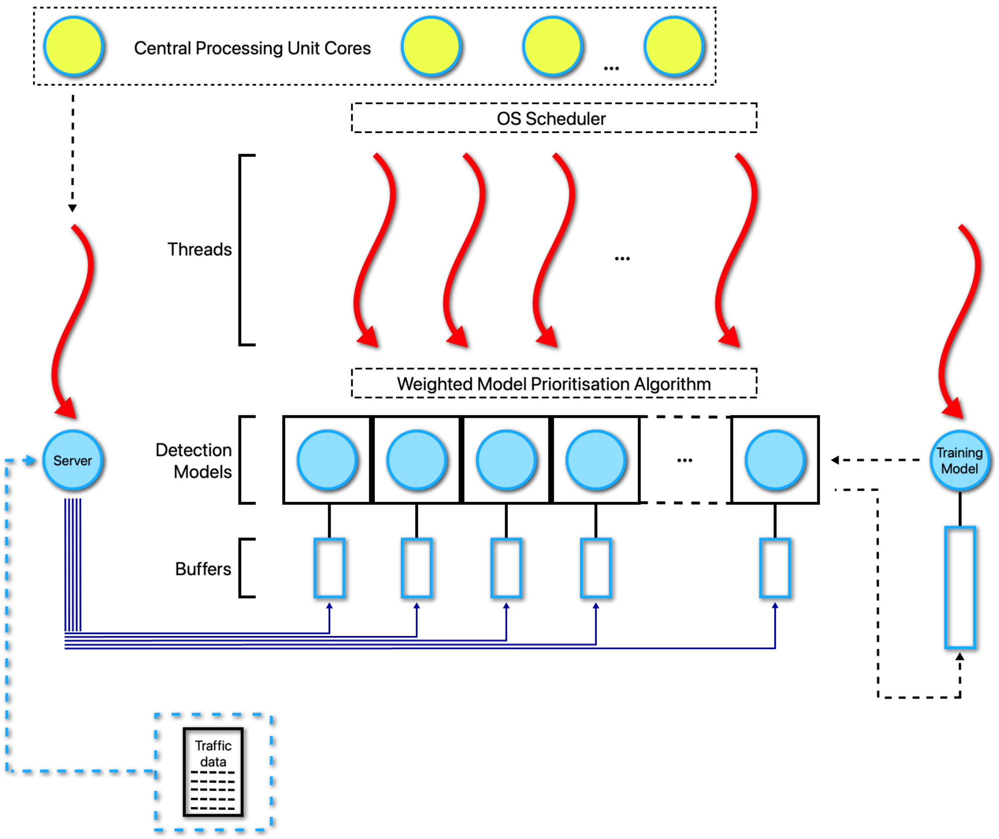
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Investigating Dynamic User-Level Scheduling to Improve AI-Based Intrusion Detection Systems on IoT

Master's thesis in Computer Science and Engineering

Aria Mirzai & Ali Zülfükar Coban

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

Investigating Dynamic User-Level Scheduling to Improve AI-Based Intrusion Detection Systems on IoT

Aria Mirzai

Ali Zülfükar Coban

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Investigating Dynamic User-Level Scheduling to Improve AI-Based Intrusion Detection Systems on IoT

Aria Mirzai
Ali Zülfükar Coban

Supervisor: Magnus Almgren, CSE
Advisor: Wissam Aoudi, Clavister
Examiner: Pedro Petersen Moura Trancoso, CSE

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Displays the architecture of a dynamic user-level scheduler, which is the main contribution of this thesis work. A more detailed version of this image along with describing text is provided in Section 4.2.

Typeset in LaTeX
Gothenburg, Sweden 2022

Investigating Dynamic User-Level Scheduling to Improve AI-Based Intrusion Detection Systems on IoT

Aria Mirzai
Ali Zülfükar Coban
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Internet of things devices with their inherent convenience factor have exploded in numbers during the latest decade, however at the cost of rising security concerns. This is largely due to their incapability of solving complex and computationally heavy numerical problems especially when dealing with large data-sets, a key component for computers in today's world for fending off attacks.

The main contribution of this thesis is investigating how a dynamic user-level scheduler can improve the detection capabilities of AI-based intrusion detection systems and to enable retraining of an AI algorithm on an IoT device. The models are assumed to be made of lightweight and data-driven machine learning algorithms, such as "PASAD" which we chose to utilize for this work. The scheduler was created after having initially developed a basic framework for allowing the PASAD models to detect attacks, denoted as our "baseline" system.

The experiments that followed proved that the dynamic user-level scheduler provides several additional advantages compared to the baseline, mainly a substantial throughput increase which reduces the time until attacks are detected, a critical factor from the security aspect. Additionally, a model prioritization feature was built to allow the scheduler to allocate more processing resources towards nodes it is suspecting to be under attack. Both of these variables play an important role in pawing the way to having our IoT devices being protected by more robust security schemes, even for those devices considered too resource limited today.

With our scheduler implemented on an Nvidia Jetson Nano, is it possible to calculate approximately 57,000 anomaly scores per second, which are used in the attack monitoring process, for roughly 97 detection models while simultaneous retraining is taking place (results are for when PASAD is the utilized detection algorithm). Furthermore, with 75 PASAD models the scheduler is able reach ≈1.46 times the performance of the baseline with retraining enabled and with retraining disabled it reaches ≈2.15 times the performance of the baseline.

Keywords: Internet of things, Anomaly-based intrusion detection system, User-level scheduling, model training

Acknowledgements

We wish to greatly thank our two supervisors, Magnus Almgren and Wissam Aoudi, who wisely guided us throughout this work using their exceptional expertise in computer security. A special thanks to Tobias Bertilsson as well who became our main advisor when practically building our solutions, even without technically being required to gift his time for this thesis. Lastly, we want to thank CELTIC-NEXT AI-NET-PROTECT (C2019/3-4) and Clavister for providing us with the resources needed to make this a reality.

Aria Mirzai & Ali Zülfükar Coban, Gothenburg, July 2022

Contents

List of Figures

List of Tables

1
Introduction

The Internet of Things (IoT) signifies a major paradigm-shift, enabling us to have nodes collecting information and then process that information (at the edge or in the cloud) to thereafter actuate changes in the physical environment.

1.1 IoT devices in our daily life

The convenience of IoT has caused a rapid surge in the number of such devices. In fact, McAfee anticipates that the number of IoT devices will have increased to about 75 billion by 2025 [4]. In addition to low hardware prices and ever-increasing network communication speeds, the adaption of IoT devices will likely be sped up by the incoming mainstream use of 5G according to Nokia [5].

The key underlying characteristics that make IoT devices work are what is often called *"Data acquisition and control"* as well as *"Data processing and storage"* [6]. Essentially, IoT devices can contain sensors that acquire data, and actuators controlling or acting upon that data. IoT devices are also incorporated with the computing capabilities to process and store sensor data locally if necessary.

1.2 The IoT security trade-off

IoT offers many advantages, however there are multiple concerns with regard to cyber security since these end nodes can easily be attacked and used for massive attacks. According to the Open Web Application Security Project, the potential security vulnerabilities of IoT devices most often stem from factors such as weak passwords, poor default security settings, lack of network communication encryption, and poor user-serviceable device management [7]. In April 2019, security researchers in the Microsoft Threat Intelligence Center uncovered that an actor had used three IoT devices to gain initial access to corporate networks [8]. Two of them were compromised because of still using the default password given by the manufacturer, while the third was caused by not having installed the latest security update. After having gained an initial network presence through these devices, performing a simple network scan was all that was needed to look for more insecure devices, possibly containing higher-privileged data and accounts.

Another example that show how IoT devices can easily be used to perform large attacks is the Mirai Botnet [9]. Here, malware-infected devices were found through internet scans and essentially turned into a network of remotely controlled bots used to launch distributed Denial-of-Service attacks. In one famous attack [10], the Mirai botnet consisted of a hundred thousand hijacked IoT devices and was used to bring down the Domain Name System (DNS) provider Dyn. As a result, users could not access popular websites such as PayPal, Amazon, Twitter and many more. Mirai's source code still remains a threat to this day due to its ability to mutate, giving birth to numerous variants that exploit different weaknesses and are equipped with unique capabilities.

The explanation for manufacturers seemingly treating security in their IoT devices as an afterthought, is often that it is a natural trade-off for ensuring them being cost-effective. It is already difficult enough to provide an inexpensive, reliable, resource-constrained device that can connect to a wireless network while using very little power.

1.3 Intrusion detection systems

To combat security threats, intrusion detection systems (IDS) have become invaluable tools for attack detection. An intrusion detection system is, simply put, a system that monitors and analyzes an event stream such as network traffic data. IDS are vital since early reaction and identification of malicious traffic ensures that the IDS can stop an adversary before they can compromise devices and cause severe damage [11]. IDS can be of two sorts, either signature-based or anomaly-based.

1.3.1 Signature-based IDS

The essence of Signature-based IDS (SIDS) is storing the signatures of previously recorded attacks, or intrusions. Through a pattern matching algorithm, it can then compare incoming network traffic packets with the known attack-signatures to determine if the traffic is malicious or not. This is also referred to as Knowledge based Detection or Misuse Detection [12]. SIDS are very effective at detecting attacks due to the comparison of attack-signatures with data streams. However, this is also their main limitation since it renders them unable to detect zero-day attacks, i.e. novel attacks that have not occurred previously. Some of the most popular SIDS are Snort [13] and Zeek (formerly known as Bro) [3] both of them being open source [14, 15]. We will go into more detail about Zeek in Section 2.1.

1.3.2 Anomaly-based IDS

To make the IDS even more effective, the approach of incorporating machine learning algorithms has gained momentum. A major reason for using Anomaly-based IDS is the fact that it gives you the possibility of monitoring communication traffic through machine learning, based on common techniques such as supervised or unsupervised learning [16, 17].

It all starts with big data analytics, which enables the discovery of trends, preferences and the prediction of behaviours via either associating dependencies between input variables or through Data Mining (DM) tools and techniques [1]. Data mining is the process of discovering hidden patterns, relationships and previously unknown correlations within the data, which can be used to predict and react properly. Used widely along with data mining is the artificial intelligence technique known as Machine Learning (ML), where algorithms are deployed to obtain a predictive analysis from the data.

There are many reasons as to why researchers increasingly try to exploit machine learning for intrusion detection:

- Machine learning offers the potential of detecting zero-day attacks.
- Machine learning can improve both attack detection accuracy and speed.
- Machine learning equipped IDS based on clustering and outlier detection can learn traffic patterns continuously, meaning that they are more effective at detecting slightly varied attack patterns. This in turn also means that these systems require less frequent updates by its developers.
- Traditional solutions need to constantly match each signature with a corresponding IDS database, which is a CPU intensive process.

1.4 Thesis motivation

Despite all of the benefits that machine learning brings, it has caused IDS to increasingly transform into more performance demanding processes. This has mostly to do with the fact that machine learning models require extensive data collection, as well as being trained, both of which are computationally demanding processes. This, along with the resource-limited hardware of IoT devices results in hardships when wanting to execute the complete system locally on an IoT device.

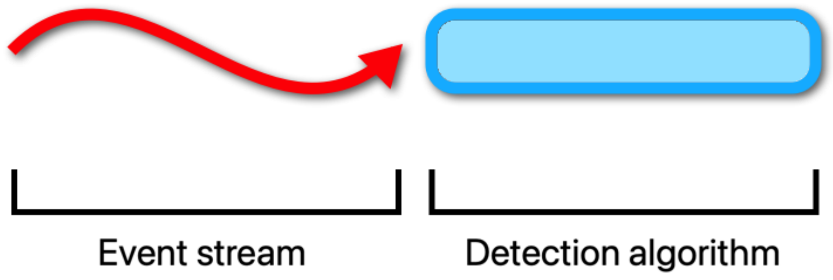


Figure 1: An anomaly-based IDS runs its data-driven algorithm to analyse whether specific units of data are indicating malicious behaviour.

Whenever an anomaly-based IDS runs its detection algorithm, it does so to analyse whether a particular unit of data can be considered malicious or not, a procedure known as ”inference“ monitoring and is abstractly visualized in Figure 1. Each of these runs requires a certain amount of computation to occur, and constantly achieving low execution time is crucial here as the IDS needs to keep up with the incoming data stream from the end node being monitored, each unit of which should be checked for potentially malicious indicators.

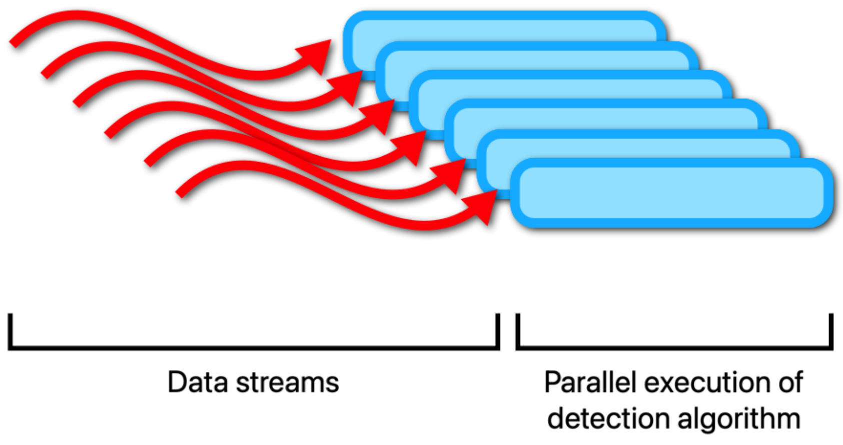


Figure 2: Increasing the number of times the detection algorithm is executed in parallel raises the computational resource consumption of the IDS.

Additionally, the IDS could be monitoring multiple nodes simultaneously, leading to its algorithm having to be executed in parallel as illustrated by Figure 2, further increasing the need for computational resources.

