Towards An SDN Assisted IDS

Robert Sutton[1], Robert Ludwiniak[1], Nikolaos Pitropakis[1], Christos Chrysoulas[1], and Tasos Dagiuklas[2]

[1]School of Computing, Edinburgh Napier University, Edinburgh, United Kingdom
[2]School of Engineering, London South Bank University, London, United Kingdom
Email:*robbo.sutton@gmail.com, {r.ludwiniak,n.pitropakis,c.chrysoulas}@napier.ac.uk, tdagiuklas@lsbu.ac.uk*

***Abstract*—Modern Intrusion Detection Systems are able to identify and check all traffic crossing the network segments that they are only set to monitor. Traditional network infrastructures use static detection mechanisms that check and monitor specific types of malicious traffic. To mitigate this potential waste of resources and improve scalability across an entire network, we propose a methodology which deploys distributed IDS in a Software Defined Network allowing them to be used for specific types of traffic as and when it appears on a network. The core of our work is the creation of an SDN application that takes input from a Snort IDS instances, thus working as a classifier for incoming network traffic with a static ruleset for those classifications. Our application has been tested on a virtualised platform where it performed as planned holding its position for limited use on static and controlled test environments.**

***Index Terms*—SDN, IDS, Network Security**

I. Introduction

Traditional networks have a static architecture which is decentralized and complex. The needs of modern society lead towards networks that have more flexibility and easy troubleshooting. Software Defined Networks (SDN) are the answer to the aforementioned problem, used to centralise the expensive, computationally inefficient elements of traffic routing and forwarding from the conventional practice of having these elements present on every forwarding device in a network [1]. SDN abstracts these processes away from the forwarding devices and handles them by using a network controller to keep track of the network infrastructure and calculate forwarding paths which are then communicated back to the forwarding devices.

In a conventional network, Network Intrusion Detection and Prevention Systems (IDPS) are placed at strategic points in the network over which the majority of the traffic will traverse to offer them the best chance at identifying malicious traffic. IDPS usually add latency to the network traffic and the more complex the inspection of any given packet is, the more time the packet is delayed on its journey. For a large, aggregated network link, which would otherwise seem to be the best place to place such a security mechanism, the latter configuration causes the potential for inefficiency and slowdowns of the traffic along with the risk of missing packets, thus indicating a security issue.

To mitigate this, IDPS devices can often be deployed in a more distributed design, with sensors being placed around the network and reporting security events back to a central sensor or controller. Traffic classified as worthy of inspection based on the usecases of the IDPS can be rerouted away from the normal traffic path and pushed through another IDPS that is tuned specifically to inspect the specific segment of the traffic. In this way, an IDPS can have a very small ruleset and pass traffic in the least possible time as multiple rules are not checked against every packet passing the sensor, thus minizing the overhead. As a modern network has virtualised infrastructure, it may also be possible to make IDPS decisions on individual sensor load and to even dynamically provision new virtual IDPS sensors as required. This is feasible as indications of load being fed to the forwarding tables via traps from IDPS fed back into the controller API. This results in an improvement by placing the sensor at a network bottleneck as means smaller loads of network traffic are being inspected by more IDPS [2].

978-1-6654-4399-9/21/31.00© 2021 $IEEE$

The aim of our work was to design and develop a system that leverages the configurability of an SDN to divert traffic of some interesting nature out to an Intrusion Detection System (IDS) system for inspection. The motivation behind our work is two fold: a) a network segment might be too busy for a single IDS sensor to cope with the load; and b) the majority of traffic over a link is known to be safe and can be discounted, or cannot be inspected and therefore can be manually selected not to be inspected by an IDS. For the shake of our experimentations SNORT [3] has been used, one of the most common and well-supported IDS platforms available. We can also take advantage of the previous work surrounding its performance, capabilities and variations on deployment including in a distributed environment and with use in an SDN.

The contributions of our work can be summarised as follows:

- The creation of a testbed network using an SDN vSwitch, a controller for the same, one or more IDS and a device to run SNORT.
- The forwarding of suspicious data into an SDN controller that itself then pushes data to specific points on an SDN for inspection.
- The evaluation of the performance levels of the SDN, and the IDS being used to inspect traffic.

The rest of the paper is organised as follows: Section II briefly describes the related literature, while Section III analyses our experimental setup. Section IV describes the implementation of our methodology and provides the evaluation of our experimental setup, and the applied mechanisms and technologies.

