed networks. However, existing studies focus on investigating IDS performance under 20 Gb/s networks or using driver-dependent module such as PF_Ring [10]. We found that the performance of open-source IDS under 100 Gb/s throughput seems not to have received much attention in the literature.

The objective of this study is to understand the feasibility of popular open-source IDSs, including Snort and Suricata, in a 100 GB/s network without relying on a new packet capturing mechanism or updating the existing hardware. We would like to highlight the challenges of these IDSs in modern high-speed networks and propose optimisations to improve their performance. We list our research questions to fulfil the objective.

There are emerging research questions: do we need a powerful server to run the open-source IDS tool for handling a high-speed throughput? How much memory and how many CPU cores required to support a high-performance IDS? What is the main challenge for running IDS instances and other applications in parallel? Answering these questions provides performance baseline of the IDSs with different packet capturing mechanisms in a high-speed network. We also suggest optimisations for IDSs to maximise their performance in high-speed networks. For example, running IDSs with 60 Gb/s traffic requires a CPU with at least 12 cores to distribute the load from the IDSs and Network Interface Card (NIC). If an IDS system does not have enough resources, it may lead the system overload and cause an IDS to miss malicious activities [3]. Also, we need to understand the challenges of using both IDSs in high-speed networks, such as whether two common packet capturing mechanisms are able to handle 100 GB/s traffic without any missing any packets. Schaelicke *et al.* [11] and Ptacek *et al.* [12] reported that even a limited packet loss is critical to the accuracy of IDSs. Further, we would like to investigate the current mechanisms used by the existing IDSs, whether the current packet capturing mechanisms and packet detection mechanisms can still maintain high efficiency and low packet drop rates under more complex network flows. Besides, both Snort and Suricata have released new versions with some performance improvements, so we want to assess if these new versions of IDSs could be used directly under high-speed network traffic without any configuration changes.

Research Contributions. In this article, we evaluated the feasibility of using IDS with 100 Gb/s traffic, highlighted the main challenge of running IDSs in the high-speed networks, as well as proposed possible solutions. More importantly, we found that many factors affect IDS performance and packet drop rates in the high-speed network, such as different traffic volumes, different flow types and system resource allocation. We summarise our research contributions as follows.

- We assess the performance and accuracy of two open-source IDSs under different network throughputs. Our results show that it is not possible to handle 100 Gb/s traffic using both IDSs with the existing packet capturing mechanisms, including Libpcap and AF_PACKET. We noticed the CPU bottleneck with a default configuration of the IDSs, causing to drop 99.9% packets when an incoming throughput reaches to 40 Gb/s. We found that AF_Packet improved the limitation to 60 Gb/s, but both IDSs started to drop packets above 60 Gb/s. We also discovered that we can capture 100 Gb/s with eXpress Data Path (XDP) in Suricata but with a single detection rule; however, it is not possible to detect any malicious activity with just a single rule.

- We observed that not only a larger volume of traffic affects the performance of IDSs, but also the complexities of the multiple flows impact both performance and accuracy. For instance, Snort 3.0 adopts the multithreaded architecture, which improves CPU usage and reduces packet drop rates compared to the previous experimental results [3]. However, when dealing with a large volume of multiple flows, Snort 3.0 and Suricata 4.1 will suffer from performance degradation. As a result, the CPU usage reaches 99% and the packet drop rate becomes as high as 99.9% when handling 33000 multiple flows per second. As expected, the accuracy of malicious flow detection decreases as the packet drop increases.

- Both Snort 3.0 and Suricata 4.1 have improved resource utilisation compared to their previous versions (including Snort 2.8 and Suricata 3.1.4) two years ago [3]. For example, Snort 3.0 adopted the multithreaded architecture, which optimises CPU usage and reduces packet drop rates compared to the previous results from Snort 2.8 [3]. Nevertheless, when dealing with a larger volume of multiple flows, Snort 3.0 and Suricata 4.1 suffered from competing for the system resources with other applications. The major challenge we found in our experiment is to balance resource allocation among IDS instances, Ipref3 instances, and Soft Interrupt Request(IRQ) instances. Even though we specified different cores for handling SoftIRQ, IDSs, and Ipref3 processes, we found CPU usage from handling SoftIRQ caused to drop packets in Snort and Suricata. When the CPU cannot handle SoftIRQ from the NIC, both IDSs began to drop packets.

The rest of this article is organised as follows. We provide a brief overview of two popular IDSs in Section 2. Section 3 describes our methodology, testing environment, and use case scenarios. In Section 4, we discuss experimental results with different IDS configurations. Section 5 shows the significant

impacts of using existing IDS configurations in high-speed networks. We also provide some solutions to improve IDS performance. Section 6 concludes this article, provides some recommendations for practitioners, and highlights research directions for future work.

2. Open-source Intrusion Detection Tools

In this section, we discuss two open-source IDS tools: Snort and Suricata. Both tools are widely deployed by many organisations [13] to protect their networks. We begin with the design goals and then describe the architectures of both Snort and Suricata. It will help us understand why both tools perform differently even though both implement a multithreaded architecture. The other architectural components of an IDS including the packet capturing mechanisms and the packet detection mechanisms will be covered later. The previous studies [3, 1, 2] mentioned that different packet capturing mechanisms and packet detection mechanisms can affect IDS performance. Our goal is to understand these technologies in detail, then test each of them with suitable configurations in different network environments.

2.1. Snort

Snort is an open-source signature-based detection tool that offers both network intrusion detection and mitigation; it comes with a set of relevant rules and features that detect potential attacks and probes in order to discover security holes. The key idea of Snort's design is to make the open-source IDS flexible enough to configure and deploy in different networks. Unlike some commercial network-based intrusion detection tools (such as Cisco Secure IDS, CyberSafe Centrax, and Network Ice Blackice Defender), Snort allows network administrators to add customised signatures into the existing rule base. Once a new signature is created and enabled, Snort will immediately apply the signature to its intrusion detection process. Snort also supports passive traps to detect malicious traffic headed to ports that are not configured with any services in the network. In general, network administrators are aware of which services are available on their network. Therefore, they can specify Snort rules to watch for traffic that tries to interact with non-existent services. If any incoming packets attempt to call unused ports or services, an alert will be generated and logged. For instance, if a network is not using the File Transfer Protocol (FTP) service, network administrators can configure a Snort rule to raise an 'FTP Probe' alert if they detect that the packet intends to connect to port 21.