This thesis intends to investigate how scheduling techniques can be utilized to maximize the number of machine learning models that can be run in parallel on IoT devices with a multi-core processor. As recently explained, this would directly enable an IDS to monitor as many end nodes as possible for attack detection. Allowing such large-scale deployment of the monitoring capabilities of a single IDS increases its overall effectiveness against attacks, such as the Mirai Botnet described in section 1.1.1, therefore fulfilling a vital security purpose.

In addition, this thesis will investigate whether the IDS can continue its monitoring tasks even when the need for retraining one of the machine learning models arises. This is a challenging task for IoT devices, as training is typically the most computationally expensive part of machine learning. However, achieving this is crucial for security, since having to pause monitoring tasks in order to retrain a model would result in blind spots where an attacker can operate freely.

Alternatively, one could offload the training to more capable remotely connected machines, however this would entail having to transfer sensitive data via networks. Therefore, executing all parts of the IDS, even the most demanding tasks such as training, locally still remains the most attractive solution.

1.5 Thesis aim

In this section, we will list our research questions as well as our primary and secondary goals for the thesis.

Research questions:

1. How can scheduling techniques, which take advantage of the computational resources found in ordinary IoT devices today, improve the detection capabilities of anomaly-based IDS built upon lightweight and data-driven machine learning algorithms?

2. How does the performance of the scheduling techniques from (1), react to scaling of the problem size? The ability to anticipate a system's reaction towards any particular resource demand is useful knowledge for future users considering to implement the architecture.

Project goals:

1. Develop a basic anomaly-based IDS. This IDS needs to at least be capable of capturing communication traffic, parse that traffic for the extraction of data-points, and thereafter feed the data into detection models that calculate anomalies.

2. Use the IDS developed for the previous goal as a baseline system, then further build upon it to incorporate the performance improvements needed to tackle all the challenges an IDS must be ready to face as described in Section 1.4. The specific techniques of parallelism utilized to achieve this goal, such as scheduling, is the main topic of Section 3.1. Some additional features valuable for security will be incorporated into the architecture as well, and are listed in the beginning of Chapter 4. The developed IDS must be hardware agnostic to IoT devices with a multi-core processor.

3. Deploy the solution on suitable hardware for testing. As this thesis is targeting the IoT environment, it is important that the chosen hardware can be motivated as living up to but not beyond the capabilities of what can be considered as an IoT device. Details regarding hardware specifications can be found in Chapter 5.

4. Carry out a performance analysis of the system developed during the second project goal, and evaluate it against the baseline system in order to draw conclusions on the project's level of success. The metrics utilized for this evaluation, such as throughput, is the topic of Section 3.3.

5. Finally, this thesis project will also investigate whether it is possible to retrain a machine learning model on the IoT device locally, without halting the other

ongoing tasks of the IDS. As was more thoroughly explained in Section 1.4, this is a challenging undertaking; however its accomplishment would hold great value for security.

1.6 Scope

Architecture designing for Machine Learning based systems has become a broad research area, as it is well-known that while these systems can thrive on data today, they are held back by insufficient architecture practices [18]. It is therefore important to emphasize that we are overall only trying to solve a small problem within this domain. An example of this is that we will not go into the details of the detection algorithms themselves, as we would then have to spend significant time on researching and learning about their inner mathematical workings. They will rather instead be treated as black boxes, meaning that the only concern from our point of view is the amount of resources needed for their execution. We are also making the assumptions that there are not any data dependencies between each instance of the detection algorithm, and that they are already trained when the program starts (although they will still need to be retrained during runtime).

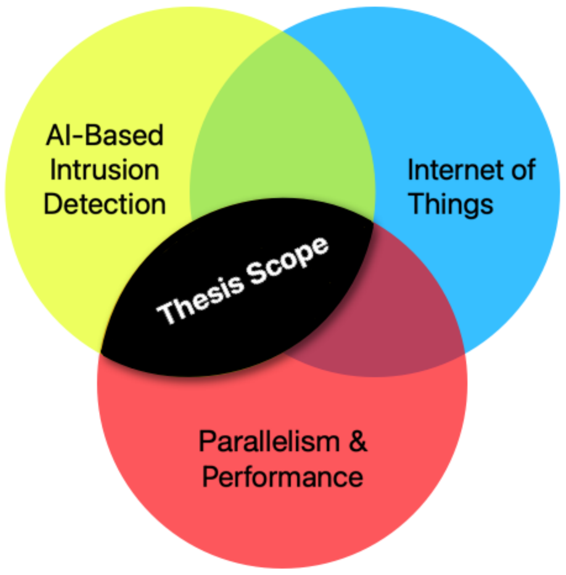


Figure 3: High-level scope overview. The image is inspired by a figure from source [1].

Our solution will act as middleware software, which to a large part incorporates aspects of the fields shown in Figure 3, and there currently does not exist a single set of requirements detailing what such software must support, so we will take it upon ourselves to define them [19]. Firstly, we must limit our solution to work for lightweight data-driven algorithms, as we obviously cannot satisfy all types.

An IDS relying on a complex deep learning algorithm (for example one with several neural network layers), likely requires computational resources that we cannot squeeze out of common IoT hardware despite optimizing. The selection of which detection algorithm to utilize will therefore be heavily driven by a foregoing literature review. Having made sure that the pick suits our parallelization plans as well as the resources available, will in the end make it easier to provide a global performance guarantee.

Lastly, while we know that it is desirable for the architecture to optimally schedule the execution of already existing and well-tested code, we cannot promise a complete application working out-of-the-box since it can not be assured that the architecture will be optimally applied by others. Still, this work shall be valuable for any IDS backed up by machine learning and suitable for resource-limited hardware such as IoT.

1.7 Thesis Outline

The first chapter consisted of an introduction to IDS and the two common techniques for performing intrusion detection (SIDS and anomaly-based IDS). In additional to that, the motivation for the thesis along with its goals were described.

The second chapter consists of related work that aided us in reaching our goals and find answers to our research questions. Here we are mainly looking at how intrusion detection systems work along with scheduling methods to improve performance.

The third chapter goes through important concepts that were briefly mentioned in the introduction, primarily regarding program parallelization and performance evaluation, to then thoroughly detail the theory within these domains. The material written here, especially regarding system architecture and scheduling techniques, is essential for solving the overall problems and therefore has considerable impact on our practical work.

The contributions of this thesis begins from Chapter four, here we go over the architectural logistics for the developed systems. Chapter five thereafter describes what experiments were conducted on our developed solutions and why. This chapter also introduces the machine learning algorithm we chose to use in this thesis as well as the testing hardware.

The results from our experiments are presented in chapter six. A discussion of the methods deployed throughout our thesis work, along with whether the project was successful or not based on its deliveries has been written in chapter seven. Finally, chapter eight summarises the thesis and provides guideline towards future work.

2
Related Work

This chapter begins with providing an overview on how IDS go about when detecting attacks, covering both signature-based and anomaly-based approaches. Efforts to improve the performance of these systems on IoT devices are covered as well, such as the development of a scheduling algorithm.

2.1 Performance deficiencies in IoT security

IoT security has become a rising field as numerous researchers and engineers have been proposing various kinds of IDS in order to deal with cybercriminals. Khraisat et al. [16] have constructed a survey paper where they present an in-depth review of modern IoT IDS, the techniques used in modern IDS, validation strategy, commonly used datasets for IDS and much more.

The authors highlight the challenges when developing an IDS for IoT. Some of the issues discussed in the survey have to do with the IoT devices' restricted hardware and limited computation power. The restrictions results in a lightweight IDS that consists of minimal available security measures to drain as little power consumption as possible. The restricted architecture in IoT devices causes some malware to bypass the attack detection in the proposed intrusion detection systems.

2.2 Anomaly-based IDS for IoT

As the first IDS example, it would be beneficial to find a system made for a similar use case to compare with, namely IoT devices. We believe that such a system was proposed by Hosseinpour et al. [2], who have developed a real-time, distributed and lightweight IDS for IoT based Logistic Systems. This setup is at its core based on a so called "Artificial Immune System (AIS)"-algorithm which allows for an effective combination of edge, fog and cloud computing.

Fog computing can be described as the layer between IoT devices and the cloud. It is equipped with enough computing capacity within routers or gateways to reduce the need of transferring data to the cloud. Fog computing enables low latency attack detection and decision making, close to the source sensors. It also provides the ability to easily share parameters among neighbouring nodes. However, local processing requires that both the algorithms and applications are lightweight and energy efficient.

More advanced analysis and processing are offloaded to the cloud layer with the cost of increased latency. However, these are typically tasks which neither require real-time computation nor constant communication with the fog layer.

The AIS algorithm used in this anomaly-based IDS is made up of the following three parts:

1. The training engine:

   Uses an initial learning data set to train detectors. This is claimed to be a complex process which requires execution on powerful processing units in order for the whole system to perform real-time monitoring. Hence it is offloaded to the cloud. Importantly, this process does not require extensive communication with the edge nodes.

2. The analyzer engine:

   Whenever the now trained detectors report an anomaly, this engine is responsible to come up with an intrusion alert or reject the false positive signals. As this step requires more communication between infected edge nodes and the engine, it is deployed on the fog layer.

3. Detection sensors:

   Each node monitoring the network is equipped with detection logic, creating a collaborative and distributed setup where an attack could be exposed by multiple detectors. In fact, a number of detectors are specifically generated for each type of attack during the learning phase, to increase the overall detection precision. As soon as a threshold is reached, the anomaly is reported to the analyser engine for further analysis after which an alert is possibly triggered.

Moving on to the whole IDS architecture, the authors have here chosen to go for a three-layered structure separated between the cloud, fog & sensor layers as can be seen in Figure 4. The cloud layer is in turn divided into two sub-engines called the clustering and training engine. The clustering engine divides network traffic into normal and intrusion packet flows through unsupervised methods. This data is then used by the training engine to train a set of detectors which are thereafter distributed to devices at the network's edge.

The detectors then monitor the behavior of the edge device they have been distributed to. Whenever an anomaly is detected, its detector will produce a so-called smart data cell for the purpose of further investigation. However, the smart cell is only sent up to the fog layer once a specific threshold has been reached, which the authors chose to set to three detectors. After this, new detectors will be created to detect this particular novel type of attack and distributed to all other devices at the edge.

The downside of this approach however, is that connections failing to breach the threshold are omitted, hence the IDS will not continue to monitor its trends in case a stealthy attack should occur. To deal with this problem, a time dimension is introduced to enable the detection of long-term attacks. Whenever the number of triggered detectors is less than the threshold, the profile of a suspected connection is encapsulated within a smart data cell and handed over to the fog layer. Here, the smart data cell will be aggregated with other smart data cells from other devices where a similar anomaly has been detected, over time making the stealthy attack more visible. Once the number of similar attacks finally exceeds the threshold, it will be analysed as well so that an intrusion alert may be triggered.

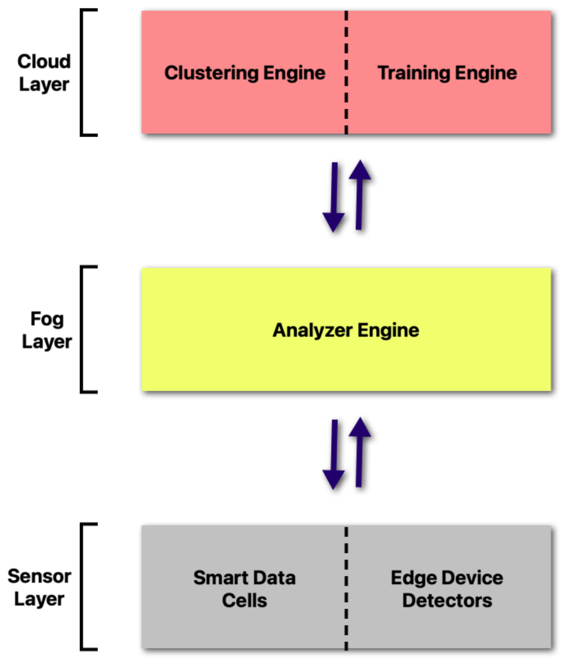


Figure 4: IDS for IoT Logistics Systems - architecture overview. The image is heavily inspired by a figure from source [2].

We should mention why we believe this particular work to have an importance for ours. First of all, it allows us to weight in all the pros and cons between offloading difficult tasks to the cloud, versus executing all computations locally as we are trying to do. Achieving this is not just a matter of being impressive, it is of importance for security as it allows for further avoidance of having to transfer sensitive data over networks. The paper also frequently brings up how fog computing reduces latency compared to cloud computing, and it is an interesting comparison to see if we can provide further reductions by computing directly on the edge device. Lastly, their system has one similarity to ours in terms of the chosen detection model as they utilizes a lightweight AI-algorithm which has been adapted for handling stealthy attacks.

2.3 Anomaly-based IDS - PASAD

As explained in Section 1.3.2, an anomaly-based IDS monitors communication traffic through the use of machine learning. PASAD is an example of this and has been described in the following manner [20]:

> ”a stealthy-attack detection mechanism that monitors time series of sensor measurements in real time for structural changes“

so instead of trying to predict the future, PASAD seeks to decide if present sensor readings are departing from past readings or not, which would be an attack indication.

PASAD is capable of sounding an alarm for significant as well as subtle attack-indicating deviations in the data stream, making manipulations strategically placed within the noise level harder.