Finally, Section V draws the conclusions giving some pointers for future work.

II. Literature Review

A. SDN and Attacks

Hakiri et al. [4] provide an excellent overview of SDN and its present challenges when it comes to networking. In [2], Ibrahim et al. review several other works in SDN and IDS. Giotis et al [5] note that related literature seems to concentrate heavily on DDoS attacks and their detection and mitigatio while pointing out the control plane overloading as part of these detection mechanisms. Papadogiannakis et al. [6] suggest that there is sufficient time for a detection mechanism to be successful while alerting other system. However, this seems too slow to prevent certain attacks which have short lifespan, resulting in late detection of a threat which will have already been propagated.

Jantila et al. [7] take the findings of the previous works one step further by discussing an SDN classification application which prevents DDoS against a webserver. They score the interactions of the webserver and feed this information back through an SDN controller to the flow tables of the switching fabric. However, the specific methodology as it is web based it produces delays not meeting real world needs. Shin et al. [8] detail an application development framework, namely FRESCO for security services within an SDN. They note that there is limited functionality within an SDN for embedded security functions and attempt to mitigate that by adding state databases, providing the framework with support for security modules and interaction externally via an API, and enhancing what they believe is a very simplistic approach for feeding back responses to the SDN.

AlEroud et al. [9] focus on attacks against the SDN itself instead of attacks flowing over the SDN. Their work investigates the limits of an SDN when only inspecting the packet headers while achieving good detection rates over attack traffic. Xing et al [10] focus on measuring the performance of an IDS sited away from a controller, with the end goal of providing mitigation from the IDS to the SDN. Their work investigates mitigations such as blocking, quarantine and redirection to a deep packet inspection device, but solely on a cost to the SDN basis. Chung et al. [11] have a setup in their IDS devices which is being transparently placed in front of tenants in a cloud environment. Their traffic classification is performed on traffic to and from tenants and from untrusted networks. If necessary the traffic can be inspected fully by a transparent IDS in its path or subject to other countermeasures as appropriate to the classification.

B. IDS

Ajaeiya et al. [12] propose a solution with an IDS integrated within an SDN controller where they classify flows and apply coarse entries into the flow-tables based on these classifications. From another point of view Sagala et al. [13] attempt to integrate an honeypot setup alongside an IDS where the honeypot is feeding a ruleset to the IDS based on the behaviour observed upon itself. In specific they create a system that watches logs coming from the honeypot, thus generating Snort rules from them and performing checks.

The authors of [14] present a system which is a self-contained Snort IDS, IPS and database for historic traffic and alerts. Their work use dynamic rule creation and redirection to honeypot servers. Kirill Shipulin [15] discusses some techniques for IDS bypassing and specifically for overloading an IDS CPU with crafted requests against poorly designed signature rules. The work by Yuan et al. [16] describes a system to load-balance traffic across a distributed Network Intrusion System (NIDS) infrastructure in response to network load and the load of individual IDS components. The authors argue that a single IDS on a large and busy network segment can be vulnerable to packet loss and as a result of that some form of distributed IDS setup is imperative.

Vallentin et al. [17] propose a system comprised of a commodity grade array of Bro (now Zeek) sensors [18] which are used to share the load of monitoring of a busy network. This system uses some SDN-like features to achieve its task. As an example the authors created a hashing algorithm to determine to which of the Bro sensors a traffic flow should be send, as a result of the hash, thus the destination MAC of the packets is rewritten and sent to the chosen sensor. Another interesting element of this work is the presence of the orchestration element that is responsible for starting and stopping sensor nodes as well as monitoring for failures and bringing backup sensors into function. From another point of view Pitropakis et al. [19] gather intelligence from multiple sensors and and correlate it with open source intelligence to face more advanced threats.

C. Snort Performance

Alhomoud et al. [20] compare Snort and Suricata over several levels of traffic load on two different host operating systems and one virtualised environment. The work shows a clear advantage of Snort over Suricata on almost all of the criteria selected for comparison. In [21], the authors discuss the issue of categorising Snort rules into a useful set for specific administrators in large environments. The authors argue that the current classification schemes as provided by the default Snort installation are poor, lacking detail and formalisation, thus making it difficult to find an appropriate set of rules tuned for a test or environment.