2.2. Suricata

Suricata is another open-source IDS, which aims at improving the protocol identification and introducing the script-based detection. For the protocol identification, Suricata allows network administrators to define either the protocol type or the particular port in the rule file. Also, Suricata provides a larger number of keywords that can be used for matching with protocol fields. The pattern matching mechanism in Snort only tests the relevant rules for each incoming packet; there is no obvious way to check for pattern relationships among the packets within a flow. In order to check for pattern relationships, network administrators need to manually compare the previous information with the current content, which is not possible using the pattern matching mechanism. Instead of using pattern matching, Suricata introduces script-based detection and well-designed data structures for parsing and logging flow information for further investigation.

2.3. Design Architecture of an IDS

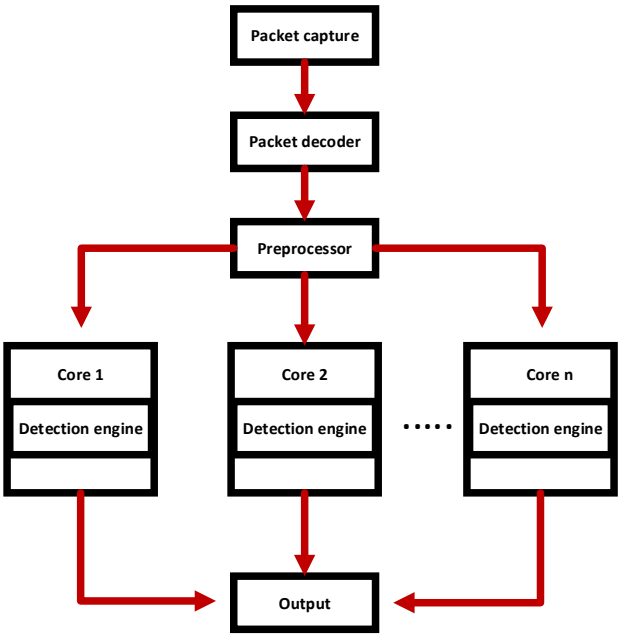


Figure 1: The IDS dataflow is composed of five components including packet capture, packet decoder, preprocessor, detection engine, and the output. The packet capture component is responsible for capturing packets from the network interfaces. The packet decoder analyses the Ethernet header to classify whether the packet is an IPv4 packet or IPv6 and decodes 5-tuple information from the IP and transport layer header. After decoding, the preprocessor may defragment packets and assemble TCP data from multiple packets in the same flows. The core part of IDS is in the detection engine. An IDS runs multiple threads to read packets from the preprocessor and match them against configured rules. Once a match is found, the detection engine notifies the logging and alerting system based on the behaviour defined in the rules. Finally, the system will output the alert or log accordingly.

In the previous section, we discussed Snort and Suricata with their primary design goals. For instance, Snort is a basic signature-based IDS and can be used as a light IDS for solutions that do not require much customisation. Whereas, Suricata is designed for more flexible detection solutions to allow network administrators to customise the script-based detection.

In this section, we will explain how Snort and Suricata detects an attack. Figure 1 illustrates an example of how Suricata and Snort process incoming traffic with multithread detection processors. First, IDSs collect packets using a packet acquisition module (libpcap by default) and pass the captured the packets on to the decoder layer. The decoder layer decodes raw packets based on the protocol types and passes on to the preprocessor. The preprocessor will defragment packets, reassemble

TCP flows, and track TCP or UDP sessions. If any anomaly is detected, the decoder component raises an alert before the detection engine processes the packet. The detection engine is a core component in an IDS; it is designed to access the packet contents and identify any malicious behaviour that matches the detection rules. In the multithreaded mode, network administrators can configure a number of threads to use and which cores to use to run those threads. In this case, each thread acts as a detection engine that processes multiple packets in parallel. If any malicious behaviour is detected, the IDSs notify the network administrator by raising an alert or drop the packet and log alert for the output module.

2.4. Packet Capturing Mechanisms

IDSs need to capture packets from the NIC to forward them to the preprocessor and the main detection engine. Both Snort and Suricata depend on external packet capturing libraries including Libpcap and AF_PACKET. By default, Suricata and Snort use Libpcap as a default packet capturing mechanism.

Libpcap. Libpcap is a hardware-independent open-source library that allows network administrators to capture packets from a NIC. The NIC driver grabs the packets and sends them to the protocol stack. The OS network protocol stack analyses the packet and allocates packets to the relevant application. Libpcap is located at the boundary of the kernel space where it can monitor both incoming and outgoing packets from the NIC. The packet capturing procedure includes three steps:

- *Device Initialisation.* Libpcap allows network administrators to call its *pcaplookupdev()* function to list all network devices, it then uses *getifaddrs()* to get their IP addresses and related information. All such network devices are saved in the pcapif list.
- *Berkeley Packet Filter (BPF) [14].* This provides a filter function for the sniffer so that it can forward only specific packets. The BPF is applied after the driver receives the packets from the network interface.
- *Packet Processing Loop.* Snort calls the *pcapdispatch()* function from the libpcap library to read packets from the NIC. Snort then uses a *PcapProcessPacket()* function to process each captured packet based on different protocol types. The packet decoder passes decoded packets to the preprocessor module for further investigation.

AF_PACKET. *AF_PACKET* [15] is the Linux native network socket. Similar to libpcap, *AF_PACKET* enables network administrators to configure a memory buffer for captured packets. This means that the memory allocated for the buffer is shared with the capture process, so instead of the kernel sending packets to the capture process, the process can just read the packets from their original memory address. This method saves time and CPU resources.

PF_RING. *PF_RING* [16] is another high-performance Linux kernel module that optimises load balancing through the ring cluster design. In the packet capturing process, the application copies packets from the NIC to the *PF_RING* circular buffer. Then, the IDSs read the packets from this circular buffer. *PF_RING* can distribute incoming packets to multiple rings and it allows multiple applications to process packets simultaneously.

2.5. Packet Detection Mechanisms

Traditionally, an IDS inspects packets deeply by scanning every byte of the packet; however, several improvements have been proposed in the last two decades [17, 18, 19, 7]. By reviewing studies in the past [20, 17, 18, 19, 7], we found two packet detection mechanisms that have been used most widely in the current IDS tools.