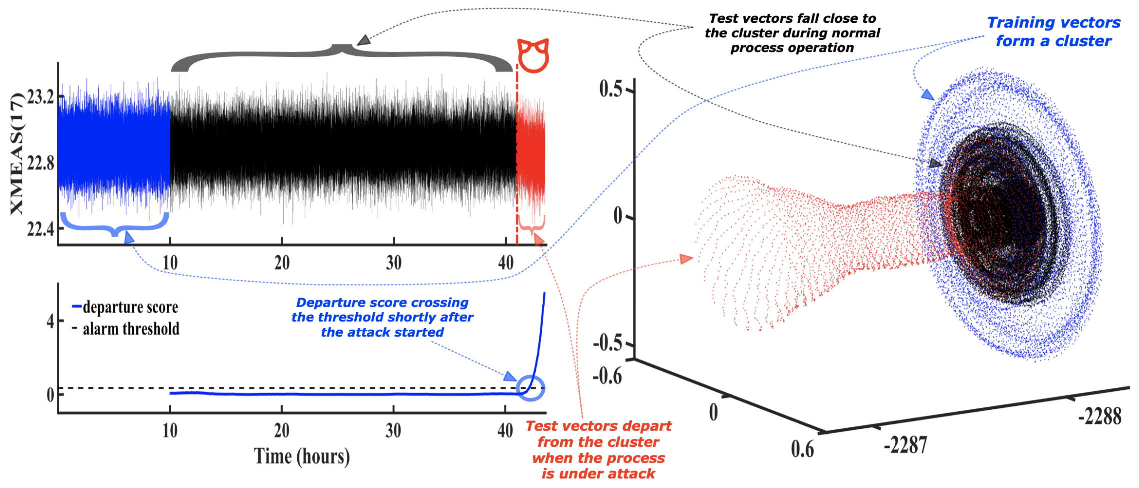


Figure 5: PASAD forms a cluster of training vectors (shown in blue color). Test-vectors falling close to this area is considered as normal non-malicious data (image used with permission).

The logical outcomes of the PASAD machine learning (ML) algorithm can be summarised with the following three steps:

1. The ML-algorithm's training phase forms a cluster of vectors in a special vector space referred to as the signal subspace, signifying normal behavior.

2. Vectors from continuously incoming observations continue to fall close to the cluster during normal process operation.

3. Anomaly detection is made by a single matrix multiplication. PASAD has a lag parameter L that indicates the number of data points needed to calculate a departure score. If vectors from the most recent observations depart from the cluster, it is interpreted as a malicious change to the process. Figure 5 displays an example of this phenomena occurring.

PASAD relates to this thesis work by belonging to the category of ML algorithms that we are aiming our scheduler to be compatible with. More specifically, it has the qualities of being a lightweight algorithm intended for machine-to-machine communication and which looks at sensor data.

2.4 The Midbro component

The lightweight data-driven machine learning algorithm PASAD is agnostic to the specifications of a system as it requires no prior knowledge of the system dynamics and can work with various types of data streams. This exact property was taken advantage of when PASAD was deployed to function in a paper factory [21].

Since PASAD needs sensor data to be served on the fly during real-time operations, the authors built the so called *Midbro component*, which operates in the following manner:

1. Capture traffic by listening to a network interface. Thereafter, parse each packet to filter out the payload.

2. Store the packets in a dynamic FIFO buffer until requested by PASAD. As PASAD is single-threaded and may be busy with other tasks, it is not feasible for it to directly collect values. The buffer had a two thread implementation: one ”producer“ thread which constantly listens to the socket and adds values to the queue, along with a ”consumer“ thread that pops values from the queue in order to be served to PASAD.

The Midbro component allowed PASAD to be extended into a complete and deployable system. It was however built on top of the Zeek framework (due to Zeek's extensibility property), and one major downside mentioned about this is that Zeek is single-threaded. This ultimately limited the amount of advanced data analysis techniques that could be deployed.

It is important to keep in mind that when using a buffer and its consumer thread can not keep up with the data rate, it will fill up. This leads to the detrimental consequence of packets being dropped, and in case this should occur the authors chose to always drop the oldest packets so that at least the data being fed to PASAD is up-to-date.

We view the Midbro component as a source of inspiration to our work when developing our baseline system. This is because Midbro follows a simple producer-consumer setup that can be further enhanced using scheduling techniques. The decision to always drop the oldest packets when needed is to be continued with in our work as well.

2.5 Signature-based IDS - Zeek

Zeek, previously known as *”Bro“* is a signature-based network intrusion detection system which detects intruders in real-time by passively monitoring a network link over which the intruder's traffic transits [3].

During development, three key requirements were set up to ensure the maximum efficiency of Zeek:

- High-speed: Zeek must manage the packet monitoring process correctly even during high volume arrivals, otherwise the queue buffer will drop packets.

- Real-time notification: Avoiding delays is a key factor for detecting attacks quickly.

- Extensible: Allowing for future additions of not yet known attacks to the IDS.

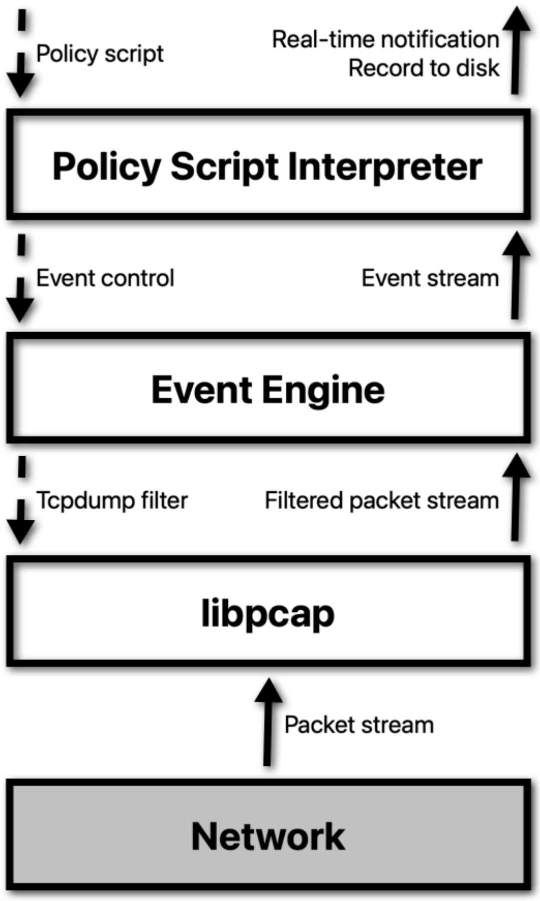


Figure 6: Structure of the Zeek system. The image is a remake from source [3].

The structure of Zeek can be seen in Figure 6, but we will go through the most vital components briefly below:

”**libcap**“ is Zeek’s packet-capturing library which isolates the system from the network link technology, with the added benefit of easing ports to other Unix variants. It also lets Zeek operate on tcpdump save files, making offline analysis possible. The key to packet filtering is the selection of which packets to keep or discard.

Choosing an optimal *”snapshot length“* is also important when collecting traffic, meaning how many bytes of each packet to capture. A smaller snapshot length accelerates processing and avoids loss, hence one should aim to only store the bytes necessary for the particular analysis procedure.
The ”**Event engine**“ reassembles IP fragments and performs several integrity checks to ensure that the packet headers are well-formed. Once the check is concluded, the packet is dispatched to a handler which indicates whether the engine should record the entire packet, just the header, or nothing at all. The event engine is capable of identifying over 70 types of unusual behaviours, such as incorrect connection initiations, checksum errors, packet length mismatches and protocol violations.

Once the engine has finished processing, the ”**Policy script interpreter**“ checks if any events were generated there, if so they are kept in a "first in first out" queue for an appropriate response to be executed for each.

The responses are derived by running written scripts where event handlers are specified. These handlers might generate new events, log real-time notifications, record data to disc or modify internal state for access by subsequently invoked event handlers.

In our work, we will not use a SIDS but we include it here due to the significant research effort that has been devoted to Zeek.

2.6 Scheduling algorithm for CPU load-balancing

Due to the hardware limitations of IoT devices, making full use of all available resources is crucial. Performance improving techniques such as an efficient scheduling algorithm can enable IoT to execute computationally intensive tasks faster. Karsten et al. [22] have proposed a user-level M:N threading system. The system consists of a scheduling algorithm for CPU load-balancing and user-level I/O blocking. The authors have focused on M:N threading for contexts execution. Contexts are described as threads but without preemption and the authors denote context execution as ”freds“. In their system, the number of user-level freds is M and N is the number of kernel threads (threads provided by the operating system).

The proposed scheduling algorithm maps executable freds to available CPU cores in order to provide high performance and efficient resource utilization. To keep all cores busy, the authors have implemented CPU-local queues and a shared queue among cores for fred execution. The authors are able to increase the scheduler’s performance by minimizing CPU cores inactivity. They do so through focusing on even load balancing among CPU cores. If the scheduling algorithm detects load imbalances among processors, it moves freds from overloaded processors to the shared queue in order to achieve equal load balancing. To execute a fred, a processor searches first through its local queue, then the shared queue and lastly by work stealing.

The authors tested their architecture in Linux Ubuntu 18.04 environment and used a 4-socket, 64-core AMD Opteron 6380 with 512 GB RAM. The authors have compared their scheduling against other user-level systems namely Libfiber, Qthreads, Pthreads, Go, Boost, Arachne, Mordor and uC++. The results are split into two sections; scalability and efficiency. The test used 32 cores and executed 1024 concurrent loops which was set up with 32 locks for synchronization. In terms of scalability, the authors' scheduling algorithm along with Libfiber and Qthreads, scale to 25X where each 32-core throughput is normalized by the throughput of single-core execution. Go, Boost and Pthread came closely behind at around 20X while Arachne, Mordor and uC++ did not even reach 15X. To determine efficiency, the identical circumstance were used as in the scalability experiment where the authors have investigated the cost of each loop iteration. When utilizing all 32 cores, the proposed runtime system, Libfiber and Go had the lowest cost at around 50 micro-seconds. The cost for Qthreads, Arachne and Mordor was approximately 100 micro-seconds while Boost and uC++ was considerably higher. Karsten et al. have developed a capable user-level M:N threading system with high scalability and good efficiency, rendering it almost on par with Libfiber and better than most of the other user-level runtime systems.

Importantly for this thesis, is the question of how to utilize scheduling algorithms on IoT, not only from the perspective of performance but also for security. For instance, can it aid in enabling an anomaly-based IDS to run a higher amount of machine learning models in parallel?

3
Theory

This chapter will present the theory necessary to understand and follow our work in this thesis. The theory presented in Section 3.1 and 3.3, is to a large extend based on the book ”Parallel Programming for Multicore and Cluster Systems (2nd edition)“, written by Thomas Rauber and Gudula Rünger [23].

3.1 Parallelizing a program

Parallelization of computer programs is one of the key domains within the field of High-Performance Computing, as it deals with utilizing the computer architecture of a hardware to its maximum capacity. Put simply in the form that is most relevant for this thesis, is that it is the concept of making sure that processor cores can compute independent tasks in parallel, minimizing idleness and communication among cores.

Chip manufacturers have for several years now been producing processors that contain several power-efficient compute units, so called ”cores“ in the same chip hence turning them into ”multi-core“ processors. Often times these cores can access the same memory concurrently, and make each desktop computer a small parallel system. Physical reasons drove development towards multi-core processors, as ever-increasing clock speeds of chips equipped with more and more transistors led to overheating issues.

Constructing software for parallel execution is challenging but important, as developers can no longer expect improved performance to take place automatically with the rise of increased computing power. Several of the most common steps for building a parallel program are to be described through the following subsections.

3.1.1 Profiling

Profiling requires the developer to complete a performance analysis of the sequential program/application in order to identify the most time consuming and computationally intensive functions or tasks. In the case of an anomaly-based IDS, the most time consuming and computationally heavy parts are:

- Data processing: It is rare that raw received data is suited to be fed directly into the detection algorithm without any pre-processing. Exactly how to process the data is many times dependent on the chosen detection algorithm. Processing a single unit of data may be done quickly, however as one needs a considerably large amount of data to train the detection model, data processing as a whole becomes a time consuming part of the system.

- Buffer management: The system needs to hold data until the detection algorithm is ready to receive input, which is a requirement mainly satisfied through the addition of memory resources. The buffer is also vital for avoiding data losses in case a short burst of soared input rate level to the system occurs.

- Detection algorithm: The most computationally expensive part of the IDS, which in turn is split up into two variants: the ”inference phase“, where an anomaly score is calculated and the ”re-training phase“, where the model learns what normal behavior of the event stream is.

3.1.2 Partitioning

Partitioning deals with decomposing computations into smaller tasks and identifying dependencies between tasks. Parallelism of tasks can be achieved through different parallelism methods at different levels of the application. These are called instruction level parallelism, data parallelism, loop parallelism and task parallelism. As this thesis aims to practically carry out its investigations at the user-level, data parallelism and task parallelism are the most suitable methods of the four to explore:

- Data parallelism: parallelizing an operation executed multiple times on elements of a data structure, where each element is independent. The most common use case for data parallelism is when dealing with arrays. A simple example is when constructing a new array where each element is equal to the multiplication of two other array elements. This is also known as single instruction multiple data (SIMD). When the elements are independent of each other, the cores can split work between them where each of them is responsible for a part of the data structure.

- Task parallelism: involves parallelization of independent tasks of a program where these can be executed by different CPU cores simultaneously.

3.1.3 Scheduling

Scheduling is the process of assigning tasks to processes or threads. A scheduling algorithm can either be static or dynamic. Static scheduling maps the order in which tasks are performed before program execution whereas dynamic scheduling maps processes/tasks during run-time. There are three important factors to consider when scheduling tasks, load balancing, information/data exchange between cores and memory accesses.