More recently, Bul'ajoul et al. [22] studied the performance of the Snort IDS and this performances variation to the addition of Quality of Service (QoS) on the legacy network equipment that is feeding a Snort sensor. They found that the switch fed the sensor traffic as fast as the forwarding would allow and this led to Snort dropping huge amounts of packets in the rush to queue and process them. To partially address this issue the authors experimented with multiple Snort sensors connected to a switch and forwarded traffic to each based on ACLs (Access Control Lists) within the switch. Their experimentations proved that by using one sensor each for UDP, TCP and ICMP traffic, they were able to bring down the processing time to a third of that with a single sensor. From a similar point of view, Bechtolsheim et al. [23]

leverage similar ideas but move the network infrastructure away from legacy network equipment and ideally away from using slower features of switches such as QoS and ACLs. Papadogiannakis et al. [6] propose an interesting take on improving the performance of an IDS by selectively either ignoring or discarding packets from possible inspection. They state that most security incidents are going to be detected in the initial few thousand bytes of a network flow and that it is unwise and possibly damaging to the efficiency of an IDS to spend inspection time on flows that last for much longer than this, giving VoIP, streaming, P2P traffic and file transfers as examples. Vasiliadis et al. [24] use the CUDA architecture for executing Snort, naming their system Gnort. They managed to transfer large volume of the computational overhead to the GPU, thus speeding up the efficiency of the NIDS, reducing at the same time the overhead on the CPU.

Our work differs from all the previous approaches because we are aware of the attacks a network under inspection may be subject to as signatures may already be in place on one or more Snort sensors to provide the detection for the attacks. The QoS element is out of the scope of our work and instead we focus on the distribution of the detection overhead over more sensors rather than letting one sensor take as long as it needs to complete the task. We do not send data to a Snort sensor but the data are mirrored off to one or more sensors as appropriate aiming for the native acceleration of the traffic throughput. Additionally, the goal of our work is to distribute the detection over more sensors rather than letting one sensor do it as it needs more time to complete the task. We are also making a diversion from the ACL approach by adding flows into the forwarding table per flow on a controller, rather than per frame with an ACL. For larger flows this should represent a time and processing saving.

III. Experimental Setup

Before explaining our experimental setup in detail, it is considered necessary to discuss the use of two tools that had an important role. The first one is *PySnorter* a tool for routing traffic to dedicated Snort instances which is written in Python 3. Its functionality can be summarized as: a) Monitoring an alert file from a Snort instance behaving as a classifier; b) Pulling in information from the alert; and c) Using the information in the alert to generate a REST API command that is then sent to the controller. The aim of PySnorter is to have a very generic Snort instance always listening for suspicious traffic, which then generates alarms capable of sending off the traffic to other Snort instances where the ruleset is specific to it. The latter functionality is hoped to reduce the amount of rules that are checked against traffic for which there is no chance of a match, thus reducing the likelihood of missed packets in the IDS process and accelerating the traffic inspection.

The expectation for any production environment will vary based on the exact requirements so it is rather uncertain whether a fully automated system of applying rules to Snort instances is useful or desirable. For the shake of our experiments, it is useful if the volume of rules could be broken down to be checked based on the nature of the flows that are observed coming into the switch or to the destination servers. To resolve this issue another Python 3 program has been developed to take a parsed list of destination ports against the rule files theyre specified in. Soon afterwards these files would be selected and added them into a Snorts configuration file for inclusion in the rules the traffic was checked against. After careful consideration we decided that a reasonable criteria for choosing a specific Snort instance to work on a set of traffic with a set of rules should be the similiarity of the destination. This is broadly in line with choosing rules that would only be applicable to the services being targeted of interest to an administrator or analyst.

Test Network and Server Infrastructure: Our platform of choice was a GNS3-hosted infrastructure. Because of the selected platform, VMs prepared for the controller, Snort sensors, generator and receiver machines could be built and tested independently and then imported into the GNS3 project containing an OVS vSwitch (with management features) docker container running on the GNS3 VM. This should then persist over many iterations of testing. The network has been set out in a very basic layout as shown in Fig. 1. This configuration allows traffic to be sent from the source host to the destination host via the vSwitch. All the devices on the network also have a back- channel connection to a NAT cloud to allow for device updates and traffic that has no requirement to be routed through the SDN. This is an effective managed network for the virtual hosts. The only traffic that will be traversing the vSwitch will be the test traffic. API calls to the controller and controller management and GUI will be sent on the management network.