2.5.1. Aho-Corasick Algorithm

Aho *et al.* [20] proposed a simple and efficient algorithm in 1975 (used in existing IDS tools including Snort and Suricata) that uses a default pattern searching algorithm. In this approach, they use a pattern matching machine to represent a predefined language as a set of strings; network administrators can test whether an input string matches any set of the given strings. The pattern state machine processes an input text string and is composed of three functions: a [goto] function, a [failure] function, and an [output] function:

- A [goto] function constructs a goto graph; the goto graph starts with a root node that represents a state, 1. Each input keyword is entered into a subsequent node. A search starts from state 1 and a path through the graph spells out a keyword. If no failure is detected during the search, the matched keyword will be passed to an output function.
- A [failure] function is triggered when the [goto] function reports failure. For example, if a current input character is not found in the current node or the sub-nodes on the same path, the pattern matching machine will call the [failure] function to search alternative paths for processing the character.
- An [output] function merges duplicated output states into a new output state.

2.5.2. Regular Expression Signatures

A regular expression mechanism is another signature-matching algorithm; it uses character classes, unions, optional elements, and closures to enhance a signature-based IDS flexibility. Moreover, it improves search efficiency by adding effective schemes to perform pattern matching. A normal regular expression can be represented by a finite state automaton. In [21], Hopcroft *et al.* introduced two finite state automata: a Deterministic Finite Automaton (DFA) and a Non-deterministic Finite Automaton (NFA). The DFA takes input symbols and then the transition function outputs a single next state. Instead of returning a single next state, the NFA solution returns a set of states. Existing studies [17, 19, 7] show that NFAs are compact but slow; whereas, DFAs are fast but may require more memory while processing. In the last decade, most studies focused on making

Table 1: The role, the model, and hardware specifications of each tested device.

Role	Model	CPU	Memory	NIC
Sender	Dell PowerEdge R740XD	2 x Intel XEON Gold 6126 2.6 GHz, 12 cores per CPU	196 GB	Mellanox ConnectX-5 100GE
Receiver	Dell PowerEdge R740XD	2 x Intel XEON Gold 6126 2.6 GHz, 12 cores per CPU	196 GB	Mellanox ConnectX-5 100GE
Switch	Dell Z9100	MPC8541	2 GB	Firebolt-3 ASIC

DFAs more efficient, such as [17], where Gong *et al.* reduced the construction time, memory and matching time by using a multi-dimensional finite automaton in the original DFA model.

3. Our Methodology

In this section, we describe our proposed methodology and test environment. For our experiments, we use three different methods to generate high throughput traffic. In the first experiment, we verify the performance of IDSs under a controlled environment and use Iperf3 to generate multiple TCP flows with a packet size of 1500 bytes. We measured the performance of each IDS with increasing throughput from 10 Gb/s to 100 Gb/s to test their capability of handling packets in a 100 Gb/s network. In the second experiment, we measure the detection accuracy under high throughput traffic along with some malicious traffic. To achieve our goal, we introduce Pytbull, which is a flexible IDS and Intrusion Prevention System (IPS) testing framework that covers a broader scope of attacks. We run Iperf3 and Pytbull at the same time to verify whether each IDS can detect the attacks under a certain amount of background traffic. In the third experiment, we assess the accuracy of each IDS with real-world traffic. To this end, we use TRex to generate L4 to L7 traffic based on their real-world traffic templates. We extend the first and second experiments by testing each IDS and its packet capturing mechanism with real-world background traffic and some malicious traffic. To conduct our experiments, we set up a 100 Gb/s testbed with three machines. Our testbed uses two Dell PowerEdge R740XD servers: one as a sender and the other as a receiver. Also, we use a Dell Z9100 as a switch connecting the two. The hardware detail is specified in Table 1.

3.1. First Experiment: Performance Checking

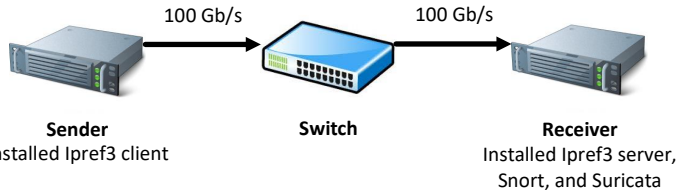


Figure 2: Two test machines were configured for our 100 Gb/s test environment. A sender used Iperf3 to deliver large quantities of data. Two IDS tools were installed on the receiver side to measure IDS performance while monitoring 100 Gb/s TCP flows with a longtime duration.

This experiment aims to evaluate the performance of Snort and Suricata while processing a TCP flow throughput from 10 Gb/s to 100 Gb/s. As a result, the experiment was set up in a controllable environment, and there is no background traffic between two machines. The experiment consisted of a logical network diagram as shown in Figure 2. Both IDSs installed on a receiver server. We used Iperf3 to generate the TCP flow with a packet size of 1500 bytes which is the most common Maximum Transmission Unit (MTU) size for the commodity Internet. The test starts with a default configuration and rule set. We used *node_exporter* with *prometheus* to monitor the CPU, memory and network utilisation along with *tcpdump* to monitor the packet drop rate. Our previous study [3] shows that different packet capturing mechanisms and packet detection mechanisms can impact the CPU and memory usage as well as the packet drop rate. Therefore, in our first experiment, we generated a single TCP flow for 1800 seconds and measure IDS performance. Based on the result, we modify the packet capturing mechanisms and the packet detection mechanisms to optimise the performance. When we find the best combination of the packet capturing mechanism and the packet detection mechanism, we run the same test 10 times and take an average result.

3.2. Second Experiment: Accuracy Checking

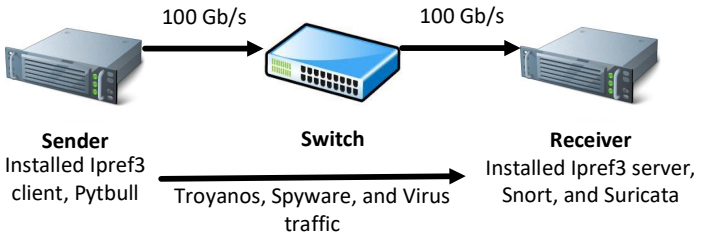


Figure 3: Two test machines were configured for our 100 Gb/s test environment. A sender used Iperf3 to deliver large quantities of data and Pytbull to generate malicious packets in parallel. Two IDS tools were installed on the receiver side to measure IDS performance as well as the accuracy while monitoring 100 Gb/s TCP flows.