Load balancing refers to how much work each CPU core performs during execution. In an effective parallelized program, the workload is evenly split between CPU cores. The tasks assigned to a CPU core should involve minimal communication between other cores since information exchange results in more overhead and thus longer execution time. In case the need for such communication still occurs, this may be done via an interconnection network by explicit communication operations. A CPU's execution time of a task is correlated to the number of memory accesses it has to complete and where data lies in the memory structure. Optimizing the tasks for cache locality, meaning that data is located close to the CPU in cache and not in memory, results in less memory access overhead.

Furthermore, threads or processes can be assigned different priorities as a means of resource allocation, since a higher priority results in that element being dedicated a larger portion of the available execution time on the hardware. One could also bring to fruition a form of task prioritization at the user-level, meaning that how often a task is assigned to a thread for execution is proportionate to a set level of priority for that task. This is the approach being taken in this thesis, since keeping the prioritization logic at the user-level eases the changing of priority based on set events during runtime, for example suspicions of an on-going attack.

3.1.4 Mapping

Upon completing the previous steps, the processes/threads will need to be mapped to different cores for execution. In the simplest case, each thread is mapped to a separate core. However, should the number of threads exceed the number of cores, then multiple threads will have to be mapped to the same core to be executed by sharing the available execution time, commonly known as ”time-slicing“.

If the tasks of a program are constrained by data or control dependencies, a specific order when mapping parallel tasks for execution could be required. However, as was stated back in section 1.6, we assume no such dependencies between each instance of the detection algorithm. Because of this, there are not any performance improvements to be expected by executing certain tasks before others, even if it would be fully possible to implement this logic into the IDS software. Therefore, the responsibility of mapping threads to cores will be handed over to the operating system.

3.2 Producer - Consumer architecture

A producer - consumer architecture splits up a system into two co-dependent parts, a producer part and a consumer part. Between them, usually, is a buffer or queue located. The producer would gather data and prepare it to be consumed by the consumer by inserting pre-processed data into the buffer. The consumer would then fetch data from the non-empty buffer and begin its execution.

This type of architecture can practically be achieved through the use of producer and consumer threads. It is a setup which benefits anomaly-based IDS as the incoming flow of packets may arrive at a higher rate than what the machine learning detection algorithm can handle, see discussion of the Midbro component in Section 2.4. Therefore, while the consumer is busy processing, for example to calculate anomaly scores, the producer can keep preparing data units and inserting them into the buffer.

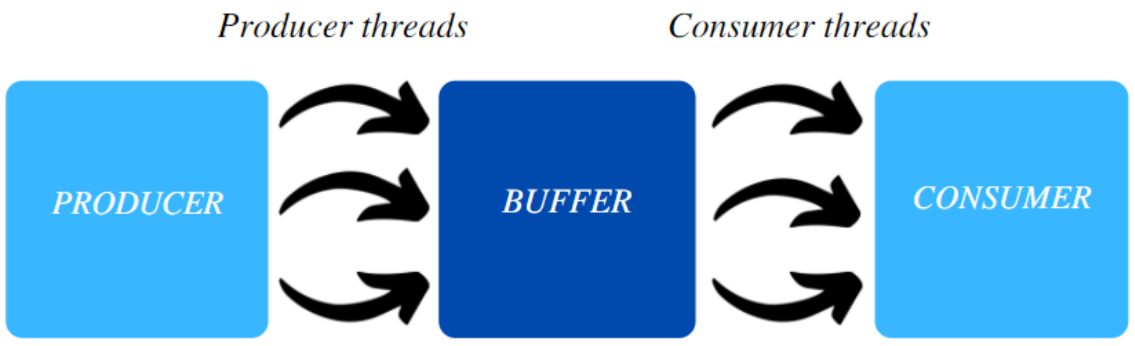


Figure 7: A high-level overview of producer-consumer architectures.

As Figure 7 hints at, this type of an architecture allows for parallel execution where the producer and consumer can work independently as long as the buffer is not empty. Furthermore, due to the likelihood of the machine learning model being more computationally intensive than pre-processing incoming data, in a producer-consumer architecture one can allocate cores either statically or dynamically so that each side gets assigned just enough resources to complete their tasks and thus minimizing CPU idle time.

3.3 Performance evaluation metrics

Performance metrics play a key role when evaluating success rate of the goals set up in Section 1.5, and there are several methods for measuring the effects from this project's efforts to enhance CPU utilization.

$$Speedup(n) = \frac{T^*(n)}{T_p(n)} \tag{3.1}$$

The traditional way to evaluate the effectiveness of the implemented actions is by measuring the speedup or scale-up. Speedup, as shown in equation 3.1 is a quantitative evaluation, where the time of the best sequential implementation ($T^*(n)$) is divided by the parallel time for executing the same task ($T_p(n)$). The total execution time is actually a summation of the following factors:

- ”Local“ computation time: local meaning for each processor.

- Data exchange time: sending/receiving data in distributed memory structures.
- Synchronisation time: accesses in shared memory structures.
- Idle time: processor waiting (no progress) / parallel execution management (overhead).

Scale-up on the other hand, consists of increasing the problem size in order to observe if the execution time stays the same. This would for example mean doubling the amount of work while still achieving the same execution time since the tasks are now run on two separate threads. Scale-up is one of the most common metrics within the domain of High-Performance Computing, since the essence of implementing parallelism is for the purpose of increasing the CPU's effectiveness. This leads to having the ability of running more tasks/processes without overloading the cores.

Moreover, the degree of parallelism as well as the throughput are also useful to measure. The degree of parallelism involves analyzing how much parallelism can be done on a given task and how much parallelism can be exploited in regards to the particular hardware used. Increasing throughput, on the other hand, means attempting to perform more work in the same unit of time instead of executing work faster.

While all performance metrics mentioned in this section are both useful and insightful, they are not all well-suited for evaluating the particular type of system developed for this thesis. Both speedup and scale-up in particular rely heavily on execution times, while an anomaly-based IDS is meant to keep running for an undetermined amount of time. The consumer threads of the producer-consumer architecture therefore do not keep track of whether, for example, all buffers have been completely emptied and hence therefore we are "finished". Incorporating such logic into the system only to measure these metrics would not provide a fair comparison, as the logic of the system has then been significantly altered.

We believe that throughput will be the best method to evaluate the goals of this thesis. As was stated in Section 3.2, the work of the producer thread can be executed faster than that of the consumer. Therefore, if the throughput of the consumer is able to keep up even if the producer's throughput is increased, an overall performance enhancement has been proven.

3.4 Anomaly-based IDS

Anomaly-based IDS provides the possibility of detecting anomalies in network traffic, meaning traffic that behaves differently than normal traffic. It achieves this through a trained machine learning model based on supervised or unsupervised learning.

3.4.1 Supervised learning IDS

Supervised learning IDS consists of a training phase and a testing phase. The training data is labeled as either intrusion or normal so that the system can learn what behavior is considered as normal or malicious. Additionally, feature selection is usually used to make computation more efficient, eliminate redundant features (thus decreasing unnecessary data) and to make the model more general. The essence is to have generalization so that the model does not overfit or underfit.

The learning algorithm observes the training set and attempts to find common patterns i.e. it generalizes patterns so that it can make accurate predictions when faced with new data. A model must have reliable generalization (which refers to the model's capability to handle unseen data) since that is the essential part to detect zero-day attacks. Overfitting and underfitting will drastically affect the model's prediction accuracy of intrusions and might lead to undesired outcomes (such as treating normal traffic as malicious or miss zero-day attacks).

The testing phase consists of testing the system on a set of unlabeled and new datasets in order to observe how the model generalizes new entries as either intrusion or not. Common machine learning models used in supervised learning are models such as Decision trees, Naïve Bayes, Support vector machines and Hidden Markov model [16, 24, 25].

3.4.2 Unsupervised learning IDS

Unsupervised learning deals with unlabeled data. It uses a machine learning model to analyze the unlabeled data to find hidden correlations between them since no output labels are given to the model. One common way of accomplishing this is by grouping similar data into clusters. From an IDS point of view, the collected data is split into groups where the data points in the same group have some correlation. Outliers, or data points located outside the bigger groups are, usually, considered as abnormal behavior and are thus classified as an intrusion. Common machine learning classification models used in unsupervised learning are K-means and Hierarchical clustering [16, 24, 25].

3.4.3 Advantages and challenges

One challenge associated with supervised learning is the creation of a general model. As mentioned above, over-fitting or under-fitting a model drastically impacts the prediction accuracy. The cause for this lies mostly with the training set's data distribution and feature selection, which make it difficult to adapt appropriately when faced with new entries.

Anomaly-based IDS suffer the issue of delivering higher false-positive rates than SIDS. The cause of this is new and harmless activity on the network being classified as an intrusion simply because it deviates from the model's perception of what normal network behavior is [16, 24, 25, 26, 27].

Lastly, a downside of modern Anomaly-based attack-detection techniques overall is that they often involve solving complex and computationally expensive numerical problems, especially with large data-sets in the training phase. Not being able to handle this is a major concern, considering that depending on what the IoT device is in charge of controlling, even our physical safety could be put at risk by an attacker [19].

4 Architecture Design

In this chapter we aim to go through the architecture for our implementation of an anomaly-based IDS for IoT devices in detail. This will be done for both the "baseline" version as well as the fully fledged dynamic scheduler which builds upon it, as was laid out by the project goals in Section 1.5.

In understand the difference between these two IDS versions, one should begin by looking at what set of tasks each system is required to achieve. The baseline system, which primarily builds upon the principles of producer-consumer architectures as described in Section 3.2, has to fulfill the requirements listed in Table 4.1 below.

Hardware agnostic towards machines with two or more cores (scalable)
Capture communication traffic
Pre-process data for analysis
Insert prepared data into buffers until a detection model is ready to receive it
Feed stored buffer-data to detection models for anomaly score calculations

Table 4.1: Baseline system requirements.

The more advanced dynamic user-level scheduler needs to fulfill all of the requirements that the baseline system had, as well as numerous additional ones. All requirements for this system are listed below in Table 4.2.

Hardware agnostic towards machines with two or more cores (scalable)
Capture communication traffic
Pre-process data for analysis
Insert prepared data into buffers until a detection model is ready to receive it
Feed stored buffer-data to detection models for anomaly score calculations
Parallelization through mutiple consumer threads for throughput increase
Dynamic resource prioritization based on determined runtime events
Anomaly detection based on set soft and hard thresholds
Even load balance among consumers

Table 4.2: Dynamic user-level scheduler requirements.

Additionally, model retraining is integrated into the proposed scheduler to fulfill the final goal of this thesis (see Section 1.5). However, running the scheduler while retraining models requires a machine with at least three CPU cores or more.

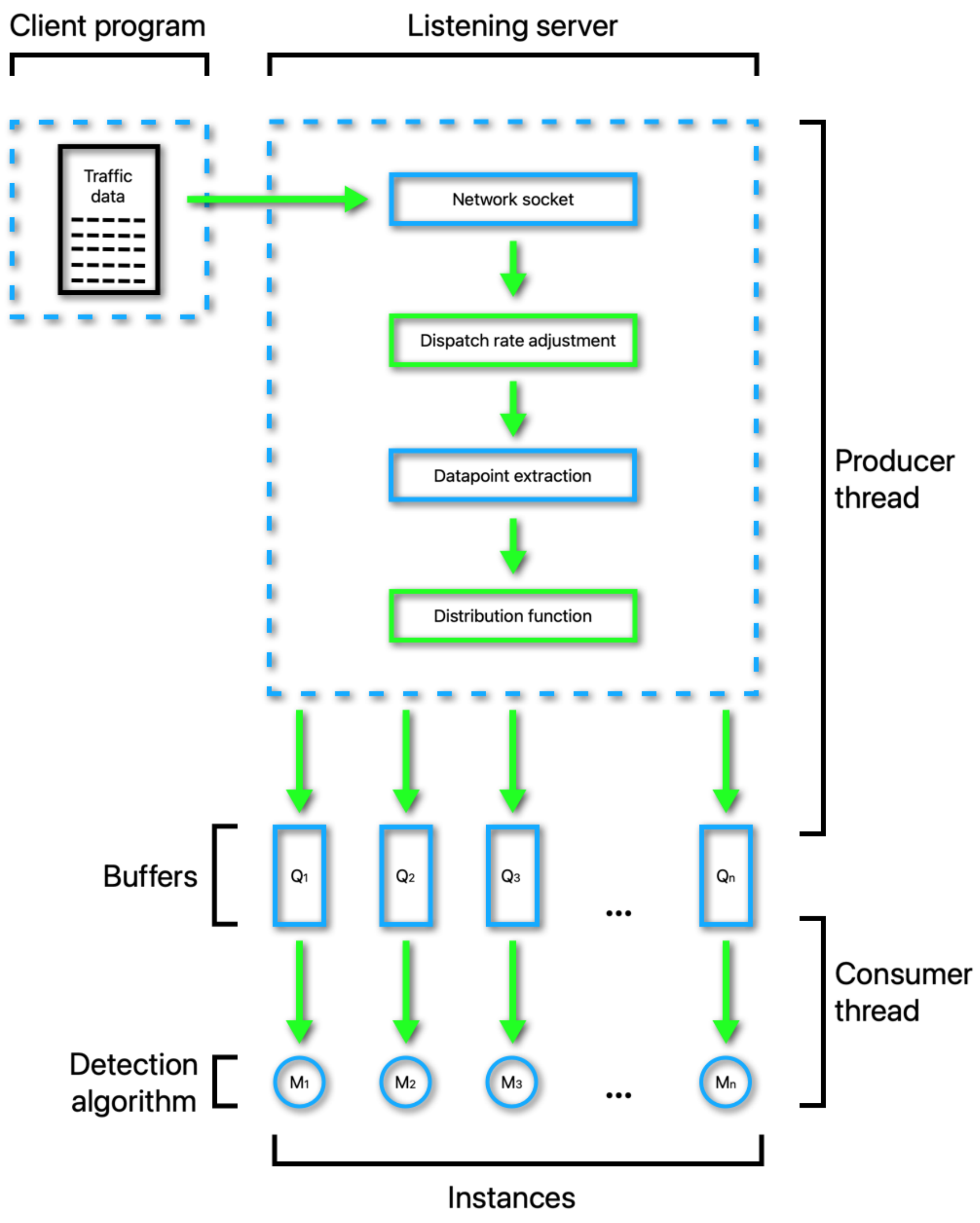


Figure 8: Overview of the baseline architecture. Green boxes indicate elements which have been inserted solely for experimental purposes, and would likely not be present inside a real IDS.