Controller and flow manipulation: During the initialization of the testbed, particualar emphasis has been given to directing traffic from the generator machine to the receiver machine via the vSwitch. This is an easy step, because by default the GNS3 vSwitch works as a standard Layer 2 switch even without being connected to an external controller (Ryu) [25]. The GNS3 vSwitch container allows connections to a controller or a controller network via one of the virtual ports. In our case, a VM running the OF controller has been directly connected to this port. The vSwitch was configured with the IP address of the controller and once the controller service was starting, then the control of the switchs features was handed over to the controller. Although Ryu provides a lot of functionalities related to our work, it does not focus on separating traffic amongst Snort instances.

Traffic is passed freely amongst the devices, only limited by the basic network constraints of the devices such as MTU/MSS and the virtual throughput of the vSwitch and the hosts performing the generator and receiver roles. To begin classifying and redirecting traffic to IDS devices relevant to the monitored traffic, it should be mirrored off to an initial IDS. This required a manual flow to be added to the switch.

Snort Triage Device: The Snort process has been configured through a file to both point at the relevant rules file and also output the alerting in a format that is compatible

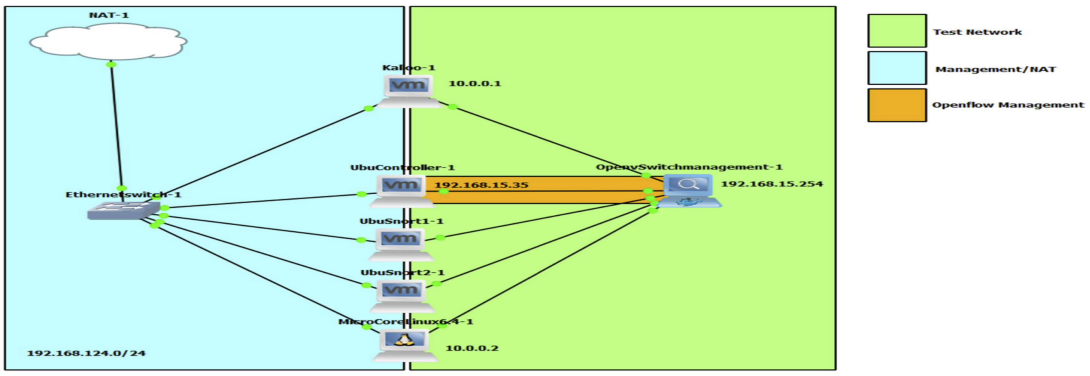

Fig. 1. Initial network setup for POC

TABLE I
PCAP FILES FROM CONSECUTIVE NMAPS

Filename	Nmap Duration (s)	Filesize (kB)	kB/sec
nmapRun1	342.14	53058	155.08
nmapRun2	347.76	52903	152.13
nmapRun3	347.36	52936	152.40
nmapRun4	350.50	55050	157.06

with what pySnorter parser. From this point on the pySnorter software has been constantly watching for new elements being added to the alerts file specified in the Snort.conf file. If an alert has been triggered, a new REST request has been created within pySnorter using the fields passed in the Snort alert to direct that category of traffic to another port on the vSwitch. There, a Snort instance with rules specific to issues and vulnerabilities associated with TCP port 445 (or SMB/CIFS) is ready and waiting to begin more intensive inspection on traffic that has a closer match to its ruleset. This proves that subsets of interesting network traffic can be arbitrarily passed to destinations on the SDN for inspection by network security devices. In this case, this is a Snort instance.

IV. Implementation and Evaluation

The performance measurements for different variations of passing traffic though Snort instances were initiated by selecting a fixed packet of data between two hosts that could be repeated over and again to make sure that the traffic sent was useful for the purposes of tracking differences in the infrastructure surrounding the endpoints. First we selected the nmap tool against our target which is vulnerable to metasploit. For the initial runs of this test, traffic has been captured by the host at the receiving end of the conversation to allow to both check that the behaviour is consistent over the runs and to allow analysis of the resultant traffic to produce some per-Snort rulesets in later tests or future work. The tests has been configured to also generate a PCAP for each run. It is interesting to note the differences in capture sizes over these runs as depicted in Table I.