The second experiment is a stress test to investigate how accurately the rule set of Snort or Suricata to classify the legitimate and malicious traffic under a 100 Gb/s TCP flows. We set up two experiments to test the accuracy of Snort and Suricata: (i) measuring the false positive rate, the true positive rate, and the packet drop rate of both IDS tools with malicious traffic and without any background traffic; and (ii) measuring the false positive rate, the true positive rate, and the packet drop rate of both IDS tools with combined legitimate and malicious traffic at a fixed 100 Gb/s TCP flow. We used a default rule set from both IDS tools and install Pytbull on the sender (see Figure 3). We launched 7 test cases to assess if rules defined within the default rule will detect these attacks. In this test, we analysed detection accuracy of both Snort and Suricata while processing the legitimate and malicious network traffic under a heavy

load. If there is any packet drop in this experiment, we want to investigate whether it affects the IDS detection accuracy or not.

3.3. Third Experiment: Checking Both Performance and Accuracy

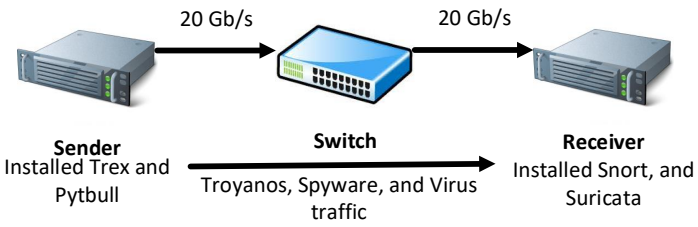


Figure 4: A sender used Trex to generate different flows with different time duration. The throughput is configured between 1 Gb/s to 20 Gb/s. Pytbull is installed on the sender for checking IDS accuracy when we simulate different throughput. An IDS is installed at the receiver side, it monitors all incoming traffic.

In the first and second experiments, we built a testbed to emulate network traffic in a controlled environment, such as we set up the throughput, the network protocol and the number of flows. In the third experiment, we tested both IDSs with a real-world scenario, where we generated packets based on the live capture from Trex[2]. Our aim is to analyse the feasibility of real-time intrusion detection by processing traffic with different packet sizes or different protocols. However, our campus network only has 2 Gb/s live traffic, which is not sufficient for our experiment. So, we decided to use Trex to simulate the real world scenarios, such as the flows contain different protocols, each flow has a different ending time, and the throughput can be adjusted based on our requirements. Similar to the previous experiments, we run our IDS tools at the receiver side (see Figure 4), it takes all incoming traffic from the sender. The only difference is that the receiver is not running the Ipref server in this experiment; as a result, we have more cores to be allocated to the IDS packet capturing process or the packet detection process. Besides, we extend the second experiment by monitoring flows that contain different protocols.

For each experiment, we measured the following: 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 an analysis of the HDD usage was not necessary.

4. Experiment Results

This section presents the results of our experiments described in Section 3. In each experiment, we monitor performance factors including CPU usage, memory usage, and packet drop rates. We start with the default configuration to compare the performance of Snort and Suricata using a single 10 Gb/s flow. Next, we adjust the packet capturing mechanism and the packet detection to measure the performance difference. For each experiment, we run Snort and Suricata individually and repeat the experiment multiple times. For our performance results, we took an average of all the test runs.

Inspired by [22, 23], we collected our comparison data from the IDS itself and the traffic generators, including IPerf3 and TRex. Both IDSs provide data on actual attacks and statistic of the traffic being analysed along with the packet drop rate. We launched our test in a controllable environment to test Suricata and Snort with the different packet capturing mechanisms under different traffic. For each test, we also collected the amount of ingress and egress of the network traffic as well as the CPU load and memory usage of the machine the tests were running on.

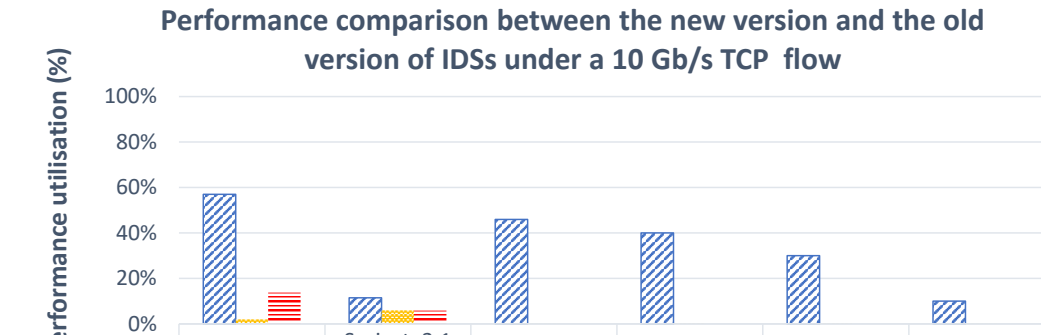
IDS Performance Checking. We divide our first experiment into two phases. First, we make a performance comparison between different versions of IDSs in a 10 Gb/s network. Then, we put the newer version of IDSs under different throughput and made a performance check against the default configuration. Furthermore, we adjust the packet capturing and packet detection mechanisms of the IDSs based on the packet drop rates. Schaelicke *et al.* [11] discovered a linear relationship between packet loss and precision loss in IDSs. The similar result has been discovered by Ptacek *et al.* [12], where they showed that attacks can bypass detection by overloading the IDS, causing high drop rates and increasing the chances that a successful intrusion remains undetected. Their results indicate that any packet loss can directly degrade the effectiveness of the IDS. In this work, we discard an IDS with a specific packet capturing mechanism for the further experiment once it starts losing packets.

The performance comparison results of different versions of IDSs are shown in Figure 5. The results show that the newer versions of Snort have improved their performance in terms of the CPU and memory usage and the packet drop rate in terms of resource consumption. We notice that Snort 3.0's CPU usage was 11% lower than that of Snort 2.8, its memory usage dropped from 2% to 0.1% while processing the same 10 Gb/s TCP flow. Compare to Snort, Suricata's changes are not significant. Suricata 4.0 used 16 cores from the receiver server, and each core consumed 10% CPU for processing 10 Gb/s traffic. The collected performance data shows that Suricata 4.0's memory usage is less than that of Suricata 2.10 as illustrated in Figure 5. Fortunately, the packet drop rate of Suricata 4.0 had decreased from 5.9% to 0%.

For testing the performance of newer versions of IDSs in high-speed networks, we used with Snort 3.0 and Suricata 4.0 with default configurations and generated one TCP flow with a single 20 Gb/s flow and then two flows with 10 Gb/s throughput for each.