4.1 Developing a baseline

Existing IDS contain numerous additional features that are not relevant for this thesis, resulting in needing to spend valuable research hours into fully understanding them. Therefore, building an architecture which only contains the most vital features allows us to better focus on our main goal, which is running as many detection models in parallel as possible. The baseline system also partly acts as an experiment platform, intended to be used for analyzing the impact of features added later in a comparative manner.

We chose to develop our anomaly-based IDS architecture using C/C++ in a Linux environment. Based on our related work research, we concluded this to be a common way of creating an IDS without any reported performance issues. Additionally, it is also the environment in which we, the authors, feel the most comfortable working in given our academic backgrounds.

Using the Midbro component [21] as a source of inspiration from Section 2.4, the resulting anomaly-based IDS developed is capable of capturing communication traffic, extract selected data-points from it to then be fed into the detection models, as depicted in Figure 8. This basic system is what we consider our baseline architecture. It is at its core a producer-consumer setup as was described in Section 3.2, and hence *"mutex"* locks are necessary in order to ensure mutual exclusion for accessing the common buffers, avoiding race conditions. In the following section, we motivate all design choices further.

4.1.1 The producer

The responsibility of the producer thread is to handle what we have dubbed as the *Listening server* in Figure 8. Here, a socket is opened during run-time initialization to receive data packets sent from the monitored system through a separate client program. This enables the IDS to eavesdrop and capture communication traffic from the socket.

Once a package is in possession of the producer thread, the IDS user now has the opportunity to put an artificial delay on its dispatchment towards the rest of the system. For example, one could here for the sake of realism mimic the timestamps etched within the captured traffic through Algorithm 1. After this, the listening server's following task is to extract a particular data-point for analysis from the packet header (the payload is discarded).

Algorithm 1 Variable packet flow

```
  Declare variables $timeDif$, $timeStamp$
Require: $timeDif = 0$ upon start

  for all incoming packets do
    $timeStamp \leftarrow$ time stamp from packet header
    $sleep(timeStamp - timeDif)$ delays accordingly
    $timeDif \leftarrow timeStamp$ updates $timeDif$ before next iteration
  end for
```

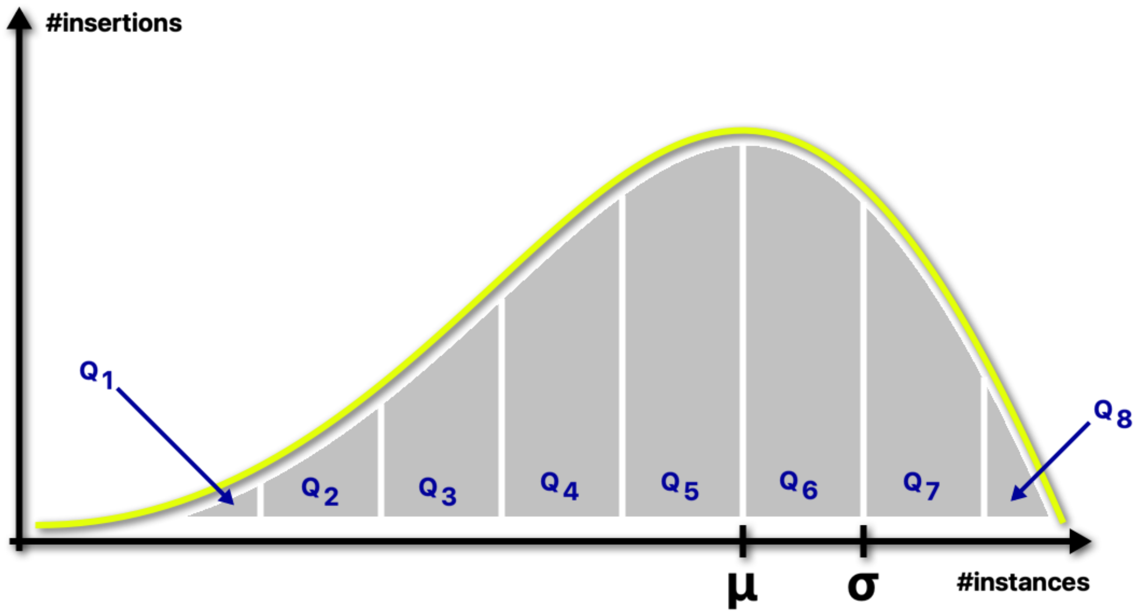


Figure 9: Example of data-points being distributed into the queues for eight different instances, with an arbitrarily set standard deviation and skewed mean.

Lastly, our server needs to insert this data-point into a buffer. Depending on how many instances we are choosing to run, a unique buffer will exist for each one. Exactly in what order the producer distributes each packet in an experimental setting is completely up to the IDS user. One could for example, recreate a more realistic scenario where instances receive different amounts of data through a probability distribution function, see Figure 9 for an example.

Following the case by the developers of "The Midbro component“ described in Section 2.3 [21], we also chose to overwrite/drop the oldest packets in case the buffer gets full, ensuring that the detection algorithm is always fed with up-to-date information. An event like this signifies that the consumer thread is unable to keep up with the data rate, which should always be avoided, so we implemented a missed packets counters to keep track of how often it occurs. Additionally, we implemented counters for the number of insertions and maximum queue size during run-time, as these are insightful metrics to analyse when testing out the system.

4.1.2 The consumer

While all of the aforementioned tasks are being executed by the producer thread, the consumer thread is busy popping elements from the buffers and feeding them to the detection models, as per the requirements in Table 4.1. When a buffer is not empty, each data-point is sent to the detection instance belonging to the queue it was taken from, after which an anomaly score gets produced. For this baseline system, where the consumer side is handled by a single thread, the instances are executed one by one. Meaning that for each instance, at most one data-point is added and one anomaly score is calculated before moving on to the next instance to perform the same actions. If the consumer observes that a buffer is empty, it will move on to the next buffer and inspect if there is any data to be added to the corresponding PASAD instance.

4.2 Developing a Dynamic User-Level Scheduler

By using the knowledge obtained through the theory research from Section 3.1, as well as the related work teachings in Section 2.6, we developed a dynamic user-level scheduler (from here on simply referred to as ”the scheduler“) in order to satisfy all requirements set up in Table 4.2. The scheduler is built as an evolution upon the baseline system, adding features to increase the performance from a security perspective without compromising on it remaining agnostic to the hardware and (to some extent) the utilized detection model.

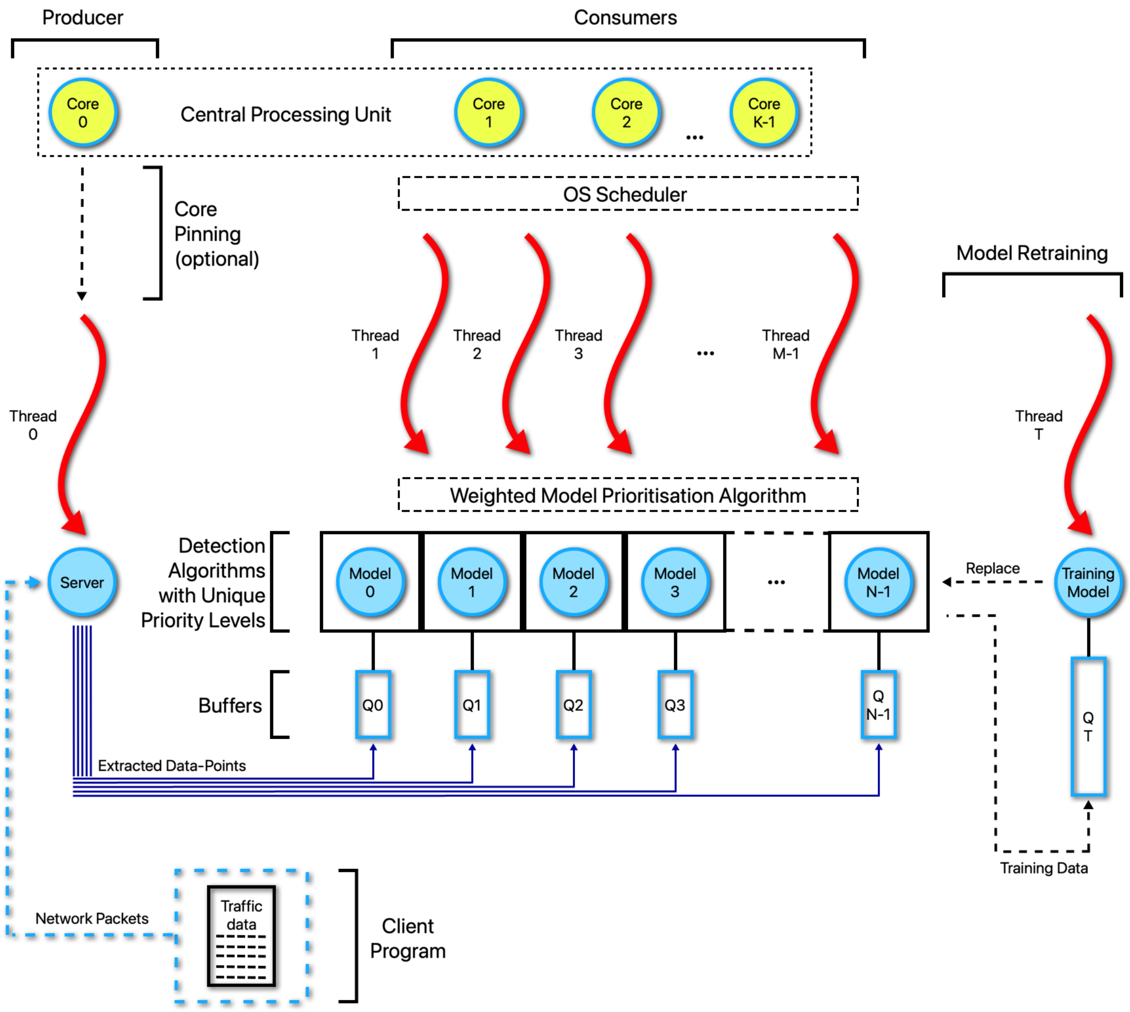


Figure 10: A high-level overview of the multi-threaded scheduler’s architecture.

Figure 10 displays an overview of the scheduler. The producer thread, represented by *Thread 0* in the image above has identical responsibilities as it did for the baseline system, which was gone over in Section 4.1.1. The scheduler achieves parallel anomaly detection by multi-threading the consumer side and placing the detection models in a task pool. An in-depth description of this can be read in Section 4.2.1.

Importantly, the scheduler is designed to adapt to available hardware resources (such as CPU cores and threads). It strives to equally distribute the workload among all consumer threads to ensure scalability as well as being hardware agnostic, which is one of the goals stated in Section 1.5. Note that in Figure 10, the number of consumer threads is less than the amount of detection models.

4.2.1 Dynamic resource prioritization

Since we are detecting anomalies within communication traffic, where each instance of a detection model monitors a specific edge device for rapid attack detection, prioritizing certain models over others is essential for increasing the overall effectiveness of our IDS architecture from a security perspective. What this means in practice is that consumer threads execute models with higher priority more frequently than others. We have chosen three conditions to affects a model's priority, which are listed in Table 4.3 below.

Models buffer size exceeding 80% of their maximum capacity
Models producing suspicious scores (surpassing a ”soft“ threshold value)
Models producing alarming scores (surpassing a ”hard“ threshold value)

Table 4.3: Conditions for initiating a priority change on a model.

The size of a model's buffer during run-time is an indication of how much data it is receiving, and remember, one of the most important tasks for an IDS is to analyze every single incoming data packet in order to detect attacks. Missing packets deprives us of the possibility to conclude if that traffic was normal or malicious, and is therefore an event the system must avoid. Increasing the model's priority level after reaching 80% of the specified buffer sizes will result in it being chosen for execution by consumer threads more often. We have set 80% as the limit in order to give the consumer threads a large enough time margin to detect the priority change before the buffer overflows, since the producer thread will never concern itself with how close the buffers are to overfilling but rather keep inserting data as usual.

Producing suspicious anomaly scores is a clear indication that an attack might be occurring. Because of this it is far more important to allocate more resources towards models producing suspicious anomaly scores than those that are not. Raising the priority level for models producing suspicious scores can allow for the scheduler to trigger alarms earlier, increasing its value from a security aspect.

The highest priority level will be given to any node which the IDS has identified as currently being under attack. Collecting data related to the attack as rapidly as possible is of utmost analytic value in order for the IDS users to quickly find out what caused the attack as well as being able to provide a sufficient response.

4.2.2 Minimizing starvation and overhead

As stated in the previous section, the reason we have associated each detection model with a priority, is to further help the scheduler to decide which model a consumer thread should execute next.

When a consumer thread is ready to perform some work again, it calls firstly on the "Weighted Model Prioritisation Algorithm" (referred to as WMPA) which returns the index for which detection model it should execute next. In its essence, WMPA first calculates the cumulative sum of the models' priorities and stores them in an array we named "sum". The function thereafter calls for a random number r between zero and the sum of all priorities. It then compares r to the cumulative sum of priorities via binary search.

In the example demonstrated in Table 4.4, five detection models have been specified along with their priorities on the row below. If the random number r is thirteen, then that value would be within the range [11,14]. WMPA would return the index for detection model four which the consumer thread then can find stored in an array containing all detection models as depicted in Figure 10.