There is not a direct correlation between the time taken to perform the scan either way. From the tests, it can be observed there is a variation of up to eight (8) seconds for the scan time and 2MB for the captures. All these tests have produced with an SDN switch in between the hosts and no attached controller. The SDN device runs within a docker container on the GNS3 VM which also handles the links between the two.

For the next series of runs, it has been decided to also add in the Ryu controller to manage the switch. The Ryu controller was initiated and four more runs of the tests have been executed. The first test had to be abandoned at 20 minutes when the nmap process on the attacking machine reaches 100%. Other than that, the other three runs are broadly similar to the previous runs without the controller.

According to our initial assumption the pySnorter tool is able to have as input a Snort alert file and divert traffic accordingly to one or more other network endpoints. In our case, it is a Snort sensor. pySnorter in its present format depends on quite a lot of manual editing when being moved from environment to environment and when the features of the infrastructure change, such as the address of the SDN controller and the makeup of the vSwitch depending on the port numbering and the switch and bridge IDs that the REST requests are then aimed at.

There is also the issue of the timeout period in the wait between checking for additional lines added to the alerts file from the Snort process. This has been originally set to a tenth of a second, but in a higher throughput network, this reading might be inadequate. In later experiments the timing has been changed to a hundredth of a second, but again, there exist networks for which several alerts could conceivably be missed within this time period.

Our tests have been able to prove that pySnorter is capable of splitting traffic over several IDS sensors. During the implementation of our work lots of nmap, nping and hping have been performed across a vSwitch with and without a controller combined with and without Snort hosts monitoring diverted traffic. In every experimentation within the limits of the infrastructure it was proved that Snort did not face any type of failure in its primary task of monitoring all traffic passing over the wire.

As previously discussed we made use of GNS3, hosting its own containers within the GNS3 VM as well as VMWare Workstation VMs built and running in the machine running GNS3. GNS3 proved to be a good choice, as there was a

pre-built vSwitch container within GNS3 and GNS3 handled much of the network inter connectivity between the devices. However, GNS3 was responsible for lots of troubleshooting during the process of connecting it with the VM Workstation network (ESXi). Consequently, we made a decision to change the Snort instances to multiple Snort processes running on a single machine, but taking care to each monitor a single interface and to be pinned to a separate CPU thread. As it turned out, we managed to reduce the overhead for an analyst and infrastructure engineer by having to only maintain a single machine for varied Snort processing.

V. Conclusions And Future Work

We have made an attempt to develop a system that takes advantage of one of the strong points of SDN, the ease to be configured, to divert suspicious traffic to an IDS system for inspection. A virtualized network has been setup where Snort has been used as our main IDS. Eventually instead of different instances of Snort, we have made use of multiple Snort processes running concurrently on the same infrastructure and diverted the traffic depending on its nature.

Our main tool pySnorter has managed to meet our needs by diverting traffic between the Snort sensors. However, it could be enhanced and grown to do more than it does at the present, which is scanning for alterations to an alert file and generate REST requests to a controller from the content of the alert outputs. It would be possible to change its functionality, thus making it able to communicate to several SDN devices instead of just a single. It could also mitigate a known attack as well as just monitor for them; a flow identified by one of the tiered Snort instances in these tests could have a set of rules marked up in such a manner to generate a flow to be added with drop actions on the SDN device(s) closest to the source of the newly classified malign traffic.

Our future work includes the addition of a large internet-facing links along with a series of small gigabit capable devices to be used as Snort sensors and a larger SDN (with more than 5 physical ports) that would allow to slice up incoming traffic into much more specific slices that could have customised Snort instances waiting to inspect traffic. At this point the entire testing setup could be rerun and this would generate more interesting performance testing results.

Also it would be interesting to repeat the work emphasising on Snort because the SDN parts were really a small subset of our testbed and there are opportunities for way more experimentation to bee done in regarding the performance of Snort and other IDS. Additionally, we want to take advantage of solutions such as Vasiliadis et al. [24], where transferring the overhead of any IDS and the SDN controllers to the GPU would greatly benefit the future IT infrastructures.

References

[1] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in sdn-openflow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.

[2] J. Ibrahim, "Sdn-based intrusion detection system literature review," 2017.

[3] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks." in *Lisa*, vol. 99, no. 1, 1999, pp. 229–238.