From our results, we observe that a single 20 Gb/s flow overloaded the receiver with too many interrupts. As a result, IDSs dropped a lot of incoming traffic. To reduce the packet drop caused by the interrupt, we used two 10 Gb/s flows and

[2] https://trex-tgn.cisco.com



	Snort2.8 10 Gb/s AF_PACKET	Suricata2.10 10 Gb/s AF_PACKET	Snort3.0 10 Gb/s Libpcap	Snort3.0 10 Gb/s AF_PACKET	Suricata4.1 10 Gb/s Libpcap	Suricata4.1 10 Gb/s AF_PACKET
CPU usgae	57%	11.50%	46%	40%	30%	10%
Memory usage	2%	6%	0.10%	0.20%	0.20%	0.10%
Drop rate	13.80%	5.90%	0%	0%	0%	0%

Figure 5: Comparing the performance of the different versions of IDSs: we set up a network environment with a 10 Gb/s TCP flow. The name of each title in the table is defined based on the following naming convention: IDS name, version number, test environment, and the packet capturing mechanism. For instance, if we test Snort 3.0 in a 10 Gb/s network to assess IDS performance using Libpcap then we call this test Snort3.0_10 Gb/s Libpcap.

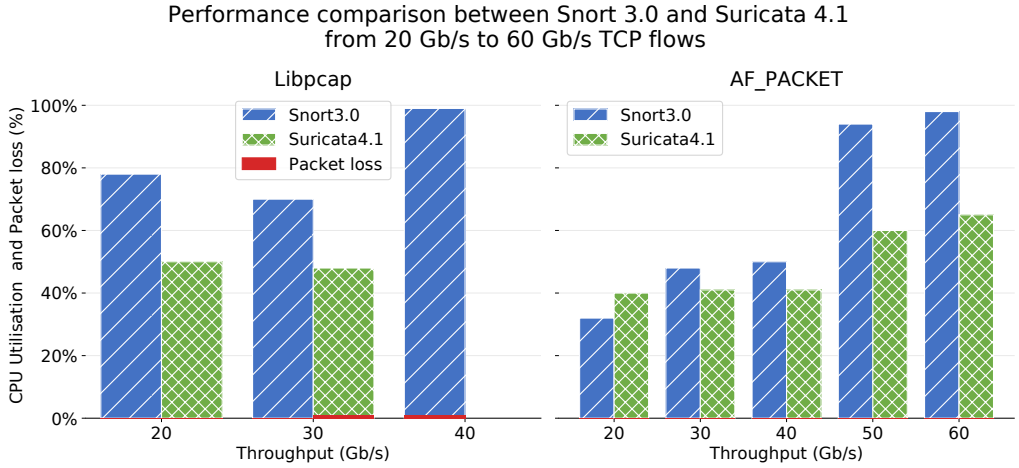


Figure 6: Performance comparison between the new version and old version of IDSs having 20 to 60 Gb/s TCP flows, 10 Gb/s per flow using different packet capturing mechanisms: The left side of the y-axis shows CPU usage, and the right side indicates the packet drop rate. Overall, Snort 3.0 and Suricata 4.1 show a high-performance result when AF_PACKET was used as a packet capturing mechanism. Memory utilisation throughout the experiments stays at 10%.

used the Receiver-Side Scaling (RSS) to spread hardware interrupt. We further increased the throughput by increasing the number of 10 Gb/s flows.

We started the throughput from two 10 Gb/s flows and measured CPU, memory, and packet drop rate when using Libpcap and AF_PACKET. We removed an IDS with a specific packet capturing mechanism from further experiments when it starts dropping packets because a small percentage of packet loss can cause IDSs to lose track of the potential attacks [11]. We also want to compare the absolute performance of each IDS with different packet capturing mechanisms in terms of their capability of processing throughput without packet loss. When we tested Suricata with three 10 Gb/s throughput, we discovered that Libpcap had 1% drop rate and we removed the Suricata with Libpcap from the experiments with more number of flows. Fortunately, Suricata with AF_PACKET did not lose any packet until the throughput reaches 60 Gb/s.

Based on Figure 6, we find that the Libpcap was not satisfactory when the throughput is over 40 Gb/s. Therefore, we decided to use AF_PACKET as the packet capturing mechanism for Snort and Suricata to monitor the traffic over 40 Gb/s. After applying AF_PACKET, Suricata's average CPU increased to 60% while monitoring 50 Gb/s traffic, and the packet drop rate decreased to 0%. We use the same packet capturing mechanism in Snort 3.0, the CPU utilisation of Snort is higher than Suricata 4.0 while processing traffic under 50 Gb/s throughput. Snort's CPU consumption was 94% along with 0% packet drop rate. We tried up to 60 Gb/s because we observed some packets being dropped. However, we discovered that Snort and Suricata do not accurately reflect the packet drop rate from the network layer aspect. For example, when we tested IDS performance using the 60 Gb/s throughput, both Snort 3.0, and Suricata 4.0 showed that the packet drop rate was 0% with AF_PACKET. After we deeply analysed how many packets were sent from the sender, we found that the receiver side lost about 0.01% traffic.

All in all, our results statistically displayed the process with TCP flows under different throughput and resource overheads. It clearly shows that when the throughput starts to increase, an IDS consumes more resources to maintain the low packet drop rate.

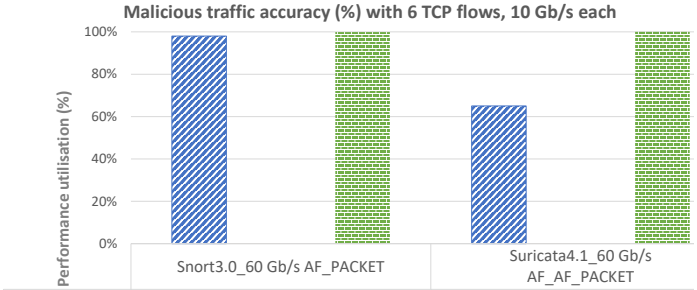


Figure 7: Evaluating the accuracy of both IDSs under 60 Gb/s throughput using Pytbull to generate the same attacks for both Snort and Suricata.

