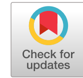
Future Generation Computer Systems 124 (2021) 254–267

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

A dynamic and scalable parallel Network Intrusion Detection System using intelligent rule ordering and Network Function Virtualization



Hårek Haugerud [a,*], Huy Nhut Tran [a,b], Nadjib Aitsaadi [c], Anis Yazidi [a,d]

[a] Department of Computer Science, OsloMet – Oslo Metropolitan University, P.O. Box 4 St. Olavs plass, N-0130 Oslo, Norway
[b] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway
[c] Universités Paris-Saclay, UVSQ, DAVID, F-78035, Versailles, France
[d] Faculty of Information Technology and Electrical Engineering, Norwegian University of Science and Technology, NO-7491, Trondheim, Norway

ARTICLE INFO

Article history:
Received 28 October 2020
Received in revised form 8 May 2021
Accepted 27 May 2021
Available online 8 June 2021

Keywords:
Network Intrusion Detection Systems (NIDS)
Elastic architecture
Rule distribution
Network Function Virtualization (NFV)

ABSTRACT

A Network Intrusion Detection System (NIDS) is a fundamental security tool. However, under heavy network traffic, a NIDS might become a bottleneck. In an overloaded state, incoming and outgoing packets in the network might suffer from long delays since previous packets are still being inspected, and eventually the NIDS starts to drop packets when it runs out of hardware resources. Although many solutions have been suggested in the literature to counter this problem, they are not completely reliable as each of them has limitations. This paper investigates the design of a lightweight elastic architecture which allows parallel processing in an existing NIDS while maintaining the filtering integrity. Furthermore, we propose two adaptive algorithms which dynamically adjust and divide the signature rules evenly across NIDS nodes using a node level parallelism method in order to achieve intelligent rule ordering. We test our approaches in real-life settings by implementing a functioning prototype involving different modern networking technologies. The prototype presented is a Network Function Virtualization (NFV) of an intrusion detection system which utilizes Open vSwitch and Docker containers running Snort in order to provide an elastic system. To the best of our knowledge, there has been no work that orchestrates both scaling and rule splitting and re-ordering of IDS signatures as a part of a holistic elastic IDS solution.

The results of this study show that the proposed algorithms are able to equally split the IDS workload and thereby enabling the system to scale by adjusting the number of virtual components which analyse the network traffic. At the same time the experiments indicate that the algorithms can be tuned by a single parameter in order to avoid that some packets go unexamined while simultaneously craving a minimum of the dynamically available computer resources.

© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

A Network Intrusion Detection System (NIDS) is a security tool which is composed of software and/or hardware designed to detect cyber attacks against networks and servers [1,2]. Unlike a packet filtering firewall, which only accesses and filters network packets based on the limited information of the packet headers, a NIDS can identify details of intrusion attempts from the payload of the network packets using Deep Packet Inspection (DPI) and thereby reveal their hidden agenda of compromising the computer systems they are addressed to. A NIDS funnels all network traffic through its sensors to detect intrusions and anomalies. As the network traffic grows, the use of a single NIDS in a network may lead to congestion problems if the network throughput is too large. Deep Packet Inspection might involve extensive pattern matching of network packets against complex attack rule signatures. Pattern matching is an expensive operation, requiring significantly more computer resources than a firewall, and this can overload a NIDS [3,4]. If a NIDS is overloaded and starts dropping or neglecting the content of packets, this can compromise the security of the network. Eventually some intrusions might not be detected as some packets which are part of the same attack might evade the inspection of the NIDS, resulting in an incomplete packet matching.

There are many solutions available when it comes to dealing with large network traffic volumes for NIDS including:

1. Upgrading hardware by installing dedicated packet capture cards and installing more computer resources (CPU, Memory), as well as tuning the NIDS software so it can handle more traffic.

* Corresponding author.
E-mail address: harek.haugerud@oslomet.no (H. Haugerud).

https://doi.org/10.1016/j.future.2021.05.037
0167-739X/© 2021 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

2. Using a cluster of NIDS and distribute both the network traffic and the signature rules across the NIDS nodes.

The first solution consisting of upgrading hardware and tuning the NIDS software is very costly and not scalable. Network bandwidth speed is continuously increasing by a factor of ten every four years [5], and continuously upgrading hardware to operate a NIDS is expensive. Tuning a NIDS to make it handle more traffic is a challenging task with many trade-offs, which can make the NIDS more complicated than intended.

The second solution which resorts to clusters of NIDS is affordable and scalable. The solution can be adapted to the current traffic flow and resources can be released and used for other purposes when the network traffic is low. Many studies [6,7] take the advantage of using low cost computers with NIDS as a cluster to handle the high network traffic load. Moreover one can add additional NIDS instances to the cluster if needed. However, both the way the traffic is distributed across NIDS instances and the way the signature rules are distributed play a very important role to guarantee the success of this solution.

Snort is the de facto standard open source intrusion detection and prevention system [8] and is the tool we base our proposed IDS solution on. A shortcoming of Snort is that the current stable release still is single-threaded, awaiting the Snort 3.0 release, and this makes our approach even more important. There exists multi-threaded IDS solutions like Suricata [9] but still the approach of distributing the workload to completely independent virtualized computational units makes the system even more scalable and flexible. This paper aims to explore and design an architecture that runs NIDS in parallel to reduce the processing time of the pattern matching and reduce the number of packets dropped. This architecture is restricted to NIDS performing rule based intrusion detection making it possible to split the rules into smaller sets which can be processed in parallel which is not necessarily true for other NIDS based on machine learning [10] or other detection principles. The most important key factor is to develop an algorithm that distributes signature rules evenly in an intelligent manner across NIDS sensors in the cluster. Additionally an algorithm for automation of a dynamic scaling of NIDS sensors is developed. The algorithm adapts the number of NIDS sensors in response to the workload.

There is a number of studies that seek to improve the performance of NIDS. The vast majority of these studies focus on splitting up, routing and load balancing the network traffic across different NIDS nodes in the system. In this paper, we rather focus on dividing and distributing signature attack rules across the NIDS nodes. In other words, we adopt rule splitting across different NIDS nodes. The work on rule splitting is extremely scarce and the only work we could find in the literature is due to Jiang et al. [11]. Jiang et al. performed rule splitting over different IDS instances in an offline manner based on a tree like data structure called Leaf Pruning (LP). However, the rules were not profiled and the splitting of the rules based on LP is static. Thus, if the matching time of some rules in the IDS increases due to the dynamics of the traffic, some of the instances might become bottlenecks in terms of matching time. Note also that rule splitting and dynamic scaling entail some orchestration challenges. Our propounded dynamic scaling is responsible for defining communication between snort sensors, as well as defining the processes that trigger the scaling behaviour. It will also ensure that the integrity of snort sensors is maintained in terms of inspection of packets while updating new rules to the snort sensors. Our scaling was achieved by exploiting the virtualization capabilities of containers, thus, we can call our IDS elastic. Traditional approaches for parallel IDS usually use multicore architecture and multi-threading to balance the pattern matching load, and in some cases sophisticated hardware such as GPUs [12–14]. In other words, the absolute vast majority of works on parallel IDS focus on accelerating the pattern operation by distributing the workload of the pattern matching per packet. When it comes to elastic IDS, [15] proposes a reactive approach to adjust the number of IDS instances based on the drop rate. The traffic is captured at each instance for the last 10 s and stored in pcap files, and whenever the drop rate is high, a new instance is launched and the recorded traffic is sent to it. The latter approach is rather simplistic and entails clear overhead and inspection delay.

We summarize the contributions of this paper as follows:

- An innovative design of a parallel NIDS based on Network Function Virtualization and rule distribution is presented.
- Using a modified parallel technique, NL-FP-1, we create a working parallel environment and devise two algorithms which dynamically adjust and divide the signature rules evenly across NIDS nodes.
- Prototype experiments show that the dynamic scaling algorithm at any time adjusts the number of Snort nodes needed in response to the CPU load on the system.

The remainder of this paper is organized as follows. In Section 2 we review the state-of-the-art while focusing on the most notable studies related to our work. Section 3 presents our solution and Section 4 is dedicated to the experimental results. Section 5 gives some closing remarks.

2. Related work

In this section some notable attempts in the literature to boost the performance of NIDS under heavy traffic situations are reviewed.

A framework of parallel techniques in the field of intrusion detection was presented by Wheeler and Fulp [7] to improve the performance of signature based intrusion detection systems. Three parallelization designs were discussed.

The first method is called Node Level Function Parallel 1 (NL-FP-1) where single Snort rules are categorized based on their rule group and evenly distributed across the IDS nodes. A rule group in Snort is a way to put rules in groups based on their source IP, destination IP or port number. For instance, all Snort rules for ssh are placed in rule group port 22, while all web-traffic rules are placed in rule group port 80. Packets are duplicated to all IDS nodes for inspection using a traffic duplicator.

The second method, called NL-FP-2, considers complete rule groups instead of single rules in different rule groups and then distributes rules to all the IDS nodes. For packet distribution one can either duplicate all the packets and send them to all IDS nodes or alternatively use a traffic splitter and route traffic based on source ip, destination ip or port number to the IDS that has these Snort rule groups. E.g all web traffic should be routed to the IDS node that is configured for the Snort web traffic rule group.