[4] A. Hakiri, A. Gokhale, P. Berthou, D. C. Schmidt, and T. Gayraud, "Software-defined networking: Challenges and research opportunities for future internet," *Computer Networks*, vol. 75, pp. 453–471, 2014.

[5] K. Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments," *Computer Networks*, vol. 62, pp. 122–136, 2014.

[6] A. Papadogiannakis, M. Polychronakis, and E. P. Markatos, "Improving the accuracy of network intrusion detection systems under load using selective packet discarding," in *Proceedings of the third European workshop on system security*, 2010, pp. 15–21.

[7] S. Jantila and K. Chaipah, "A security analysis of a hybrid mechanism to defend ddos attacks in sdn," *Procedia Computer Science*, vol. 86, pp. 437–440, 2016.

[8] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, M. Tyson *et al.*, "Fresco: Modular composable security services for software-defined networks," in *20th Annual Network & Distributed System Security Symposium*. Ndss, 2013.

[9] A. AlEroud and I. Alsmadi, "Identifying cyber-attacks on software defined networks: An inference-based intrusion detection approach," *Journal of Network and Computer Applications*, vol. 80, pp. 152–164, 2017.

[10] T. Xing, D. Huang, L. Xu, C.-J. Chung, and P. Khatkar, "Snortflow: A openflow-based intrusion prevention system in cloud environment," in *2013 second GENI research and educational experiment workshop*. IEEE, 2013, pp. 89–92.

[11] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "Nice: Network intrusion detection and countermeasure selection in virtual network systems," *IEEE transactions on dependable and secure computing*, vol. 10, no. 4, pp. 198–211, 2013.

[12] G. A. Ajaeiya, N. Adalian, I. H. Elhajj, A. Kayssi, and A. Chehab, "Flow-based intrusion detection system for sdn," in *2017 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2017, pp. 787–793.

[13] A. Sagala, "Automatic snort ids rule generation based on honeypot log," in *2015 7th International Conference on Information Technology and Electrical Engineering (ICITEE)*. IEEE, 2015, pp. 576–580.

[14] G. Ahmed, M. N. A. Khan, and M. S. Bashir, "A linux-based idps using snort," *Computer Fraud & Security*, vol. 2015, no. 8, pp. 13–18, 2015.

[15] K. Shipulin, "We need to talk about ids signatures," *Network Security*, vol. 2018, no. 3, pp. 8–13, 2018.

[16] W. Yuan, J. Tan, and P. D. Le, "Distributed snort network intrusion detection system with load balancing approach," in *Proceedings of the International Conference on Security and Management (SAM)*. The Steering Committee of The World Congress in Computer Science, Computer , 2013, p. 1.

[17] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney, "The nids cluster: Scalable, stateful network intrusion detection on commodity hardware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 107–126.

[18] D. Andrews, J. Behn, D. Jaksha, J. Seo, M. Schneider, J. Yoon, S. J. Matthews, R. Agrawal, and A. S. Mentis, "Exploring rnns for analyzing zeek http data," in *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, 2019, pp. 1–2.

[19] N. Pitropakis, E. Panaousis, A. Giannakoulias, G. Kalpakis, R. D. Rodriguez, and P. Sarigiannidis, "An enhanced cyber attack attribution framework," in *International Conference on Trust and Privacy in Digital Business*. Springer, 2018, pp. 213–228.

[20] A. Alhomoud, R. Munir, J. P. Disso, I. Awan, and A. Al-Dhelaan, "Performance evaluation study of intrusion detection systems," *Procedia Computer Science*, vol. 5, pp. 173–180, 2011.

[21] L. Etienne, "Malicious traffic detection in local networks with snort," Tech. Rep., 2009.

[22] W. Bul'ajoul, A. James, and M. Pannu, "Improving network intrusion detection system performance through quality of service configuration and parallel technology," *Journal of Computer and System Sciences*, vol. 81, no. 6, pp. 981–999, 2015.

[23] A. V. Bechtolsheim and D. R. Cheriton, "Access control list processing in hardware," Apr. 23 2002, uS Patent 6,377,577.

[24] G. Vasiliadis, M. Polychronakis, and S. Ioannidis, "Parallelization and characterization of pattern matching using gpus," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2011, pp. 216–225.

[25] Ryu, "Simple Switch SNORT," 2014.