IDS Accuracy Checking. In the second experiment, we explored how accurately Snort 3.0 and Suricata 4.0 can classify the legitimate and malicious traffic under 60 Gb/s throughput. We used Pytbull, an open-source IDS test framework to test specific attack scenarios. Each attack scenario assesses the default rule set in Snort 3.0 and Suricata 4.0 and targets the relevant alert. We ran both IDSs with the default rule set and configurations. The attack detection rates of both IDSs are shown in Figure 7. The difference between both IDSs is minimal. Suricata detected all the malicious traffic using the default rule set; whereas, Snort missed a few anomaly packets. This difference suggests that Snort needs to add those missed rules to its default rule set. Moreover, from the performance aspect, the second experiment produced the same performance results as the first experiment for processing 60 Gb/s traffic. This result shows that the accuracy of IDSs is not affected when there is less than 15% of packet drops. Actually, IDSs can maintain a good balance between accuracy and performance with a simple network environment.

IDS Performance and Accuracy Checking. The third experiment focused on testing performance and accuracy while processing multiple flows with different throughput. We used Trex, an open-source traffic generator, which can be used to generate Layer 4 (transport layer) to Layer 7 (application layer) traffic based on preprocessing and replay the Libpcap file that contains real traffic. We evaluated the performance of Snort 3.0 and Suricata 4.0 under different throughputs (20 Gb/s to 60 Gb/s). To this end, we used Trex to generate flows using different packet sizes, different protocols, but with the same flow duration. From our existing study [3], we showed there is a performance bottleneck when processing a large number of multiple flows using default configurations in Snort. So, we would like to see if this performance issue has been addressed by using the multithreaded architecture in Snort. We observed that both Snort 3.0 and Suricata 4.0 show the CPU and memory overheads along with high packet drop rates while processing more than 33000 flows per second, where the flow duration was 40 milliseconds.

We started with the default configuration (the packet capturing mechanism is Libpcap, and the packet detection is Aho-Corasick (AC)) and observed that Suricata used 16 cores and consumed 99% CPU per core. The same behaviour was found in Snort 3.0; it consumed 100% of 16 cores to process the same size of traffic. When Trex stopped sending traffic, we stopped the Snort and Suricata instances manually, and we found that the packet drop rate was close to 100% when Libpcap was used. To understand whether IDS performance can be improved by modifying the packet capturing mechanism and packet detection mechanism, we launched different experiments with different configurations. The best result from these experiments is with AF_PACKET in Suricata. Our results show that the packet drop rate of Suricata is down to 68% after using AF_PACKET. As for Snort, no matter whichever combinations we choose, Snort's performance showed no difference.

5. Discussion

First, we explain the limitations of existing IDSs, then we discuss insights from the experiment. We found a large volume of multiple small flows can impact the CPU and memory usage as well as lead to the higher packet drop rate. The packet detection mechanisms require more memory to process traffic with the predefined rule set in the high-speed network. The existing packet capturing mechanisms have a packet loss issue. To sum up, IDS performance can be affected by the number of multiple flows as well as the high throughput. As a result, system resources are completely exhausted, and there is no way to handle new requests. To address these problems, we provide some possible solutions. First, we suggest a load balancing mechanism, where multiple flows from a high-speed network can be distributed to a collection of IDS instances, each one processing 13000 flows (about 2 Gb/s network traffic) per second. Second, we suggest using efficient regular expression algorithms for reducing the cost of matching the packet payload with the predefined rule set. Moreover, we highlight the importance of enabling the Data Plane Development Kit (DPDK) as a new capturing mechanism; we provide some data to prove that DPDK can significantly increase traffic throughput while incurring a lower performance cost in Section 5.1.

In our study, we did not consider Zeek[3] due to the following reasons. First, Zeek only supports Libpcap and PF_RING; however, due to the limitation of our experimental environment, we do not have a PF_RING module installed on our test servers. Second, from the previous studies [3, 24], we learned that the multithreaded Suricata is better than a single-threaded Snort while processing a larger volume of traffic. However, with Snort3.0, a multithreading framework has been enabled. Therefore, we want to compare Snort and Suricata's performance again and check if Snort's performance improved with multiple threads. Third, after we finished the experiment, we found that Suricata released the latest version 4.1.4. In this version, Suricata includes extended BSD Packet Filter (eBPF) and XDP support. With this new feature, Suricata can directly execute in kernel context, before the kernel itself touches the packet data, which enables the packet capture processing at the earliest possible point after a packet is received from the hardware. Leblond [25] finds a decrease in the packet drop rate after enabling eBPF and XDP in Suricata 4.1.4. Their experiment results motivated us to use the new Suricata 4.1.4, and then repeat our previous experiments. As we observed before, Suricata began to drop packets when there is no enough time for CPU cores to process SoftIRQ from the NIC. By enabling eBPF and XDP in Suricata 4.1.4, we can reach 79.4 Gb/s throughput, but at the same time, we observe that Suricata dropped 0.81% of total packets.

Inspired by [25], we increased the number of Suricata threads, binding them to specific cores to avoid overloading cores handling SoftIRQ from the NIC. As shown in Figure 8 and Figure 9, we found the performance of Suricata is CPU intensive. Figure 8 shows Suricata processed 100 Gb/s traffic when we only

[3] https://www.zeek.org

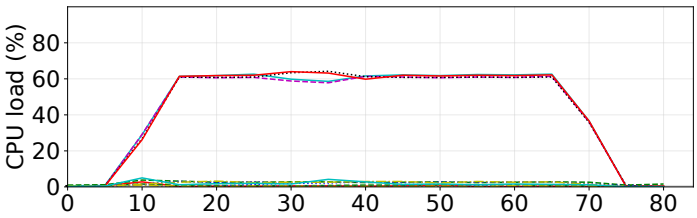


Figure 8: Comparing Suricata's CPU usage with SoftIRQ's CPU usage when enabling one detection rule from the Suricata configure file. Suricata can monitor 100 Gb/s traffic with 0% drop rate, also the CPU of Suricata was stable at 60% while processing a single rule file. Each coloured line represents the CPU usage of a core.