The last method is called Node Level Data Parallel 1 (NL-DP-1). According to this method, each IDS node has the complete Snort rule policy, and packets are distributed evenly by a traffic splitter (load balancing). The main disadvantage of this approach is the difficulty of maintaining the session integrity, reducing the ability of Snort to analyse the traffic correctly without the full picture of packet flows. In fact, delay of session integrity can be a bottleneck inhibiting the success of this approach. However, a sophisticated traffic splitter can solve this bottleneck.

Shiri et al. [6] implements the NL-FP-2 approach proposed by Wheeler and Fulp [7] by deploying two IDS nodes in parallel. In their experiments the Snort signature rules are evenly divided between two IDS nodes, so that both IDS nodes have the same number of rules. For testing the authors use a dataset from a TCPDUMP-file generated to compare the processing time

on their parallel implementation versus a single IDS node. Their experiment results show that the processing time of their implementation of two IDS nodes is 42% faster than for a single IDS node.

Recently, it was proposed to extend the idea of a parallel firewall to the cloud [16]. The authors propose combining two elements for boosting the performance of a firewall that are rarely combined (1) rule ordering and (2) rule decomposition into disjoint sets that are deployed over smaller parallel firewalls. The latter study relies on the so-called batch estimator proposed Mohan at a. [17] for estimating the matching probability of rules in a dynamic environment for the sake of rule reordering. Please note that traffic aware ordering [16–18] is an appealing idea that can be adopted in the context of IDS when the number of rules is large. Kopek et al. [3] introduce an improved version of NL-DP-1 [7] called Divided Data Parallel (DDP), which is a parallel content matching approach, where its goal is to reduce the processing time that Snort spends in the content matching stage, which can consume up to 70% of the processing time. In the DDP approach, the payload of a packet is sent to an array of processors set up in parallel and is divided into fragments where the processors inspect parts of the payload. If there is a match with any Snort rule the processor will notify the other processors that the payload of that packet is matched and the other processors will then move on to the next packet. The results show an improvement of content matching where the processing time has a speedup of 1.25 n (where n is the number of processors), while previous work had a speedup of 0.75 n.

Bulajoul et al. [19] experimentally demonstrate the inability of packet handling in NIDS as network traffic increases. The researchers demonstrate how NIDS have the tendency to drop and stop analysing packets as the speed and volume of the network traffic increases. The goal of this paper is to reduce processing time in NIDS, which in return reduces the number of dropped packets. The experiments confirm that when running the Snort NIDS in parallel, the number of packets dropped decreases. The experiment results show that as the number of IDS nodes running in parallel increases, the packet drop rate decreases consequently. With a dataset of 8000 packets; running one IDS node had a packet drop rate of 10.9%. When running two instances of IDS nodes with the same data set the drop rate was decreased to 5.7%.

The work reported in [20] proposes two pattern matching algorithms to reduce memory usage and improve the run time of an IDS. The Aho–Corasick algorithm is used when analysing a packet payload with short patterns, while the Wu–Manber algorithm is used for longer patterns. Both algorithms are multi-pattern matching algorithms, meaning they can inspect multiple packets concurrently if there are enough threads that can be launched. In the experiment the authors ran the algorithm using four threads, where one thread used the Aho-Croasick-algorithm while the other three threads used the Wu–Manber-algorithm. The results show that by using this combination, a reduction of the time spent on pattern matching by 41% compared to a normal configuration is achieved.

Umar et al. [21] propose a model for multithreading and parallelization in IDS. The main focus of this paper is the distribution and update of signature based rules to IDS agents. The authors propose to aggregate all rules in a centralized database (mySQL), while agents have their own database called Small Signature Database (SSD). The rules in the centralized database are divided into smaller portions and distributed to the agents SSD, e.g rule 1–200 is used for agent 1, 201–400 for agent 2 and so on for additional agents. The results of the experiment show that their proposed model is 50% faster when using two IDS agents compared to one. In addition the authors suggest using an algorithm to update the main database with frequent rules that raise alerts on the agents.

Kumar et al. [22] propose a technique to maintain and update databases consisting of the most frequent attack signature rules. By using the Rule Induction Algorithm (RIA) (a modified version of the SRLF algorithm) the solution builds smaller databases which contain only the most frequent attack signatures from the large database. The RIA algorithm also takes care of updating the smaller databases whenever the number of most frequent attacks changes in the large database. The results of the experiment show that when using an IDS where all rules were enabled, the total amount of dropped packets was 5%, while for the proposed model the total amount of dropped packets was reduced to 1% over 50 test sessions.

The study reported in [23] emphasizes how difficult it is for IDS to process packets when the networking speed in a high-speed network infrastructures is increased. Neither optimization of packet matching algorithms used by centralized processors nor custom-developed hardware can cope with these problems because of their inherent stateful nature. Hence, the authors have studied a pattern matching algorithm that works by performing a stateful signature matching and detection of network attacks. A parallel rule matching unit made of four sensors deployed on four Linux boxes performed better than a single sensor used for a particular set of detections on a particular dataset. Other studies in this area are either not fully compliant with a parallel matching algorithm or utilize a central sensor to process the packets. Moreover, the study in question has been built on the observations of similar studies in this area. Hence, it also facilitated the communication between all the four sensors, allowing them to synchronize the state matching and render the transport of each packet sensor-independent. This study was only limited by the limitations of the hardware utilized.

In [24], Schuff et al. discuss two parallelization strategies for the Snort NIDS. However, the solutions presented in this paper can also be applied to other NIDS which involve stream reassembly and flow-tracking. Using the system based on Snort version 2.6RC1, the analysis discovers that the two parallelization strategies show a good performance on the targeted workloads , although they face some limitations. The conservative parallelization manages to yield substantial speedups on 3 of the 5 network packet traces involved, giving as much as a three times speedup on 4 processor cores and processing at speeds up to 1.7 Gbps. However, the optimistic parallelization shows a degrade in the extra overheads performance of 10% for traces which exhibit flow concurrency, therefore, limiting the speedup to 2.8 on four cores. Despite these results, the article also mentions that the optimistic parallelization enables more intra-flow parallelism with one additional trace resulting in a good speedup with traffic rates over 2 Gbps.

Chen et al. [25] focus on a new parallel structure of a NIDS named Para-Snort. The structure is characterized by a clean-cut modular design and highly optimized core algorithms. The importance of this article is set by the innovativeness behind Para-Snort, not only it indicates more flexibility and ease while scaling new modules, but it also shows the ability to replace modules by using hardware acceleration. Furthermore, through comprehensive experimental results, the paper shows that Para-Snort manages to increase the speedup by an order of magnitude between 4 and 6 times for various traces using a 7-thread parallelizing test setup. However, as mentioned by the authors, despite the advantages presented in the article, the processing engine of Para-Snort requires further improvement. All in all, Para-Snort appears to be a Snort parallelization strategy which is designed to achieve a better performance while showing impressive results both for load balancing and for multi-pattern matching.

A white paper by Intel Corporation [26] discusses the performance benefits that result from the use of multiple processing

cores for communications applications while also illustrating the results indicated by the analysis on an open source intrusion detection application. The paper shows that Intel multi-core processors exhibits a significant increase in performance for packet processing applications. Using four execution cores with pipelining and flow-pinning gives only a minimal incremental throughput for 175 concurrent TCP connections. However, for 25,000 TCP connections the resulting throughput was more than 6.2 times greater than for the single core system. The study also indicates that software tuning to increase the L2 cache hit rate beyond the 70% recorded could potentially yield even greater performance improvement.

3. The proposed solution

In this section, we present the overall architecture for distributing network traffic and dividing and updating signature rules on Snort sensors. Network Function Virtualization (NFV) is a new paradigm in the move towards open software and network hardware [27–29]. In our work, we adopt some of these principles and use Open vSwitch and Docker containers running Snort in order to make an elastic virtualized IDS which is flexible and easily scalable. Software-defined Networking (SDN) is another technology which has been increasingly used in virtualized security scenarios [30]. Thus, our solution could seamlessly be integrated within an infrastructure using SDN [31].

The proposed solution is a scalable intrusion detection system, however, the main focus of this paper is on how to scale the solution and avoid packets passing the sensors undetected, and there is much less focus on the detection part. Nevertheless, all the snort sensors in the solution send their alerts to a central alert database so that the prototype is a fully functional IDS. As the traffic is duplicated and sent to several snort sensors, it would not make sense to drop packets which raise an alert and the solution will hence not be a candidate for an intrusion prevention system. For such a system scaling could be obtained by splitting the traffic into several streams which were redirected to a number of sensors, each containing all the rules, and then dropping packets would make sense and could stop an attack.

It is well known that it is challenging to keep the integrity of the alerts when splitting the network traffic [2,32], a design referred to as Node Level Data Parallel 1 (NL-DP-1) [7] in Section 2. In general the state of the connections need to be kept as in some cases rules are triggered by related traffic. This leads to an extra communication overhead between the sensor nodes in order to maintain the integrity of the alerts. In our solution, these problems are avoided by sending all traffic to all of the intrusion nodes. Additionally, we distribute a fraction of the rules to each node so that the sensors do not have to match each packet with every NIDS rule contained in the large set of rules in order to process the rules in parallel.