WMPA's most important trait is that it will favour models with higher priorities. However, this is also the function's biggest cause of overhead as it does not consider if a model is available for execution or not. If the highly prioritized model is already being worked on by a consumer thread, the other consumers will also have a high chance of receiving the index for that already chosen model. Since a model is only allowed to be executed by one thread at the time to prevent race conditions, the other consumer threads will be forced to repeat the call for an index (which could return the same occupied index again). This results in idleness as consumer threads keep asking for indexes, negatively affecting the scheduler's performance.

Detection model (i)	0	1	2	3	4
priority	1	3	1	5	4
sum[i]	1	4	5	10	14
range to choose model i	(0,1]	[2,4]	(4,5]	[6,10]	[11,14]

Table 4.4: An example of how a model is chosen by a consumer thread.

To eliminate this idleness we implemented an "availability"-flag to represent each detection model (zero if available, one if not). If a model's flag indicates that it is not currently occupied by another consumer thread, its priority level and index will be stored as a pair inside a new vector. The recently described function which returns indexes to waiting consumer threads will now do their pick based on what is stored inside this vector. Hence what they return cannot belong to that of an already occupied model.

It should be noted that our implementation can result in low priority models being chosen by consumer threads far to rarely. However, we conclude that this is not an issue since the occurrence of a model remaining at a low priority level is an indication of it neither producing suspicious anomaly scores or receiving much data. Because of this, it is better to focus our resources on the nodes needing it according to the determined priority altering events.

4.2.3 Model retraining

Detection algorithms based on data-driven machine learning models need to be retrained once the model's initial training set is no longer up to date with the incoming data. Ignoring this will likely result in the model producing inaccurate scores that could classify normal behavior as malicious, or the opposite.

However, model retraining is a computationally heavy task even when using a lightweight algorithm, and is therefore commonly offloaded to the cloud or more capable hardware. Investigating a solution for performing training directly on an edge device itself is highly valuable for the research community, even if it were to take a long time to complete. This is because it would aid in keeping sensitive data from flowing through the networks.

An additional, perhaps obvious but still vital security requirement for making the solution serviceable in a real scenario, is that even while retraining is taking place the IDS needs to still be able to perform all its other tasks for continuous attack detection.

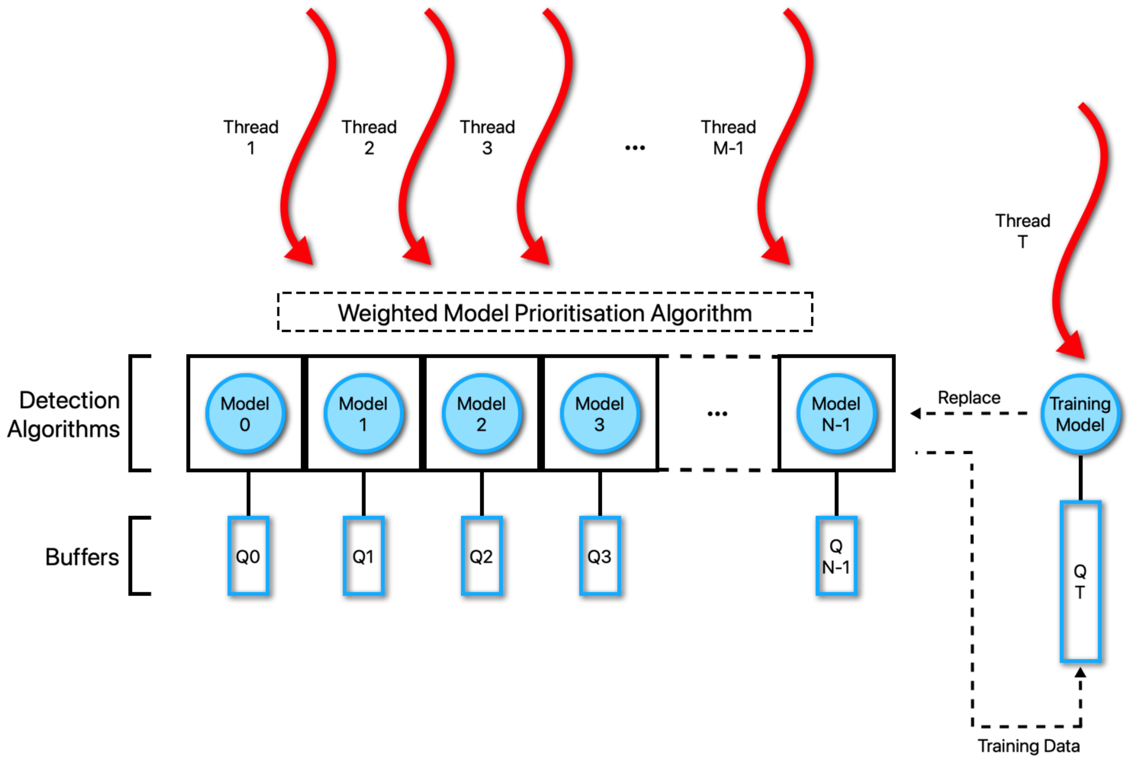


Figure 11: Cropped version of Figure 10.

In Figure 11, we have cropped out only the elements of the scheduler that are needed to understand how our solution for retraining models work. It starts with knowing that the training phase will require a certain amount of up-to-date training data in order to be performed. We have therefore incorporated a variable holding the value for the index of the model that is to be retrained, which is known to all consumer threads. So for example, if detection model of index 2 is to be retrained, then whenever a consumer thread is taking data from this model's buffer in order to calculate an anomaly score, it will also insert a copy of that data into the buffer for a separate model denoted as the *Training Model* in the image above. Training typically requires large amounts of data, hence this model has a larger buffer.

The training model is of the same object type as the other detection models, however it is not stored within the same data-structure. This is to minimize the interference of the training phase with the scheduler's other attack detection functionalities. Once enough training data has been collected, a new thread will be spawned to perform the actual training tasks. Again, the reason for a separate thread (denoted ”Thread T“ in Figure 10 & 11) here is to keep all of the elements needed for training as independent as possible from the rest of the scheduler's important responsibilities.

Once the training tasks are concluded, it is time to prepare for replacing the detection model that was to be retrained. In order to do this safely, the training thread can issue a condition variable indicating the index of the detection model it wishes to replace. The function on which all consumer threads call whenever they want to receive a new index (which was described back in Section 4.2.2) can then use this to make sure that the requested model is only available for the training thread, as soon as any potential ongoing work with that model by a consumer thread has concluded. Once the training thread ”sees“ that its condition variable has been serviced, it can safely replace the detection model with a newly trained one, whereafter it ceases to exist.

This concludes one whole cycle of retraining. What happens right after this is that the index representing which model that is to be retrained gets updated, and the process for retraining another one of the detection models begins.

5 Methodology of Experiments

As Section 4.1 described, the IDS developed for this thesis is fully capable of listening to a port. However, in order to correctly measure and compare the performance of both the baseline system as well as the scheduler system, it is vital to push both systems to their limits. This entails dispatching as much traffic to the systems as they possibly can handle, which can yield inconsistent results when done over the network.

We have in early tests (not included in this thesis) seen the arrival rate of dispatched data being bottle-necked by the local network. To avoid this, we have made the decision to during performance tests, have the producer thread reading incoming communication traffic directly from a file instead.

We have partnered up with Clavister, a cyber security vendor with over 20 years of experience who has provided us with a file containing IP-traffic for use in our experiments, as well as several additional resources to be discussed in this chapter.

5.1 Chosen detection algorithm

Although our scheduler is designed to be fairly compatible with most lightweight data-driven machine learning algorithms, we have chosen ”PASAD“ as the detection algorithm to use during testing. Current implementations of this algorithm are only able to process and analyse a single instance at a time on edge devices (a more in-depth explanation of PASAD can be found in Section 2.3). Some of our motivations for using PASAD are:

- PASAD is a lightweight data-driven anomaly detection model based on machine learning, capable of executing reasonably fast on inexpensive CPU hardware.

- Our supervisors have both been extensively involved in the development of PASAD, therefore possessing deep knowledge on the algorithm’s logic.

- We have access to a header & binary file for the algorithm, ready for software integration.

Since the detection algorithm along with its inner workings is a black box from our point of view, following the scope for this thesis as was set up in Section 1.6, we are only able make the following function calls to use it:

- One unique function call is needed for adding a data-point to each PASAD object's own test set. The detection algorithm uses this test set to calculate anomaly scores. Recall from Section 2.3 that each PASAD instance requires L points of data before it can calculate its first anomaly score, whereafter it can calculate a new score for each added data-point. Note that the test set is a separate data-structure from the buffers our scheduler initializes for each detection model.

- Another function call calculates and returns the anomaly score, using the test set built through calls to the aforementioned function.

- A third function also takes in a data-point but adds it to a training set instead of the test set. In the version of PASAD provided to us, 30,000 data-points have to be added before a model can be fully trained.

- Finally, two additional functions are called consecutively in order to execute the training of a model, using data from the training set.

5.2 Hardware devices

We tested our solutions on two different hardware devices, both of whom we confidently feel follows the scope for this thesis in terms on focusing on IoT devices, as laid out in Section 1.6. These devices are the "Raspberry Pi 4 Model B" and the "NVIDIA Jetson Nano". The reason as to why we have chosen to utilize these specific units is because of their interesting hardware differences, especially in terms of their memory architecture. Both of them are also equipped with quad-core CPU:s with support for four threads, making it easy for us to deploy our multi-threaded systems.

5.2.1 Raspberry Pi 4

The processor inside the Raspberry Pi 4 is equipped with 80KB Level 1 (L1) cache per core, along with 1MB L2 cache which is shared among the cores and a single memory channel to access the RAM. Its maximum CPU clock speed lies at 1.8 GHz, and the memory bandwidth is approximately 4.4 GB/s [28]. For a more extensive specification description, we refer to the official data sheet [29].

Processor	Quad-core 64-bit ARM-Cortex A72 (Broadcom BCM2711) at 1.5GHz
Threads	4 (1 per core, no hyperthreading support)
Memory	4GB LPDDR4-3200 RAM
Graphics	VideoCore VI 3D Graphics
Storage	Micro-SD card slot for loading operating system and data storage

Table 5.1: Specifications for the Raspberry Pi 4 model B.

In this unit's memory hierarchy, each of the four processor cores have access to their own level 1 cache. Whenever any needed data is not stored there, they are forced to go searching further down among the shared memory components such as the L2 cache, RAM or in the worst case main memory (Micro-SD card). This setup is visualized in Figure 12.

Ubuntu Server version 22.04 was loaded on as the unit's operating system. Remote access is enabled through SSH.

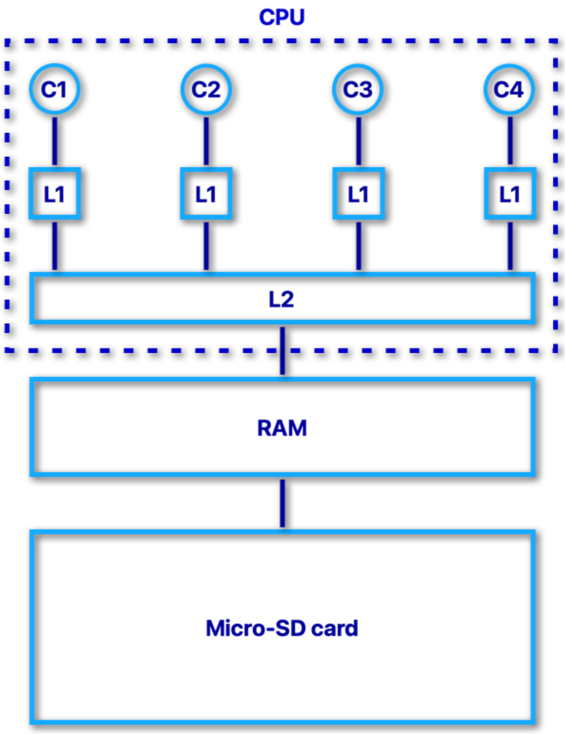


Figure 12: Memory structure of the Raspberry Pi 4.

5.2.2 Nvidia Jetson Nano

Just as the Raspberry Pi, Nvidia Jetson Nano's processor is also equipped with 80KB Level 1 (L1) cache per core, with 2MB L2 cache shared among all cores and dual memory channel access to the RAM. Its maximum CPU clock speed lies at 1.47 GHz, along with a memory bandwidth of 25.6 GB/s. For a more extensive specification description, we refer to the official data sheet [30].

Processor	Quad-core ARM Cortex-A57 MPCore processor
Threads	4 (1 per core)
Memory	4GB 64-bit LPDDR4 RAM
Graphics	NVIDIA Maxwell architecture, 128 NVIDIA CUDA cores
Storage	Micro-SD card slot for loading operating-system and data storage

Table 5.2: Nvidia Jetson Nano

In this unit's memory hierarchy, each of the four processor cores have access to their own level 1 cache. Whenever any needed data is not stored there, they are forced to go searching further down among the shared memory components such as the L2 cache, RAM or in the worst case main memory (Micro-SD card).

Ubuntu 18.04.6 LTS (GNU/Linux 4.9.253-tegra) is loaded on as the unit's operating system with Nvidia's Jetson Nano Developer Kit, as is the default configuration provided by the manufacturer. Remote access is enabled through SSH.

5.3 Experiments setup

In order to fulfill all project goals set up in Section 1.5, we prepared several important experiments listed in Table 5.3 below. Many of these experiments are performed both on the baseline system and the scheduler, as well as on two separate hardware devices in order to conduct proper a comparison and evaluation afterward. The upcoming sections of this chapter will provide a more detailed walkthrough regarding the purpose of each experiment as well as their setup.