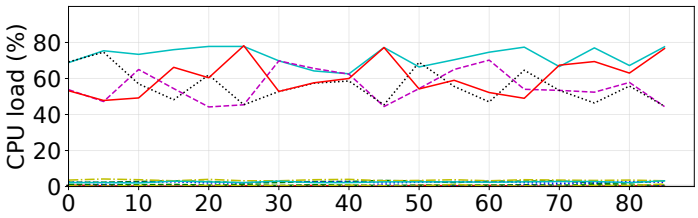


Figure 9: Comparing Suricata's CPU usage with SoftIRQ's CPU usage when enabling all detection rules in the Suricata configure file. When the Suricata's CPU reached 80%, the network throughput dropped to 89 Gb/s along with 62% traffic dropped by Suricata. The CPU usage was unstable between Suricata and SoftIRQ, thus requiring Suricata to use more CPU resources to process all the rules. Each coloured line represents the CPU usage of a core.

enabled a signature rule. In contrast, after used all the signatures in the configure file, the throughput dropped to 89 Gb/s and discarded 62% packets. The reason for the high CPU usage and packet drop rate is that each Suricata instance is processing each packet against 300000 different signatures. The existing optimisations cannot allow IDSs to handle 100 Gb/s networks. The CPU gets overloaded and there are not enough CPU cycles to process new incoming packets. We also observed a significant packet drop when both IDS's processes and SoftIRQ are handled by the same CPU core. On the receiver machine, we allocated half of the CPU cores (*i.e.,* 12 cores) for the IDS processes and another 12 cores for SoftIRQ. As shown in Figure 9, when all detection rules were enabled, it causes high CPU utilisation for each core to load and analyse each rule. While there is no enough time for the CPU cores to process SoftIRQ, the packets from the NIC cannot be handled and dropped before delivered to the process. As a result, we observe 62% of packet drop rate with 80 Gb/s traffic. To sum up our observation, we find that Suricata can not properly handle 100 Gb/s traffic by enabling all the detection rules. Second, the CPU overload will impact the IDS packet drop rate. To resolve the performance issue, we suggest to upgrade the hardware by adding more cores. As a result, we will have more CPU cores to allocate processes for Suricata, SoftIRQ, and NIC.

5.1. Recommendations

Our objective in this study is to find out how to improve performance of current open-source IDSs using recently developed techniques including data processing approaches and packet capturing mechanisms. In this section, we make three

recommendations that might help IDS developers as well as the system administrators in deploying Suricata and Snort in high-speed networks.

Data Distribution. We discovered that 5G multiple flow traffic is too much for a single IDS instance to handle. To this end, we can divide the 5G traffic into smaller pieces, each of which can be handled by a single IDS instance. In the case of using a Software-Defined Networking (SDN) and an IDS cluster, network administrators can easily filter out the traffic based on the network protocols, source, and destination addresses, port numbers, and then pass on the traffic, which can effectively be processed by a single IDS instance, such as 2 Gb/s traffic for each instance. The IDS cluster contains dozens or hundreds of IDS instances, each instance analysing a fraction of the overall traffic volume. To achieve this, OpenFlow [26] can extract traffic based on a predefined network protocol, while at the same time, network administrators can scale the IDS instances based on the traffic volume. Another approach to reducing the volume of data could be checking particular flows, for example, only assessing TCP, UDP, or HTTP flows.

Regular Expression Matching Algorithm. For a signature-based IDS, a regular expression matching algorithm is widely used for identifying application protocols and detecting network attacks. However, a major bottleneck in the existing algorithm is that most IDSs inspect each byte from incoming packets. This causes high CPU usage and memory consumption. Based on our results, besides Becchi *et al.*'s [18] discovery, the current processors are not powerful to match regular expression at 10 Gb/s or more. In order to capture network traffic on a 100 Gb/s link, several works [19, 7, 6, 8] proposed a hardware acceleration solution. For instance, Matoušek *et al.* [19] used the multi-striding technique and pipelined finite state machines in hardware to allow the existing IDS architecture to handle hundreds of Gb/s. In their solution, the pipelined automata directly mapped to the Field Programmable Gate Arrays (FPGAs). Their results show that increasing the number of automata in the pipeline can improve the packet capture speed to 100 Gb/s. Besides, they used a single input packet buffer to reduce memory consumption. Another study [7] addressed the performance bottleneck by optimising the throughput of Deterministic Finite Automata (DFA). Yang *et al.* [7] proposed an ultrahigh-throughput DFA accelerated architecture that brings all advantages from three FPGA-based algorithms: Simple State Merge Tree (SSMT), Distribute Data in Round-Robin (DDRR), and multi-path speculation. In order to reduce memory consumption by the DFA transition table, they used a classical compression algorithm to compress the table. The results showed that in most cases the memory usage of each rule set is less than 15% of the total resources as well as improving Bro's processing speed to handle 100 Gb/s throughput. These studies [19, 7] indicate that by changing regular expression matching engines and combining some accelerated hardware, existing IDSs can efficiently handle 100 Gb/s throughput while maintaining the system resource efficiency.

Packet Capturing Mechanism. DPDK creates a set of data plane libraries and network interface controller drivers for providing efficient ways to handle packets in the user space. DPDK allows userland applications to access packets directly from the NIC, avoiding existing network protocol stacks in the OS. For packet processing applications that do not need to rely on the existing network stack, DPDK minimises processing resources required to access packets.

As Wu *et al.* [27] reported in their study, it is possible to accelerate the packet processing in 100 Gb/s with almost no packet loss. The study shows that with 1500 bytes UDP packets can be processed at 8Mpps (*i.e.,* 90 Gb/s) without packet loss, with a maximum of 24 GB memory and six 3 Gb/s CPU cores. Although this may throttle down to 70% drop rate with packets smaller than 64 bytes due to an increase in the number of packets and thus the processing overhead, it may still provide significant benefits to any packet processing system, including IDSs. Furthermore, XDP [28] can improve the packet processing within the GNU/Linux kernel up to 24 Mpps per each 3.6 GHz CPU core.

In an IDS, it will require much more processing power to analyse packets with traditional signature-based detecting mechanism. There will be an overhead when using XDP which has to copy packets to the user space and do context switch as the IDS runs in user space most of the time while XDP uses eBPF in kernel space. However, optimising the packet capturing mechanism will still provide an initial step to significant performance improvement and multithreading the packet processing will further improve IDS performance.