We adopt the Node Level Function Parallel 1 (NL-FP-1) design [7] with a main modification in terms of division and distribution of signature rules. The parallelization model used in this paper is divided into two main parts; the first part is to distribute Snort signature rules evenly to the IDS sensors in the parallel cluster, while the second part is the control of the network traffic distribution. A model for each part is designed for a better overview.

3.1. Distribution and update of signature rules model

The method for distributing signature rules of the NL-FP-1 technique is to segregate rules based on rule groups. A rule group in Snort permits to place by default rules in the same group based on their source IP, destination IP or port number. For instance, all rules with an incoming port number of 22 (SSH) belong to a rule group.

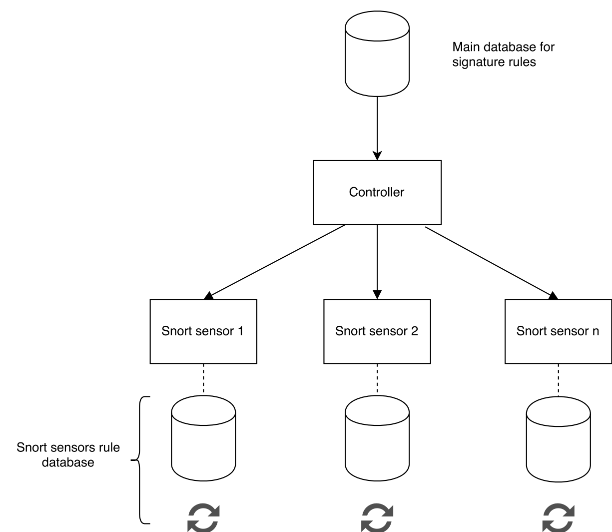


Fig. 1. Model for updating IDS sensors with new rule-sets.

However, a different approach for the distribution of signature rules is implemented in this work, where the rules are distributed evenly based on the performances of each rule with the use of a signature rule ordering algorithm that creates new rule-sets for each IDS sensor. The algorithm is discussed in greater detail in Section 3.3.1.

Whenever a new rule-set is created, these rules have to be distributed to update the old rule-set on every Snort sensor. Fig. 1 shows a simple model of how Snort sensors are updated with new rules in their database. All signature rules are kept in a central rule-set file and when there is a need to create new rule-sets, the controller will use the signature rule ordering algorithm to create new rule-sets for each Snort sensor and update the Snort sensors database with new rules.

3.2. Technical model

The technical model describes the core modules of our system for creating a dynamic scalable parallel environment of NIDS sensors. The technical model resorts to two algorithms; the signature rule ordering algorithm and the dynamic scaling algorithm.

A diagram of the technical model has been designed to show how the scaling algorithm script works, adjusting Snort sensors based on the output of a system load parameter. The diagram of Fig. 2 shows the state of all Snort sensors in the scenario where the controller using both algorithms has monitored a system resource, performed calculations and made a decision that three sensors are needed. The rule ordering algorithm has then created three new rule-sets for each sensor, and the scaling algorithm has distributed the new rule-sets and reloaded each sensor, illustrated by an updating icon for sensor 1, sensor 2, and sensor 3.

The diagram shows only a small part of the technical model, and was designed to give an overview of how the scalable parallel environment works. In the next sections a more detailed insight into how both algorithms work is given to complete the picture of the technical mode.

3.3. Algorithms

In this section, diagrams and pseudo-code will serve as blueprints for how the signature rule ordering and dynamic

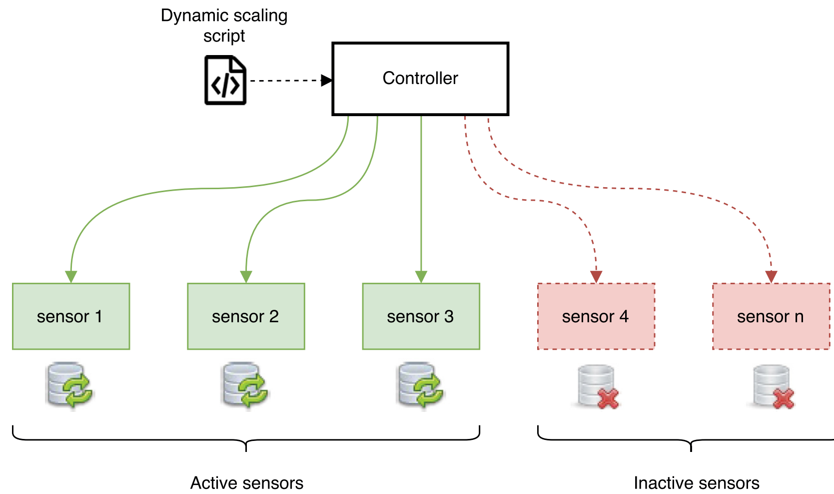


Fig. 2. Parallel architecture model.

scaling algorithms are developed. The signature rule ordering algorithm is integrated with the dynamic scaling algorithm in order to complete the design of the technical model.

3.3.1. The signature rule ordering algorithm

Three procedures interplay to manifest a working signature rule ordering method and they are called **load_profiled_rules()**, **load_rules()** and **order_rules()**. Each of these algorithms is described below. The description is supported by the workflow diagram of Fig. 3 and pseudo-code for each of the three procedures.

The numbers in parenthesis accompanying the procedure names below represent the numbers in the workflow diagram of Fig. 3.

1. **load_profiled_rules()** - (1)
   This component is used to read all rule profiling performance files generated by the Snort sensors to create a dictionary that consists of the signature rule ID (SID) and the amount of time spent on each rule performing packet inspection. In cases of multiple performance profiling files, occurrence of the same signature rule might happen, therefore duplication of signature rules is managed by this component as well. This component finally returns a dictionary of all signature rules storing the time used for packet inspection. Pseudo-code of this procedure is shown in Algorithm 1.

```
def load_profiled_rules (performance_profiling_files): {
profiled_rules_dictionary = {} // empty dictionary

for file in performance_profiling_files do
    open file ;
    for i in file do
        find SID and avg/check parameter and add to
        dictionary ;
    end
end
return profiled_rules_dictionary
}
```

Algorithm 1: Pseudo-code of the load_profiled_rules() procedure of the signature rule ordering algorithm.

2. **load_rules()** - (2)(3)
   The load_rules() component will load the main rule-set (the centralized rule-set that contains all signature rules) and use the dictionary created by load_profiled_rules() to create a list that maps the signature rule ID to the actual content of the rule, which consists of the rule options that define how the rule is constructed. Rules with unknown time measurement and yet to be matched by the detection engine will be appended to a separate list. This procedure returns two lists; where the first list consists of signature rules with a known time measurement while the second list consists of signature rules with unknown time measurement used in packet inspection. For the sake of simplicity, rules with known time measurement are called profiled rules, while rules with unknown time measurement are called unprofiled rules. The pseudo-code of load_rules() is shown in Algorithm 2.

```
def load_rules (Snortrulefile, profiled_rules_dictionary): {
 // use the SID from dictionary and find the
 // corresponding rule in the main Snort rule-set file

profiled_rules, time, unprofiled_rules = [ ]  //empty lists
for rule in Snortrulefile do
    if SID in rule then
        append rule to profile_rules list ;
        append avg/check time to time list ;
    else
        append rule to unprofiled_rules list ;
end
return profiled_rules, time, unprofiled_rules
}
```

Algorithm 2: Pseudo-code of the load_rules() procedure of the signature rule ordering algorithm.

3. **order_rules()** - (4)(5)(6)(7)
   The last and most important component of the signature rule ordering algorithm is the order_rules() procedure. The tasks of this component is to create new rule-sets by dividing profiled rules evenly across the number of rule-sets needed. The profiled rules will first be randomly distributed into a number of rule-sets. The profiled rules in the newly created rule-sets will then be reordered using simulated annealing algorithm based on how much time each rule used in the detection engine during the deep packet inspection. Once the simulated annealing process is finished, the final rule-set distribution is expected to be more evenly balanced than the initial randomized placement. Finally, the unprofiled rules are evenly randomly distributed across the new rule-sets. Pseudo-code of the procedure is shown in Algorithm 3.