Maximum throughput measurement
Evaluation of model prioritization
Evaluation of producer-consumer relationships
Performance impact of PASAD's inference-related functions
Performance impact of scheduler features
Runtime measurement
Scale-up measurement

Table 5.3: Experiments to be conducted.

In order to perform the experiments above, a number of architecture adaptations had to be made, affecting both the scheduler and baseline system. The first of these was to decide on how the producer thread should dispatch data-points. We decided to go for insertion through bursts, in the manner described by Algorithm 2 below, instead of replaying the time-stamps using Algorithm 1. The producer will also perform its distribution uniformly, meaning all models are to receive the same amount of data. These alterations are necessary as the *burstSize* variable enables us to directly change the rate of incoming data to all detection model buffers. *burstSize* is the amount of packets dispatched before waiting one second for the consumers to have an opportunity of emptying their queues.

Algorithm 2 Fixed packet bursts

```
  Declare variable $burstSize$
Require: $burstSize > 0$

  while end of traffic file not reached do
    dispatch 1 data-point to consumer thread
    $i \leftarrow i + 1$
    if $i \geq burstSize$ then
      sleep(1 second)
      $i \leftarrow 0$
    end if
  end while
```

Secondly, since our traffic data file might not contain enough packets to keep the experiment running for a sufficient amount of time, we allow the producer thread to continuously loop through the file a number of times as determined by the user. This functionality is also very important when *burstSize* is large, as that decreases the amount of time it takes to go through the file.

Thirdly, we decided on a fixed maximum buffer size of 10,000 for all detection models. This is an arbitrarily set limit, as the scope for this thesis is not concerned with the actual size of the buffers as long as they are not increasing indefinitely, causing unpredictable memory usage. In a real-life scenario however, a lengthier consideration would have to be taken when deciding on the maximum buffer size. This is because unlike in our test environment, unpredictable bursts of faster incoming communication traffic can occur there. In such a scenario, having the buffer as temporary storage of data-points acts as a safety margin for not loosing packets in case the consumer threads cannot keep up with the temporarily higher rate.

Fourthly, the scheduler will be tested with four threads in total, one producer and three consumer threads. This is because even though the scheduler is developed to be hardware agnostic, both of our test devices are equipped with quad-core CPU:s as explained in Section 5.2. We aim to avoid time-slicing of the producer thread in order to guarantee the intended incoming data rate for each given test, therefore not executing a higher number of threads than there are cores. When running the scheduler while also retraining models, one consumer thread will be exchanged for a separate training thread, following the logic for training as described in Section 4.2.3.

Fifthly, we did not take advantage of the opportunity to control the mapping of threads for execution on specific cores as described in Section 3.1.4, as we have not witnessed any performance gain by doing this (test not included in paper). This task is therefore handed over to the OS.

Finally, it should be mentioned that the results from most of our experiments, if not all, should be taken with a grain of salt as they inevitably contain some margin of error. This is because various processes running in the background inside the operating system could be affecting the result between test. We conduct different methodologies to reduce these margins of errors. In certain cases, like when we measure the maximum throughput, we are required to repeat the same test several times while slowly iterating the input rate until we find what the system can maximally handle. In other cases however, such as in the model prioritization evaluation, averaging the results from three test runs suits better.

5.4 The steady state phenomena

An important concept to grasp before evaluating any of our experiments, is the assumption that producer-consumer architectures (see Section 3.2 for more details regarding this architecture category) which both our baseline system and scheduler basically are, can reach a so-called *steady state*. To explain this phenomena in the easiest way, we will focus on the baseline system in this section.

PASAD #0		
	Insertions	125850
	Max buffer size	211
	Missed packets	0
PASAD #1		
	Insertions	787440
	Max buffer size	2557
	Missed packets	0
PASAD #2		
	Insertions	1981020
	Max buffer size	6699
	Missed packets	0
PASAD #3		
	Insertions	1977360
	Max buffer size	6708
	Missed packets	0
PASAD #4		
	Insertions	788550
	Max buffer size	2571
	Missed packets	0
PASAD #5		
	Insertions	120990
	Max buffer size	200
	Missed packets	0

PASAD #0		
	Insertions	377550
	Max buffer size	250
	Missed packets	0
PASAD #1		
	Insertions	2362320
	Max buffer size	2599
	Missed packets	0
PASAD #2		
	Insertions	5943060
	Max buffer size	6745
	Missed packets	0
PASAD #3		
	Insertions	5932080
	Max buffer size	6704
	Missed packets	0
PASAD #4		
	Insertions	2365650
	Max buffer size	2580
	Missed packets	0
PASAD #5		
	Insertions	362970
	Max buffer size	264
	Missed packets	0

Table 5.4: Reaching a steady state, 30 versus 90 loops.

As has previously been explained, the baseline utilizes two separate threads (one producer thread and one consumer thread, see Figure 8). For this duality to work well, both threads need to achieve balanced results. Specifically, there are two scenarios where this is not the case; one where the producer is faster than the consumer and the second being that the consumer is faster than the producer. If the producer is faster than the consumer, this will eventually lead to packets being missed. On the other hand, if the consumer is faster than the producer this will result in underutilized hardware resources as idleness will occur on the consumer side. Both cases are considered bad from the performance aspect.

PASAD #0		
	Insertions	251700
	Max buffer size	556
	Missed packets	0
PASAD #1		
	Insertions	1574880
	Max buffer size	5290
	Missed packets	0
PASAD #2		
	Insertions	1989182
	Max buffer size	20000
	Missed packets	1972858
PASAD #3		
	Insertions	1989305
	Max buffer size	20000
	Missed packets	1965415
PASAD #4		
	Insertions	1577100
	Max buffer size	5292
	Missed packets	0
PASAD #5		
	Insertions	241980
	Max buffer size	504
	Missed packets	0

Table 5.5: Breaking the steady state.

Table 5.4 provides a simple way to analyse whether the consumer thread manages to keep up with the producer. In this case each sub-table displays an example of six PASAD models being run on the baseline system, however the traffic file is looped 30 and 90 times respectively. Looping the file 30 and 90 times took roughly five minutes and thirteen minutes respectively until the program execution finished. Each loop iteration consisted of 192,707 data-points dispatched for analysis by the producer at a rate of 20,000 data points per second, while the buffer size for each PASAD instance was set to hold up to 10 000 data-points. ”Max buffer size“ indicates the maximum buffer size reached during runtime and it can clearly be seen that it remains roughly identical for each PASAD instance despite the variation in execution time. This points to the fact that the consumer thread was able to keep with the producer and thus preventing the buffers from growing ever larger. This is what is meant when we say that the system has reached steady-state.

However, this will not stay true under all circumstances. Because while the consumer thread will be able to keep up to a certain level of input rate, at some point the influx of data will be too large for it to manage. When this occurs the steady state "breaks". Table 5.5 displays the broken steady state where the producer thread was instructed to dispatch 40,000 data-points per second, which proved too much for the consumer to handle. This resulted in the buffers of PASAD model 2 & 3 overflowing and caused a large quantity of missed packets. This occurred despite doubling the maximum buffer size to 20,000 elements.

We emphasize once more the fact that missing packets always hurts the accuracy of our IDS, as potentially malicious data can get overwritten. This gives us a fixed upper limit on how much data our system should be given per unit of time to process when running a certain amount of models.

5.5 Maximum throughput measurement

Evaluating the performance of the scheduler is tricky business, due to it including multiple different data-structures, features and parameters that can be altered as one might understand simply by glancing at Figure 10. However, after having consulted with all partners involved with this thesis, we agreed that throughput would be a crucial metric for determining this project's degree of success.

$$Throughput_{Scheduler} > Throughput_{Baseline}$$

The core concept is quite simple and summarized through the equation above. If the scheduler can process data incoming at a higher rate than what the baseline system is capable of (without missing packets!), it has performed better in terms of throughput. An alternative description would be to say that we are scaling up the problem size without granting the system additional execution time (see Section 3.3 for a more in-depth explanation of throughput along with other metrics).

An increased throughput holds value from a security perspective since faster processing of the incoming data means that attacks might be discovered earlier. It also provides an overview for users regarding what resources are available to them when using the IDS, which could for example be taken advantage of by monitoring more nodes simultaneously.

The experiment is practically performed by tweaking the rate of incoming data (see $burstSize$ variable from Algorithm 2) too stress the hardware and see how much each system can handle before missing packets, then repeating this with a varying number of running detection models. In case the consumer thread(s) are managing to keep up with the rate of incoming data, the throughput, meaning the rate at which the tested system produces anomaly scored should be equally large. This is an assumption we are making based on the steady-state phenomena explained in Section 5.4. We determined that if the system cannot surpass 50 loops of the traffic file without missing packets, it has failed the test and the program will be terminated immediately.

5.6 Evaluation of model prioritization

One of the features built into the scheduler to increase its value from a security aspect, is that it is able to dynamically allocate more resources towards nodes needing it based on their level of priority (see Section 4.2.1 for a more extensive explanation). As an additional security metric, we have devised an experiment in order to quantify solely the effect this particular feature has.

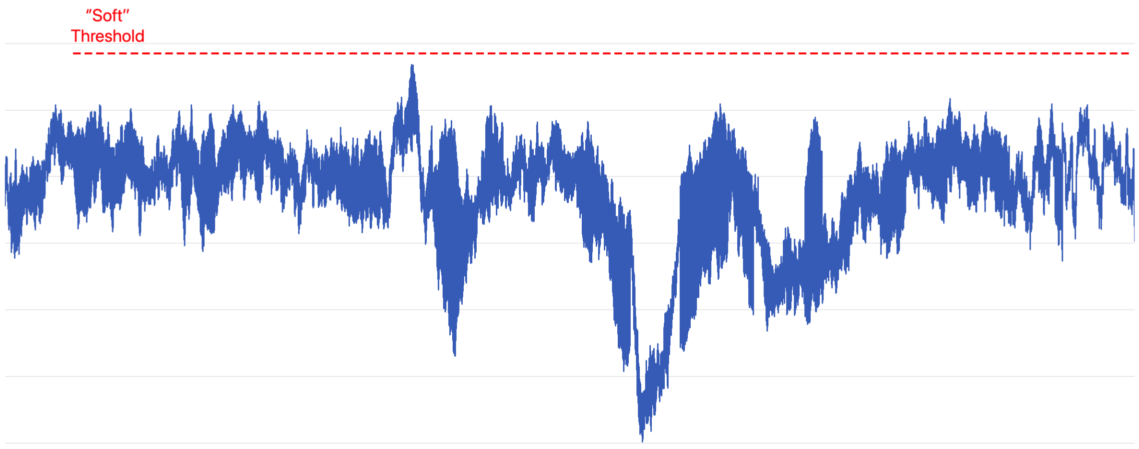


Figure 13: Both the soft and hard threshold (which lies above the vertical range of this image) are defined to be located above the anomaly score values produced from the traffic file data.

In Figure 13, the values of all the produced anomaly scores after having iterated through the traffic file once are printed out. A vertical axis for showing the particular values of these scores has been left out, as they are of no significant concern to this experiment.

The way this experiment will be conducted is by intentionally causing just one of the detection models (model with index zero was chosen) to breach both the soft and hard threshold values (the meaning of these thresholds is explained in Section 4.2.1). In order to make sure that no other model causes a breach we defined the thresholds to be higher than the scores produced from the traffic file data as indicated in Figure 13. We then dispatched a different stream of data into the buffer for the model that is intended to breach the thresholds.

Algorithm 3 Purposeful threshold breaches

```
$modelTurn$ is index of the model to receive data
The value of $X$ causes anomaly scores above soft threshold
The value of $Y$ causes anomaly scores above hard threshold
Variable $loops$ correspond to current amount of traffic file iterations

if $modelTurn = 0$ and $loops < 30$ then
  dispatch $X$ as data-point to model 0
else if $modelTurn = 0$ then
  dispatch $Y$ as data-point to model 0
else
  dispatch data-point from traffic file to model of index $modelTurn$
end if
```

Remember that the producer thread iterates and inserts data-points to all buffers uniformly as was determined in Section 5.3. Whenever it is time to dispatch a unit of data towards model ”0“, the producer will then instead dispatch data-points of a higher value which causes this model's scores to remain above the soft threshold, hence making the scheduler increase its priority level (see Algorithm 3 above). After having run the program for a certain amount of file loops (30 loops was chosen arbitrarily), the producer will then start to insert even larger values, causing model 0 to also breach the hard threshold, which the scheduler interprets as an alarm.

We then measure the time it takes until this alarm is detected, both on the scheduler with and without the model prioritization feature activated, as well as on the baseline system. The effect that a higher level of priority has on the buffer sizes will also be analysed, using the counter ”maximum queue size“ introduced in Section 4.1.1.

5.7 Evaluation of producer-consumer relationships

To gain further insight regarding how our architectures react to scaling, we devised two experiments for observing the relationship between the producer and consumer. Specifically, how they try to gain access to the mutex locks guarding the buffers between them.

In the first of these experiments, we built a very simple program where the producer thread was eliminated. This meant that all the buffers were pre-loaded with insignificant variables during runtime initialization and a single producer thread could freely access them all without needing to compete for a mutex lock. We then altered the number of buffers while keeping the sum of all data stored inside them the same, to then measure the time it takes for the producer thread to empty them all.

In the second experiment we brought back the producer thread while still keeping the program very simple. Simply put, the producer thread was to push in total one million insignificant variables into all buffers as fast as it could, while the consumer thread kept popping them out whenever it encountered a buffer that was not empty. To avoid race conditions however, each thread can only perform their task when gaining the mutex lock for a buffer, and we again measured the total execution time of this program when altering the number of buffers.