6. Conclusions and Future Work

In this work, we evaluated the feasibility of using IDSs in high-speed networks by analysing the performance of two popular IDSs using up to 100 Gb/s links. The experiment results from Section 4 show that the multithreaded architecture can significantly improve IDS performance as well as reduce the packet drop rate. Further, both IDSs show better performance when processing traffic under 60 Gb/s. We noticed some packets have been dropped when we configured the throughput to 60 Gb/s. In terms of accuracy, IDSs show a high accuracy even if some packets are dropped. Also, we found that IDSs and the receiver cannot run in parallel on the same server, because it will cause the system's SoftIRQ to get overloaded. Once SoftIRQ is exhausted, the receiver side starts to drop the packets. As a result, we cannot generate traffic up to 100 Gb/s. Furthermore, the performance becomes worse if we start to increase the number of flows per second. Our findings show that Snort and Suricata are not able to handle network throughput higher than 5 Gb/s, which reflects 30000 multiple flows per second. All packets have been dropped when the resource is overloaded. We highlighted some solutions to optimise the resource overhead, reduce the packet drop rate, and improve the detection accuracy. For example, we suggest to add a load balancing mechanism in the existing IDS infrastructure, where multiple flows from a high-speed network can be distributed to a collection of IDS instances, each one monitors 2 Gb/s network traffic that indicates 13000 multiple flows per second. Moreover, we highlight the

importance of enabling DPDK as a new capturing mechanism; DPDK creates a set of data plane libraries and network interface controller drivers for providing efficient ways to handle packets in the user space. Wu *et al.* [27] reported in their study that it is possible to accelerate the packet processing in 100 Gb/s with almost no packet loss.

With this study, we explore some new topics that can be investigated in the future. First, we would like to study the effect of flow duration on IDS performance. For example, we configure a different number of long-lived flows or a different number of short-lived flows in our experiment and observe the changes in IDSs from the performance perspective. Second, we can investigate SDN techniques such as the use of SDN switches for distributing the traffic based on the predefined network protocol that will help in better monitoring of each IDS instance and reduce overheads. Third, we will configure the network interface to use the PF_RING library, and then to repeat the same test scenarios. We then can compare the performance among Snort, Zeek, and Suricata and check any performance improvement by using PF_RING. Last but not least, we also try to find out why the IDS report does not truly reflect the network packet drop status.

References

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

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

[3] Q. Hu, M. R. Asghar, N. Brownlee, Evaluating network intrusion detection systems for high-speed networks, in: Telecommunication Networks and Applications Conference (ITNAC), 2017 27th International, IEEE, 2017, pp. 1–6.

[4] S. Campbell, J. Lee, Intrusion detection at 100g, in: State of the Practice Reports, ACM, 2011, p. 14.

[5] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, Generating realistic workloads for network intrusion detection systems, ACM SIGSOFT Software Engineering Notes 29 (1) (2004) 207–215.

[6] R. Sidhu, V. K. Prasanna, Fast regular expression matching using FPGAs, in: Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on, IEEE, 2001, pp. 227–238.

[7] J. Yang, L. Jiang, X. Bai, H. Peng, Q. Dai, A high-performance Round-Robin regular expression matching architecture based on FPGA, in: 2018 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2018, pp. 1–7.

[8] C. R. Clark, D. E. Schimmel, Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns, in: International Conference on Field Programmable Logic and Applications, Springer, 2003, pp. 956–959.

[9] M. Purzynski, P. Manev, Suricata extreme performance tuning (2016).
URL https://suricon.net/wp-content/uploads/2016/11/SuriCon2016_MichalPurzynski_PeterManev.pdf

[10] J. Johnson, Reproducible performance testing of Suricata on a budget with Trex (2018).
URL https://suricon.net/wp-content/uploads/2019/01/SuriCon2018_Johnson.pdf

[11] L. Schaelicke, J. C. Freeland, Characterizing sources and remedies for packet loss in network intrusion detection systems, in: IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005., IEEE, 2005, pp. 188–196.

[12] T. H. Ptacek, T. N. Newsham, Insertion, evasion, and denial of service: Eluding network intrusion detection, Tech. rep., SECURE NETWORKS INC CALGARY ALBERTA (1998).

[13] A. Samoshkin, Certified snort integrator program (2018).
URL https://www.snort.org/integrators

[14] S. McCanne, V. Jacobson, The BSD packet filter: A new architecture for user-level packet capture, in: USENIX winter, Vol. 46, 1993.

[15] S. Bezborodov, et al., Intrusion detection systems and intrusion prevention system with snort provided by security onion (2016).

[16] MetaFlows, Open source IDS multiprocessing with PF_RING (2016).
URL https://www.metaflows.com/features/pf_ring/

[17] Y. Gong, Q. Liu, X. Shao, C. Pan, H. Jiao, A novel regular expression matching algorithm based on multi-dimensional finite automata, in: High Performance Switching and Routing (HPSR), 2014 IEEE 15th International Conference on, IEEE, 2014, pp. 90–97.

[18] M. Becchi, C. Wiseman, P. Crowley, Evaluating regular expression matching engines on network and general purpose processors, in: Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ACM, 2009, pp. 30–39.

[19] D. Matoušek, J. Kořenek, V. Puš, High-speed regular expression matching with pipelined automata, in: Field-Programmable Technology (FPT), 2016 International Conference on, IEEE, 2016, pp. 93–100.

[20] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–340.

[21] J. E. Hopcroft, R. Motwani, J. D. Ullman, Introduction to automata theory, languages, and computation, Acm Sigact News 32 (1) (2001) 60–65.

[22] J. S. White, T. Fitzsimmons, J. N. Matthews, Quantitative analysis of intrusion detection systems: Snort and Suricata, in: Cyber Sensing 2013, Vol. 8757, International Society for Optics and Photonics, 2013, p. 875704.

[23] J. H. Stammler, Suricata performance white paper (2011).
URL https://redmine.openinfosecfoundation.org/attachments/download/763/suricata%20performance%20writup%20final.pdf

[24] D. Day, B. Burns, A performance analysis of Snort and Suricata network intrusion detection and prevention engines, in: Fifth International Conference on Digital Society, Gosier, Guadeloupe, 2011, pp. 187–192.

[25] E. Leblond, Why eBPF and XDP in Suricata matters (2018).
URL https://suricon.net/wp-content/uploads/2019/01/SuriCon2018_Leblond.pdf

[26] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: Enabling innovation in campus networks, ACM SIGCOMM Computer Communication Review 38 (2) (2008) 69–74.

[27] X. Wu, P. Li, Y. Ran, Y. Luo, Network measurement for 100 gbe network links using multicore processors, Future Generation Computer Systems 79 (2018) 180–189.

[28] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, D. Miller, The eXpress data path: Fast programmable packet processing in the operating system kernel, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, ACM, New York, NY, USA, 2018, pp. 54–66. doi:10.1145/3281411.3281443.
URL http://doi.acm.org/10.1145/3281411.3281443