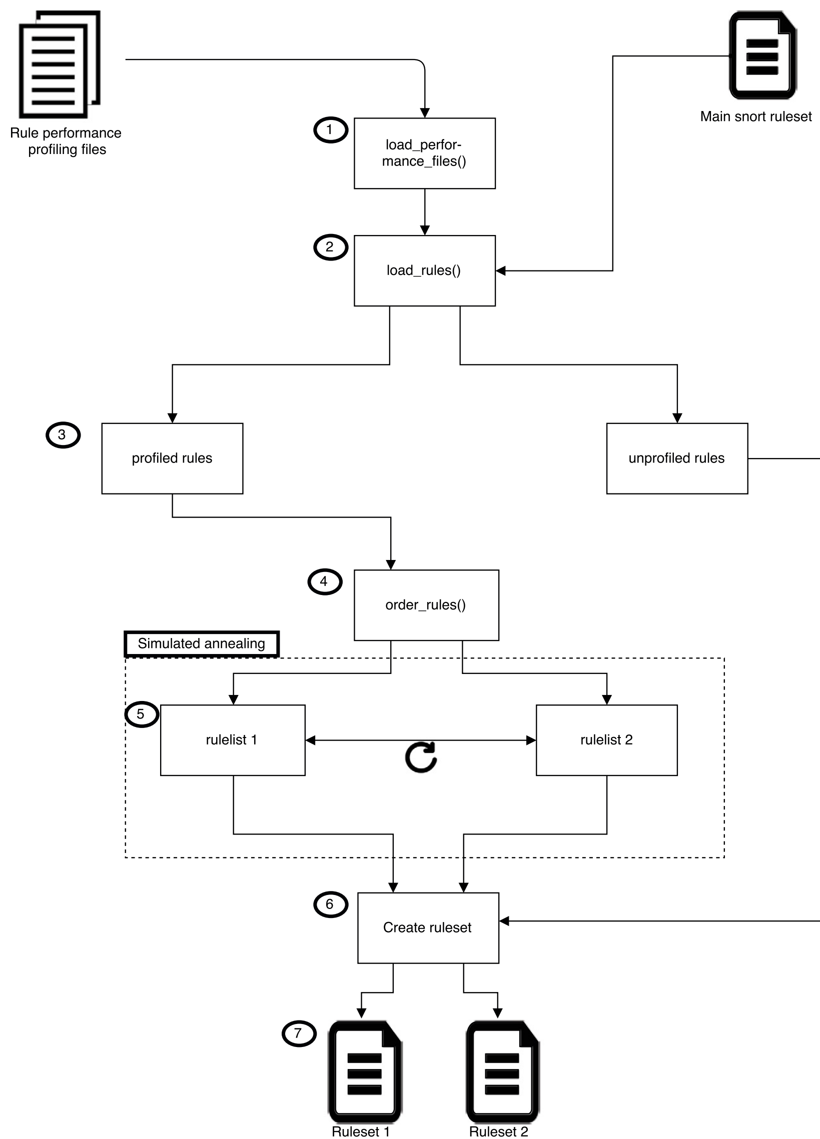


Fig. 3. Workflow diagram of the signature rule ordering algorithm.

Splitting the snort rules is a load balancing problem. In fact, by virtue of knowing the statistics of the matching time of each rule, distributing the snort rules over a fixed number of instances can be seen as a typical load balancing problem, which is known to be NP hard. In load balancing, the optimal solution is the one that minimizes the makespan of execution of a job over parallel machines. Here the makespan corresponds to the time the slowest snort instance [33] takes to process the packets. In our case, we cast the problem into equalizing workload which is not exactly the same as the above explained load balancing problem [33]. The reason why we opted for "load equalization" is to try to ensure that for any given packet, the pattern matching time is even across the different snort instances and thus the "flow duplication" and pattern matching go hand in hand, with no heterogeneous delay experienced by the same packet at the different snort instances. An algorithm that aspires to minimize the "makespan" (in this case, the time it takes to inspect the packet in the slowest IDS instance) might create a phenomenon which we call differentiated delay as a packet might be already processed with the slowest instance, while a new packet enters the fastest snort instance. In order to perform load equalization, we used Simulated Annealing which is known to be an efficient optimization algorithm. Other optimization algorithms could be used, also it is possible to devise different types of heuristics. We have for example developed in a previous work a game theoretical based approach for "equalizing" workload across different virtual machines in the cloud [34]. Usually the load equalization problem is a simple problem that can be solved efficiently using lightweight optimization algorithms since the processing times of the different rules are similar. The load equalization problem has been also studied as a variant of the Bin Packing problem, see for instance [35].

3.3.2. The dynamic scaling algorithm

While the rule ordering algorithm was developed to evenly distribute rules across the rule-sets, the dynamic scaling algorithm was developed in order to generate and distribute the new rule-sets to the Snort sensors in an efficient way. The dynamic scaling algorithm is run in the background and dynamically scales the Snort sensors in order to save system resources while sustaining the integrity of the Snort sensors and keeping a zero percent packet drop-ratio. The dynamic scaling algorithm consists of three important stages; the monitoring stage, the calculation stage and the reaction stage.

```
def order_rules(profiled_rules, time, unprofiled_rules,
files_needed): {
 // create empty lists in list based on the number
 // of rule-sets needed
bucket = [ [] for i in range(files_needed)]

 // random distribution of rules to bucket list
for rule in range (profiled_rules) do
  | append rule to bucket ;
end

 // Balancing the rule-sets using simulated annealing
if profiled_rules != EMPTY then
  INIT_TEMPERATURE = start temperature
  TEMPERATURE_DECREASE = decrease rate
  MIN_TEMPERATURE = stop temperature
  while INIT_TEMPERATURE > MIN_TEMPERATURE: do
    - use the simulated annealing method to reorder rules
    between the $bucket lists ;
    - calculate the score and accept or reject the solution;
    - remove rule from list if solution is accepted;
    - decrease temperature;
  end
append remaining unprofiled rules randomly to $bucket
}
```

Algorithm 3: Pseudo-code of the order_rules() procedure of the signature rule ordering algorithm.

The first stage in the scaling algorithm is to monitor a system resource parameter for a period of time. A system parameter can be CPU, RAM and hard disk usage among others. CPU usage turns out to be the right choice in our case. We have opted to choose CPU usage as an important factor for scaling decisions and for splitting the rules. The reason is that the CPU can be a bottleneck resource in NIDS. Snort operates with regular expression matching and thus, the CPU is the most important parameter to speed up the matching. Usually memory and disk IO demands do not differ much under heavy load of traffic due again to the computational nature of regular expression matching.

Once the monitoring part is done, the second stage will calculate how many Snort sensors are needed based on the usage of system resources in order to avoid the situation where packets are being dropped. At this junction, we give an example of how this calculation is performed. For instance, let us suppose that a system has 16 CPUs and thus a maximum CPU usage of 1600%, and assume that the CPU threshold, which is the only input to the scaling algorithm, is set to 40% in the current case. This indicates that we impose a constraint on Snort sensors to not exceed 40% and presumably this is under the assumption that a larger CPU usage could lead to packet dropping. If the total average CPU load of the whole system the last 45 s was measured to be 220%, the number of Snort sensors n needed to avoid a total CPU usage above the threshold would be calculated to be

$$n = \text{ceiling}\left(\frac{220}{40}\right) = \text{ceiling}(5.5) = 6$$

When this number has been computed, the reaction stage will adjust the number of Snort sensors that either need to be started or stopped. In case the calculation of the number of Snort sensors needed equals the current number of sensors, status quo will simply be kept. Additionally the number of Snort sensors that has been affected will need to be updated with new rule-sets. This is achieved using the signature rule reordering algorithm to create new rule-sets for the affected Snort sensors. The reaction stage will distribute the new rule-sets and reload all the affected Snort sensors so that they can resort to the new rule-set configuration.

The stages are outlined as pseudo-code in Algorithm 4.

```
CPU_THRESHOLD = threshold of CPU usage
interval = 45 s
active_Snort_sensors = active_sensors()
while True do
  total_cpu_usage = average total CPU usage last 45 s;
  n_sensors_needed = ceiling(total_cpu_usage /
  CPU_THRESHOLD)
  diff = n_sensors_needed - active_Snort_sensors
  while diff ! = 0 do
    if diff < 0 then
      Docker command: turn off one Snort sensor;
      diff + = 1
    else
      Docker command: turn on one Snort sensor;
      diff − = 1
  end
  // Have Snort sensors been turned on or off?
  if active_Snort_sensors != active_sensors() then
    - call signature rule ordering algorithm to create new
    rule-sets
    - send new rule-sets to sensors
    - reload Snort sensors
end
```

Algorithm 4: Pseudo-code of the dynamic scaling algorithm, given that the system resource considered is CPU usage.

4. Measurement, analysis and comparison

4.1. The underlying infrastructure

The underlying infrastructure consists of the important key tools and configurations needed to deploy a working parallel environment for the NIDS so that the algorithms can tested and experiments can be conducted. The system was built on two physical machines. The first machine is the main server where almost all key tools were installed and configured. The second machine is a client machine connected directly to the main server on its network interface card, replaying network traffic to the main server using the tcpreplay tool. Both machines are using Ubuntu Linux 14.04 64-bit version as operating system and a summary of the hardware specifications is listed in Tables 1 and 2.

4.1.1. Docker and snort

Snort version 2.9.9.9 was used for the IDS sensors. The Snort main ruleset file containing all Snort rules (approximately 35 000 rules) was downloaded using PulledPork, which is a rule management tool for Snort and Suricata. The reason for doing this was to retrieve the necessary directories and files structures which are later used as templates for Docker.

Docker version 1.13.0 was first installed through the apt-get package management. Before launching Snort containers, a Docker image was built to containerize the Snort application and its dependencies. Barnyard2 was used to parse binary output log files that contains alerts created by Snort and save the alerts in a centralized mySQL database.

Table 1
Main server specification.

Dell PowerEdge R710 hardware specifications	
CPU	16 cores @ 2.13 GHz
RAM	76 GB
NIC	5

Table 2
Client machine specification.

Dell Optiplex 780 hardware specifications	
CPU	2 cores @ 2.93 GHz
RAM	8 GB
NIC	2

4.1.2. Tcpreplay

Tcpreplay version 3.4.4 was installed using apt-get on the physical client machine and used to replay network traffic from a pcap file. The pcap file used in all the experiments consisted of 1.5 GBytes of recorded network traffic from a private network. Most of the traffic at such a place is just normal traffic and the number of alerts are hence few. The alerting capacity of an IDS is normally not a bottleneck, while the CPU demanding deep packet inspection using regular expressions often is. However, there are many publicly available pcacp file datasets that could be used as an alternative. We refer the interested reader to for example the large Netresec pcap dataset [36]. For the sake of repeatability, the source code of our project can be found in [37]. A Bash script was developed to replay the traffic at different settings, such as bandwidth speed and a given number of times in a loop, for the purpose of customizing the experiments.

4.1.3. Open vSwitch

Open vSwitch version 2.0.2 was installed and a network bridge was created for the Docker Snort containers in order to isolate network traffic, so that only traffic from the pcap file generated by the client machine is seen by the Snort sensors.

4.1.4. Real-time network traffic distribution architecture

The main purpose of the IDS is to analyse network traffic for intrusions, hence the importance of the network setup for the Snort sensors to analyse the correct network traffic. Regarding the physical aspect of the network setup, only one port is configured to run as a mirror port. However, the Snort sensors have their own isolated virtual network, so there is a need to duplicate and distribute this traffic further to each Snort sensor. For this purpose, a virtual switch is used to connect the physical and virtual ports together in order for all Snort sensors to receive the same data. An illustration of the networking model is shown in Fig. 4 where all data flow directions represent duplicated traffic. In this proposed architecture, the network traffic is real-time traffic.

4.1.5. Emulated network traffic environment architecture

This architecture adopts a similar approach to the real-time network traffic distribution architecture. The only difference in this approach is that instead of sniffing and duplicating real-time traffic from the network, we use tcpreplay to replay a pcap file to emulate network traffic. The server will receive the traffic replayed from the pcap file and traffic is duplicated and sent to the Snort sensors for inspection.

4.2. Initial experiments

The general procedure of the experiments was to replay the recorded pcap network traffic at a set of different speeds and record the CPU usage and the total drop rate of the Snort sensors. A Snort ruleset containing 31 607 rules and rule profiling performance files that contained 3345 unique rules were used. The latter rules are the ones which were matching some of the packets of the test pcap file and for which the profiling data were collected. The rest of the rules were also a part of the experiments, but these rules never explicitly led to time consuming deep packet inspection in the experiments conducted.

Initial experiments showed that the memory was not a bottleneck as the memory utilization was almost stable even for heavy workloads, and just increased slightly with the increase of the number of rules.

However, other initial experiments also showed that there were strong correlations between the CPU usage and the packet drop rate. Fig. 5 shows the results of a set of experiments where the network throughput is increased from 10 Mbps to 50 Mbps. At the lowest throughput the CPU usage of the Docker container running the Snort sensor is 40% while no packets are dropped. However, as the speed of the replayed TCP traffic is increased inducing a CPU usage above 40%, the Snort sensor is congested and starts to drop packets. The results show that as the CPU load increases up to 100%, the drop rate of packets also slowly increases up to 14%. An important goal of the system is to dynamically add more sensors when the current sensors reach congestion and these results indicate that the CPU load should not exceed by far 50% in order to avoid that network packets pass the IDS unexamined.

4.3. Signature rule ordering algorithm

In this experiment, we test the effectiveness of the distribution of signature rules across n number of rule-sets performs using the simulated annealing rule ordering algorithm. The results in Table 3 show three tests where the initial 3345 profiled rules are distributed into 2, 3 and 4 rule sets respectively. First the rules are distributed randomly and with an equal number in each set. The initial total time is the sum of the time in seconds each of the rules spent in the detection engine for all the rules in the given set. Initially, there is some difference and the rule ordering algorithm is applied in order to make the total time of each rule set as even as possible. Ideally, all rulesets should have an even execution time. We see that the simulated annealing algorithm in these three cases gives an almost perfect result which should be good enough for any practical case. The total number of rules used was also in this experiment 31 607, but the rules for which there where no profiling data were just evenly divided between the sets as there would not be any significant difference in processing time when comparing them.

The results of a set of experiments conducted to see how well the signature rule ordering algorithm performed when receiving real TCP-traffic is shown in Fig. 6. The signature rule ordering algorithm (intelligent rule ordering) was applied to divide the 31 607 rules into two sets and 5 different experiments was performed. An overwhelming stream of network packets was sent to both Snort sensors and the number of packets analysed and dropped was logged. If the rule ordering is perfectly balanced, the two sensors should drop equally many packets in each of the experiments. As can be seen from Fig. 6, the difference is very small. On average the difference between the percentage of packets dropped by the two Snort sensors is 0.25% for the five experiments.

However, in the random rule ordering experiments shown in Fig. 7, the difference between packets analysed and dropped

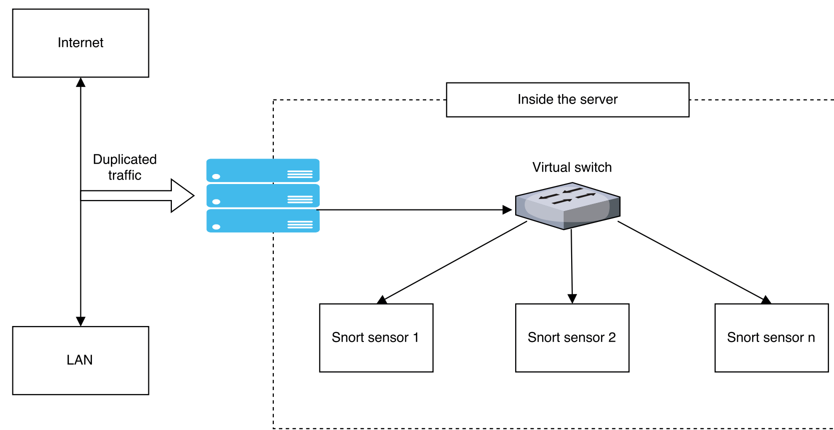


Fig. 4. Real time network traffic distribution model.

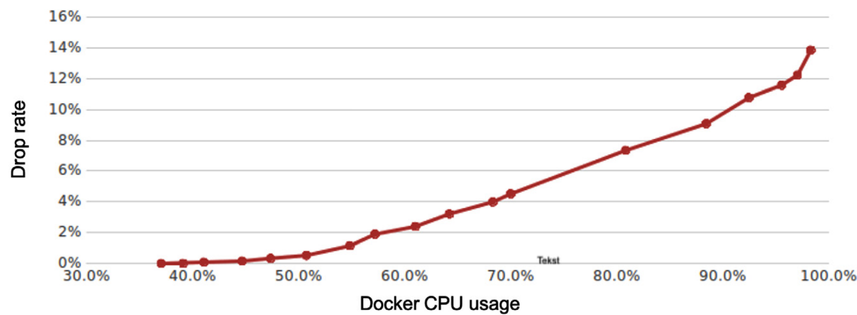


Fig. 5. Packet drop rate correlated to CPU load.

Table 3
Performance of the signature rule ordering distribution of rules. The final total time given in seconds is very close to the Balanced execution time for each of the three tests.

Test 1	Balanced execution time 992.67			
Rule-set	1	2		
Initial total time	987.98	997.36		
Final total time	992.64	992.70		
Rules in rule-set	1689	1656		
Test 2	Balanced execution time 661.78			
Rule-set	1	2	3	
Initial total time	653.67	618.43	713.24	
Final total time	661.76	661.81	661.76	
Rules in rule-set	1159	1128	1058	
Test 3	Balanced execution time 496.33			
Rule-set	1	2	3	4
Initial total time	466.77	522.89	527.13	468.55
Final total time	496.38	496.28	496.36	496.31
Rules in rule-set	832	844	830	839

varies more and in the second experiment the difference is rather very large. The latter could be a rare anomaly, but for the four other experiments the average difference between the percentage of packets dropped by the two Snort sensors is 1.5%. One can conclude that with the use of signature rule ordering algorithm, signature rules are evenly distributed across rule-sets which makes the Snort sensors balanced in terms of packet inspection. This makes it possible to set up a system which adds more Snort sensors while keeping the load balanced when the existing Snort sensors are not able to cope with the load and the CPU usage and rate of packet drop get too large. Our algorithm performs rule profiling to distinguish which rules are using the most time in the detection engine for pattern matching for packets in order to divide the rules evenly. If this logic is not taken into account when developing the algorithm, one would see an imbalance in terms of pattern matching time across the snort sensors over time. A random split of the rules might put "heavy rules" requiring long pattern matching time in the same snort sensor which would create bottlenecks and uneven processing time of the same packets across the different snort instances.

4.4. Dynamic scaling algorithm

A set of full scale prototype experiments were conducted to see how well scaling of Snort sensors performed when exposed to varying amounts of network traffic. The speed of the TCP-packet flow started at 5 Mbps and was gradually increased to a maximum of 300 Mbps and then slowly reduced back to 5 Mbps at the end of the experiments. This procedure lasted for more than two hours and the total CPU usage of the system was recorded every second together with the number of active Snort sensors. Three experiments were performed by varying the CPU load threshold of the dynamic scaling algorithm. All the parameters in the experiment are user defined and generally depend on the characteristics of the traffic. Finding appropriate parameters is usually done by manual tuning. In the experiments, we have tested 50%, 60% and 70%, as threshold CPU load based on the results of the initial experiment shown in Fig. 5.

In the first case where the threshold for total CPU load was set to 70%, the results of Fig. 8 show that the algorithm was able to adjust the number of Snort sensors in response to the CPU load. When the CPU load is substantially increased, the system adds more Snort sensors in order to keep the CPU load below the threshold. In the same manner Snort sensors are removed when the CPU load is decreased and the value stays low. In this specific case the figure shows the actual value of the CPU usage for every second and this value fluctuates strongly. In

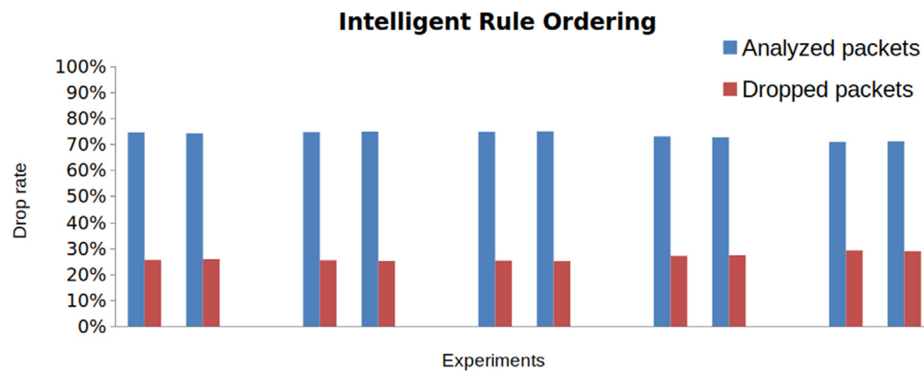


Fig. 6. Results of five packet drop rate experiments using two Snort sensors with intelligent rule ordering.

Fig. 7. Results of five packet drop rate experiments using two Snort sensors with random rule ordering.

order to avoid that new IDS sensors where added and withdrawn to often, the decision to change the number of sensors was taken every 45 s based on the average value of the CPU usage for the given interval. Loosely speaking, there are two types of scaling approaches: (i) schedule based and performance triggered. In schedule based approaches, usually, the network administrator defines a regular interval at which a scaling action might be performed. A performance triggered scaling denotes a set of approaches where some of the scaling is rather reactive to degradation in performance. We have opted for a schedule based scaling although a performance triggered approach could also have been used. The interval of 45 s is a user defined parameter which defines the reaction time of our IDS. We measure the average CPU for 45 s to take scaling action based on threshold based policies. A shorter time interval would lead to very frequent scaling actions, which might not be necessary in case of short bursts in CPU usage. In this case, we might experience a ping-pong effect, while resources can increase and decrease in an oscillating and unstable manner [38]. A longer interval might lead to either degradation of the performance or unnecessary usage of the resources over a long time depending on the workload. We cannot claim that 45 s is an optimal value, a more thorough study needs to be conducted to find an optimal value. Such a question has been very well studied within the field of elastic cloud infrastructure in general [39], for instance many works on elastic load balancing of web services have studied the optimal choice of the scaling schedule interval. We leave determining the optimal scaling schedule interval depending on the characteristic of the traffic to a future work.

The experiments lasts for 2 h and 20 min, giving the 8400 CPU load data points of Fig. 8.

In Fig. 9 the threshold CPU usage is reduced to 60%. In this sense, the scaling algorithm will try to avoid that any of the Snort sensors uses more than 60% CPU time in order to reduce the amount of packets dropped. The algorithm strives to achieve this by providing as many sensors as needed to deal with the current CPU usage without the sensors on average using a higher percentage than the threshold. In Fig. 9 the blue line shows the average CPU usage of the system during the last 45 s. Since the system has 16 CPUs, the maximum CPU usage is 1600%. When the average CPU load for instance exceeds 600% the system needs to run more than 10 CPUs at 60% to avoid exceeding the given threshold of 60%. Because of this, the amount of sensors needed is calculated to be 11 and the number of sensors is adjusted accordingly, unless the system is already using this number of Snort sensors. In this way the system dynamically responds to the fluctuating amount of network traffic, striving to keep the CPU threshold using a minimum number of CPU kernels. When the actual CPU load reaches 800% one can see that up to 13 CPUs are needed in order to cope with the sensors need for computations. As the load is reduced and the total CPU usage falls below 60%, only a single Snort sensor is needed to handle the TCP traffic as seen in the right part of the graph, more than two hours into the experiment and close to the end.

From the initial experiments presented in Fig. 5 we observed that when the Docker Snort sensor usage of approximately 50%, the system started to drop packets because of network congestion. This might suggest that the system should run at a threshold of 50% in order to both keep a good quality of service by avoiding packet drop and to use as few CPU resources as possible. The results of scaling the system at a threshold of 50% is shown in Fig. 10. In all the three cases, the same network traffic is sent and in order to force a lower threshold the system has to spawn more resources. Because of this, we can see that the system at the peek loads need to deploy the maximum possible of 16 sensors, but still the total load slips above 800% exceeding the demanded threshold. At that stage, there are however no more resources available to the system and this cannot be avoided given the current hardware setup.

In Fig. 11, we zoom in on the first 25 min of the complete experiment shown in Fig. 10. As in the 70% threshold case of

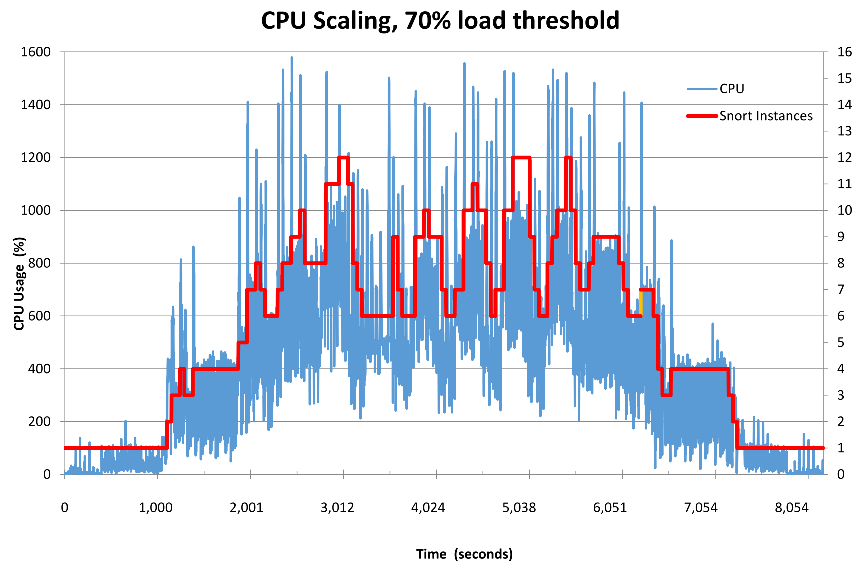


Fig. 8. Scaling performance with a threshold of 70% CPU load.

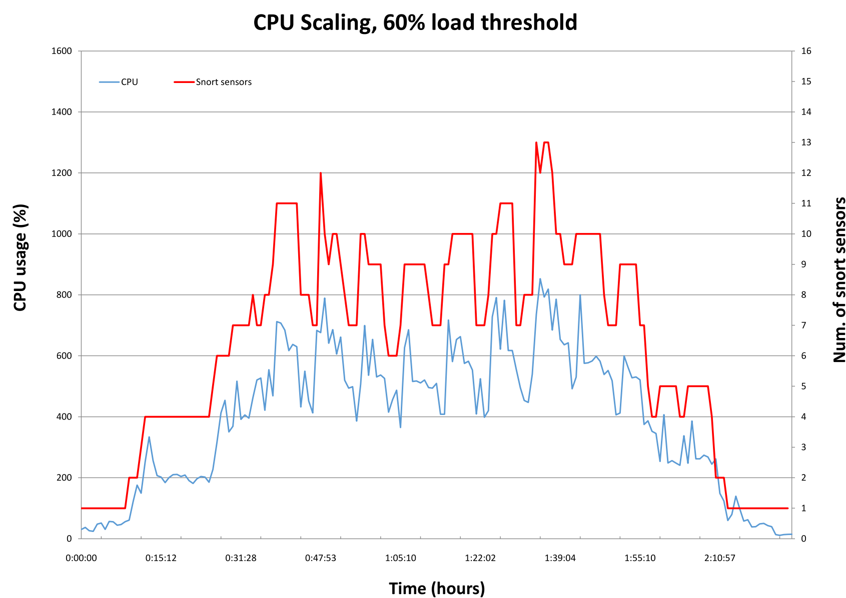


Fig. 9. Scaling performance with a threshold of 60% CPU load.

Fig. 8, the load recorded every second is shown, while the scaling algorithm calculates the average load every 45 s, at every horizontal tick-label of the graph, and makes its decision based on this number. So even though there are some high CPU peeks after two and three minutes, it is not until the average exceeds a 70% CPU usage after 11 min that the number of sensors is increased to two by the algorithm. From this graph, one can also see that when the number of Snort sensors is adjusted in response to the workload, the CPU load is temporarily heavily increased as well, leading to the large spikes of the CPU load immediately after the addition of a new sensor. The reason for this is that all the Snort sensors are in the process of being reloaded with new rule-sets, along with one (and in other cases more) new Snort sensor, thus, sharing the load among them in a new way. This CPU workload needs to be handled but the scaling algorithm has been coded to ignore the additional CPU load Snort spends on reloading in such events.

As the three experiments were conducted, the packet drop rate and average CPU load of each experiment was also logged.

Table 4
Packet drop rate and CPU load for the three scaling experiments with varying CPU threshold.

CPU threshold	Packet drop rate	Average CPU load
50%	0,00%	438%
60%	0,15%	413%
70%	1,07%	379%

The packet drop rate is the percentage of network-packets dropped by the Snort sensors during the experiments. When a Snort sensor is overloaded it will not find time to analyse all packets and has to let some packets proceed without analysing them. Table 4 shows that the CPU load threshold needs to be decreased to 50% in order to avoid that Snort drops packets.

As explained in Section 4.4, since the system has 16 CPUs, the maximum CPU usage is 1600% and the total average CPU load of the system is seen to exceed 100% in the table. By increasing the CPU threshold to 60% there will be some packets lost and even

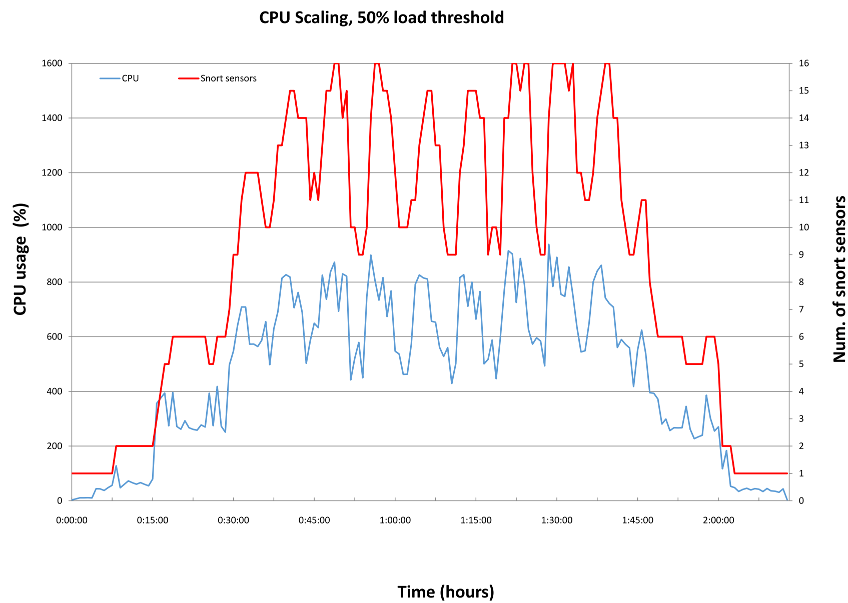


Fig. 10. Scaling performance with a threshold of 50% CPU load.

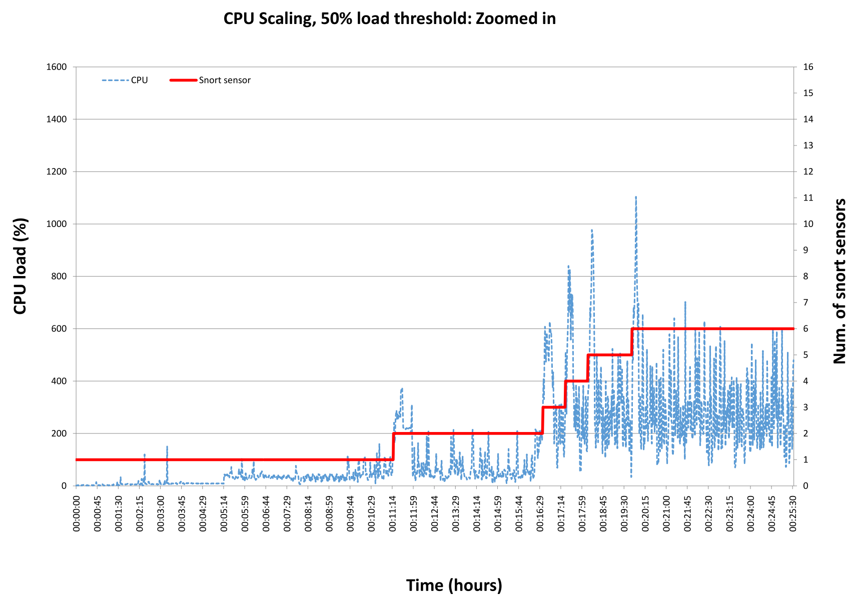


Fig. 11. Scaling performance with a threshold of 50% CPU zoomed in.

more packets are lost at 70%. These results are comparable to the results of the initial experiments of Fig. 5, when an average CPU usage above 50% occurs, the Snort sensors might have to drop packets.

Increasing the CPU threshold for the individual Snort sensors from 50% and up to 70% will in practice mean that on average fewer sensors will have to deal with all the network load. Then the packet drop rate will increase while the average CPU load of the whole system, shown in the third column of Table 4, will decrease as there is less overhead running fewer sensors and some of the packets are just dropped and not examined. As can be seen from the Average CPU load column of Table 4, allowing a higher CPU threshold reduces the total CPU usage slightly as some packets are dropped and there is less overhead starting new Snort sensors. However, this is a reasonably small price to pay for avoiding that TCP packets bypass examination. Whenever a scaling takes place requiring rule distribution, the affected snort sensors will be updated with new rulesets and restarted for the new rulesets to take effect while keeping the integrity of the affected sensors. In other words, while snort sensors are restarting with new rulesets, incoming packets should still be inspected or else possibilities of intrusions can be undetected within this time frame. The time used for a snort instance to be active or inactive should also be done in a quick manner, hence for stopping and starting a container from scratch, the Docker pause and unpause command was used. We have recorded around 0.2 s for pausing or unpausing a docker instance on which Snort runs. Snort also takes some time to reload the new configuration which is in the order of 1 s. Thus, in total, redistributing the rules, scaling and reloading the rules takes around 1.2 s. Also it is reported that reloading the configuration in snort might lead to some packets, meaning a few packets to pass uninspected according to technical documentation related to Snort [40].

Since the captured trace in our case reflects regular traffic, attacks within the traffic are envisaged. However, we did not monitor the logs of the IDS to discover this as this was not

the main aim of our study. Our system should detect an intrusion faster as the "throughput" of the IDS is boosted using rule splitting over parallel instances.

5. Conclusion

In this paper, we propose an innovative design of a parallel NIDS based on NFV and rule distribution. Using Open vSwitch and Docker containers running Snort, the IDS functions are virtualized which yields flexibility and elasticity in the face of varying network traffic. Using a modified parallel technique, NL-FP-1, we create a working parallel environment and devise two algorithms which dynamically adjust and divide the signature rules evenly across NIDS nodes. In this sense, by profiling the matching time of the IDS rules, we split the rules over the different instances in a manner that aspires to equalize the processing time of a packet over the different snort sensors. Such a strategy ensures that a packet will ensure the same processing delay across those instances.

The signature rule ordering algorithm keeps a balance across active NIDS nodes in terms of both the system resource usage and the packet drop rate. This enables the dynamic scaling algorithm to at any time adjust the number of Snort nodes needed in response to the CPU load on the system. This ensures that enough system resources are allocated when needed to avoid that packets are dropped by the IDS for large workloads while at the same time resources are released when they are not needed. Measuring the average CPU load imposed by the sensor workload, the system ensures that no or just a small percentage of packets are dropped by the IDS system. The level of perfection can simply be adjusted by decreasing the CPU load threshold, which is the only input to the system, until there are no packets being dropped. In the prototype experiments the threshold was determined to be at a total CPU load of approximately 50%, but this would probably have to be fine tuned when using the system in other environments and for other network workloads.

A possible future improvement to the system could be to also adopt the percentage of packets dropped as a factor in the scaling algorithm, in order to impose a minimal level of dropped packets.

The prototype system where the experiments were performed is just an example environment where this setup can be implemented. The virtualized nature of the solution means it can readily be implemented in various infrastructures involving Software-defined Networking (SDN), virtual machines and containers.

CRediT authorship contribution statement

Hårek Haugerud: Conceptualization, Methodology, Software, Writing, Visualization, Validation, Resources. **Huy Nhut Tran:** Methodology, Software, Writing, Investigation. **Nadjib Aitsaadi:** Writing - review & editing, Validation. **Anis Yazidi:** Conceptualization, Methodology, Writing, Project administration, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] L.N. Tidjon, M. Frappier, A. Mammar, Intrusion detection systems: A cross-domain overview, IEEE Commun. Surv. Tutor. 21 (4) (2019) 3639–3681.

[2] A. Khraisat, I. Gondal, P. Vamplew, J. Kamruzzaman, Survey of intrusion detection systems: techniques, datasets and challenges, Cybersecurity 2 (1) (2019) 20.

[3] C.V. Kopek, E.W. Fulp, P.S. Wheeler, Distributed data parallel techniques for content-matching intrusion detection systems, in: Military Communications Conference, 2007. MILCOM 2007. IEEE, IEEE, 2007, pp. 1–7.

[4] T. Limmer, F. Dressler, Adaptive load balancing for parallel IDS on multi-core systems using prioritized flows, in: 2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN), IEEE, 2011, pp. 1–8.

[5] L. Schaelicke, K. Wheeler, C. Freeland, SPANIDS: a scalable network intrusion detection loadbalancer, in: Proceedings of the 2nd Conference on Computing Frontiers, ACM, 2005, pp. 315–322.

[6] F.I. Shiri, B. Shanmugam, N.B. Idris, A parallel technique for improving the performance of signature-based network intrusion detection system, in: Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on, IEEE, 2011, pp. 692–696.

[7] P. Wheeler, E. Fulp, A taxonomy of parallel techniques for intrusion detection, in: Proceedings of the 45th Annual Southeast Regional Conference, ACM, 2007, pp. 278–282.

[8] A. Gupta, L.S. Sharma, A categorical survey of state-of-the-art intrusion detection system-Snort, Int. J. Inf. Comput. Secur. 13 (3–4) (2020) 337–356.

[9] S.A.R. Shah, B. Issac, Performance comparison of intrusion detection systems and application of machine learning to Snort system, Future Gener. Comput. Syst. 80 (2018) 157–170.

[10] B.G. Atli, Y. Miche, A. Jung, Network intrusion detection using flow statistics, in: 2018 IEEE Statistical Signal Processing Workshop (SSP), IEEE, 2018, pp. 70–74.

[11] H. Jiang, G. Xie, K. Salamatian, Load balancing by ruleset partition for parallel IDS on multi-core processors, in: International Conference on Computer Communications and Networks, ICCCN, 2013.

[12] K. Xinidis, I. Charitakis, S. Antonatos, K.G. Anagnostakis, E.P. Markatos, An active splitter architecture for intrusion detection and prevention, IEEE Trans. Dependable Secure Comput. 3 (1) (2006) 31–44.

[13] C.-H. Lin, C.-H. Liu, L.-S. Chien, S.-C. Chang, Accelerating pattern matching using a novel parallel algorithm on GPUs, IEEE Trans. Comput. 62 (10) (2012) 1906–1916.

[14] M.A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, K. Park, Kargus: a highly-scalable software-based intrusion detection system, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, 2012, pp. 317–328.

[15] W. Yuan, J. Tan, 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.

[16] S. Bagheri, A. Shameli-Sendi, Dynamic firewall decomposition and composition in the cloud, IEEE Trans. Inf. Forensics Secur. (2020).

[17] R. Mohan, A. Yazidi, B. Feng, J. Oommen, On optimizing firewall performance in dynamic networks by invoking a novel swapping window–based paradigm, Int. J. Commun. Syst. 31 (15) (2018) e3773.

[18] K. Wakabayashi, D. Kotani, Y. Okabe, Traffic-aware access control list reconstruction, in: 2020 International Conference on Information Networking (ICOIN), IEEE, 2020, pp. 616–621.

[19] W. Bulajoul, A. James, M. Pannu, Network intrusion detection systems in high-speed traffic in computer networks, in: E-Business Engineering (ICEBE), 2013 IEEE 10th International Conference on, IEEE, 2013, pp. 168–175.

[20] M. Aldwairi, N. Ekailan, Hybrid multithreaded pattern matching algorithm for intrusion detections systems, J. Inf. Assur. Secur. 6 (6) (2011) 512–521.

[21] H.G.A. Umar, C. Li, Z. Ahmad, Parallel component agent architecture to improve the efficiency of signature based NIDS, J. Adv. Comput. Netw. 2 (4) (2014).

[22] M. Kumar, M. Hanumanthappa, Self tuning IDS for changing environment, in: Computational Intelligence and Communication Networks (CICN), 2014 International Conference on, IEEE, 2014, pp. 1083–1087.

[23] L. Foschini, A.V. Thapliyal, L. Cavallaro, C. Kruegel, G. Vigna, A parallel architecture for stateful, high-speed intrusion detection, in: International Conference on Information Systems Security, Springer, 2008, pp. 203–220.

[24] D.L. Schuff, Y.R. Choe, V.S. Pai, Conservative vs. optimistic parallelization of stateful network intrusion detection, in: Performance Analysis of Systems and Software, 2008. ISPASS 2008. IEEE International Symposium on, IEEE, 2008, pp. 32–43.

[25] X. Chen, Y. Wu, L. Xu, Y. Xue, J. Li, Para-snort: A multi-thread snort on multi-core ia platform, in: Proceedings of Parallel and Distributed Computing and Systems (PDCS), Citeseer, 2009.

[26] Intel, Supra-linear packet processing performance with intel multi-core processors white paper, 2006.

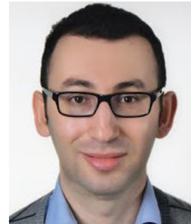
[27] R. Souza, K. Dias, S. Fernandes, NFV data centers: A systematic review, IEEE Access 8 (2020) 51713–51735.

[28] M.S. Bonfim, K.L. Dias, S.F. Fernandes, Integrated NFV/SDN architectures: A systematic literature review, ACM Comput. Surv. 51 (6) (2019) 1–39.

[29] X. Wu, K. Hou, X. Leng, X. Li, Y. Yu, B. Wu, Y. Chen, State of the art and research challenges in the security technologies of network function virtualization, IEEE Internet Comput. 24 (1) (2019) 25–35.
[30] O. Yurekten, M. Demirci, SDN-based cyber defense: A survey, Future Gener. Comput. Syst. (2020).
[31] A. Chowdhary, D. Huang, SDN based network function parallelism in cloud, in: 2019 International Conference on Computing, Networking and Communications (ICNC), IEEE, 2019, pp. 486–490.
[32] G. Vasiliadis, M. Polychronakis, S. Ioannidis, MIDeA: a multi-parallel intrusion detection architecture, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, 2011, pp. 297–308.
[33] B. Johannes, Scheduling parallel jobs to minimize the makespan, J. Sched. 9 (5) (2006) 433–452.
[34] D. Sangar, R. Upreti, H. Haugerud, K. Begnum, A. Yazidi, Stable marriage matching for homogenizing load distribution in cloud data center, in: Transactions on Large-Scale Data-and Knowledge-Centered Systems XLV, Springer, 2020, pp. 172–198.
[35] A. Trivella, D. Pisinger, Bin-packing problems with load balancing and stability constraints, INFORMS Transp. Logist. Soc. (2017).
[36] A.B. Netresec, Netresec pcap dataset, 2021, https://www.netresec.com/?page=pcapfiles [Online; accessed 20-February-2021].
[37] H.N. Tran, A Dynamic Scalable Parallel Network-based Intrusion Detection System using Intelligent Rule Ordering (Master's thesi), University of Oslo, 2017.
[38] Y. Kouki, F.A. De Oliveira, S. Dupont, T. Ledoux, A language support for cloud elasticity management, in: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, IEEE, 2014, pp. 206–215.
[39] G. Galante, L.C.E. de Bona, A survey on cloud computing elasticity, in: 2012 IEEE Fifth International Conference on Utility and Cloud Computing, IEEE, 2012, pp. 263–270.
[40] Cisco, Firepower management center configuration guide, 2015, https://www.cisco.com/c/en/us/td/docs/security/firepower/60/configuration/guide/fpmc-config-guide-v60/Policy_Management.html#concept_33516C5D6B574B6888B1A05F956ABDF9 [Online; accessed 20-February-2021].

Dr. Hårek Haugerud received his M.Sc. and Ph.D. degrees from the University of Oslo and is currently an associate professor at Oslo Metropolitan University (OsloMet). He joined OsloMet in 1998. He is member of the research group Autonomous Systems and Networks.

Huy Tran received his M.Sc. degree in Computer Science from the University of Oslo in 2017 and has since then worked for Greenbird Integration Technology, Oslo, Norway where he is currently a Senior Site Reliability Engineer.



Nadjib Aitsaadi is full professor at UVSQ Paris-Saclay university since November 2019. From September 2016 to October 2019, he was full professor of computer science at ESIEE Paris (engineering school).

From September 2011 to August 2016, he was associate professor of computer science at University of Paris-Est Creteil Val de Marne.

From June 2010 to August 2011, he was research fellow at INRIA.

In July 2016, Nadjib AIT SAADI obtained the HDR (habilitation diploma) in computer sciences from University Paris Est (UPE) in France.

In March 2010, he obtained the Ph.D. in computer sciences with honors from LIP6 - Sorbonne University (ex UPMC university - Paris 6) in France.

Professor Anis Yazidi received the M.Sc.and Ph.D. degrees from the University of Agder, Grimstad,Norway, in 2008 and 2012, respectively. He was a Researcher with Teknova AS, Grimstad, Norway. He is currently a Full Professor with the Department of Computer Science, Oslo Metropolitan University, Oslo, Norway, where he is leading the research group in Applied Artificial Intelligence. His current research interests include machine learning, learning automata, stochastic optimization, and autonomous computing.