5.8 Performance impact of PASAD's inference-related functions

Even if we are viewing the utilized detection algorithm as a black box, it is important to know at least what it costs us in terms of execution time overhead since that is affecting our results in other experiments as well. We measured how long each call to the PASAD functions for producing anomaly scores varied on average when scaling the number of detection models running simultaneously. The functions needed for retraining PASAD models are of less concern to us however, since as was explained in Section 4.2.3 it is considered a non-issue for the scope of this thesis whether the training phase takes a long time to complete.

5.9 Performance impact of scheduler features

In Sections 4.2.1 and 4.2.2 we discussed dynamic resource prioritization as well as WMPA. The purpose of the experiment is to get some insight into how much overhead features like this are actually adding to the system. We will therefore conduct throughput measurements in the same manner as in Section 5.5, however on another version of the scheduler where all features except for having multiple consumer threads are removed. Meaning that this version is actually identical to the baseline system, except for the fact that multiple consumer threads are iterating through their own part of the data-structure that contains all detection models.

5.10 Runtime measurement

A valuable performance metric is to observe how the difference between the start and termination time varies on our architectures. Traditionally, one would do this by calculating the ”speed-up“, which as explained in Section 3.3 is produced by dividing the sequential (program run on one thread) version's total execution time with that of the parallelized version.

However, this exact evaluation could not be performed since even the baseline system is executed using two threads, and would have to be altered severely in order to function on a single thread. We therefore decided to instead measure the difference between our two-threaded baseline system against the four-threaded scheduler system, using the following equation:

$$Runtime\ improvement = \frac{Execution\ time\ baseline\ system}{Execution\ time\ scheduler\ system}$$

Where a quota greater than one is indicative of an improvement. We argue that a perfect improvement here would be if the scheduler system displays a three-fold improvement over the baseline system, since we have shifted from executing only one consumer thread to having three of them, while remaining with the single producer thread. The metric can therefore act as an indication on how well the parallelization (or threads) have been implemented. Regarding this metric's importance from a security analysis perspective, the reasons are alike those stated for throughput in Section 5.5.

The experiment will be performed by looping the traffic file 50 times just like in the throughput experiment from Section 5.5, and we will measure the program's total execution time. We make the decision of how many models to run in this test, as well as the data input rate following an analysis of the throughput experiment's results. That is because this test should be performed with a large number of models as the results then become more valuable for any reader considering to follow our architecture approach, and we need to know what data input rate each system can handle with that amount of models.

5.11 Scale-up measurement

Scale-up is another valuable performance metric to observe differences in problem size for the architectures while the execution time remains identical. The general purpose of measuring scale-up was described earlier in Section 3.3 but in the case of the thesis, scale-up would indicate how many more data-points the scheduler is able to perform anomaly detection on compared to the baseline during the same time period.

We utilize the following equation to measure scale-up:

$$ScaleUp(T) = \frac{N}{M}$$

Where T is the execution time for the baseline system to perform anomaly detection using M data points and N is the number of data points that the scheduler accomplishes to perform anomaly detection on during the same period T.

For the same reasons as in Section 5.10, the number of models to run in this test, as well as the data input rate is determined following an analysis of the throughput experiment's results.

6
Results

In this chapter, the results from our experiments are presented in the same order as they are listed within Table 5.3. Certain experiments were carried out on both hardware devices to enable a comprehensive comparison, while others only required a single device to provide the needed results. Observe that in some graphs, ”anomaly scores“ have been denoted as the more general term ”departure scores“.

6.1 Maximum throughput

In this section we will present the maximum throughput test results. The results for the Raspberry Pi can be seen in Section 6.1.1 and the results for the Nvidia Jetson Nano can be seen in Section 6.1.2. A comparision between the two devices is presented in Section 6.1.3.

Important to mention lastly, is that there are three regions of interest shown in the performance graphs in Figures 14, 15, 17 and 18. The first interesting region is the linear relationship occurring from one to six PASAD models, the second is the exponentially decreasing region from six to 20 models, and lastly the plateau after 25 models.

6.1.1 Raspberry Pi 4

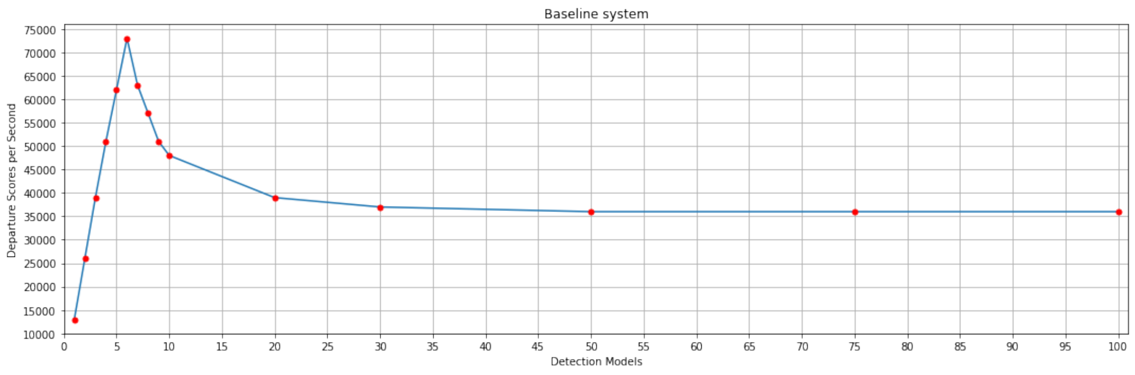


Figure 14: Throughput testing of the baseline architecture. In this case, only a single consumer thread is responsible for inference monitoring all models.

In Figure 14, the performance of our baseline architecture, described in Section 4.1, was tested. On the horizontal axis are the number of PASAD instances that were running. The vertical axis represents the system's throughput, meaning the number of departure scores it produces per second.

It is crucial to note here that as long as the producer and consumer threads can keep up with each others work, the throughput should be equal to the input rate, meaning the number of data-points being inputted to the system by the producer. Again, each PASAD model can be seen as an end node being monitored, hence having the capability of running a higher number of PASAD instances is preferred.

What the blue line shows is the maximum amount of throughput the system is able to produce, in relation to a certain number of PASAD instances running. If more data-points per second than what the red dots indicate are dispatched into the buffers, the consumer thread will not be able to keep up with the producer's speed anymore.

For instance, when running ten PASAD models, the throughput the system can achieve is roughly 47,000 anomaly scores per second. Exceeding this leads to the buffers being filled faster than they are emptied, and in turn to missed packets due to buffer element overwrites. As explained in Section 4.2.1, missing packets should always be avoided in an IDS, and therefore we have chosen this event as the system's limit.

From Figure 14 we can see a linear increase up to six PASAD instances which is also where the system reaches its peak throughput. We can see that the Raspberry Pi seems to hit a "performance wall" at around 6 models whereafter its performance starts to decrease exponentially and then flattens out when running more than 20 PASAD instances.

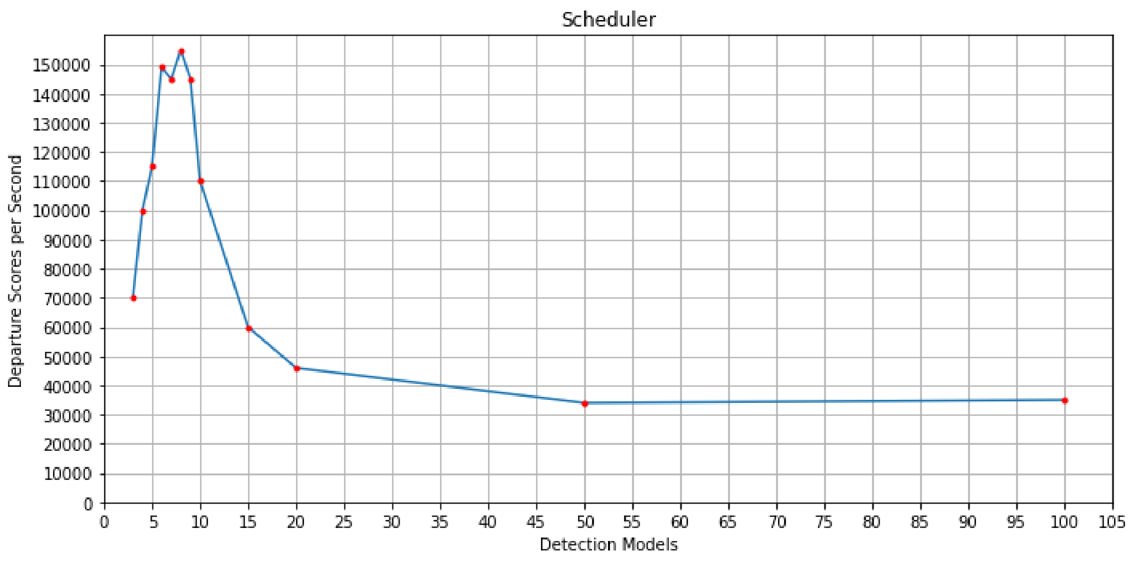


Figure 15: Throughput testing of the dynamic user-level scheduler when utilizing all the cores.

Similar to the setup for the baseline, the vertical axis in Figure 15 represents the dynamic user-level scheduler's throughput. The difference here, compared to the baseline, is that we make use of all the available cores in our hardware by allowing multiple consumer threads to work. Again, the red dots on the line indicates the maximum number of anomaly scores the system is capable of producing before missing packets, along with the number of PASAD models specified on the horizontal axis.

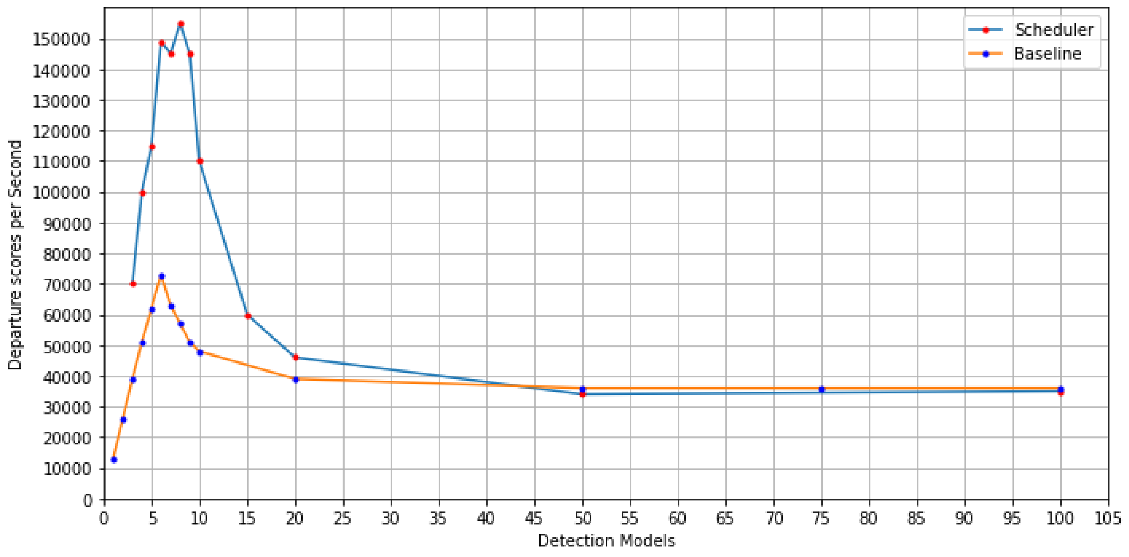


Figure 16: Comparison between the baseline system and scheduler, running on the Raspberry Pi 4.

As demonstrated by Figure 16, the scheduler achieves a significant performance boost when running just a few PASAD models in parallel. The scheduler has it peak at eight PASAD instances compared to six at the baseline. Additionally, we see a radical decrease in performance going beyond eight PASAD models for the scheduler to the point that it performs almost identical to the baseline.

Lastly, re-training PASAD models in parallel while other models are used for running inference on the Raspberry Pi 4 proved to not be possible. Re-training could only be performed successfully by completely freezing the scheduler's inference tasks.

6.1.2 Nvidia Jetson Nano

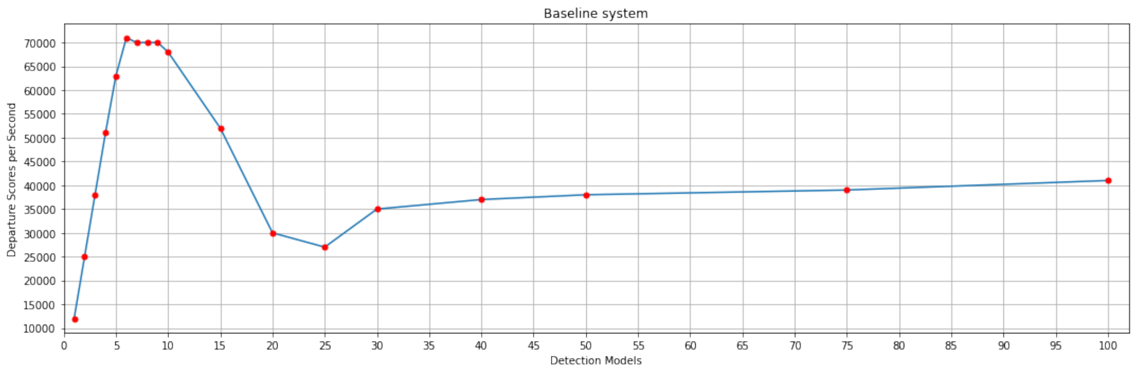


Figure 17: Baseline architecture's throughput performance.

Identical to the baseline experiment on the Raspberry Pi, Figure 17 visualizes the performance of the baseline architecture on the Nvidia Jetson Nano. As shown in the graph, the baseline's maximum throughput performance occur with six to 10 PASAD models. Adding more than 10 PASAD models decreases the baseline's throughput.

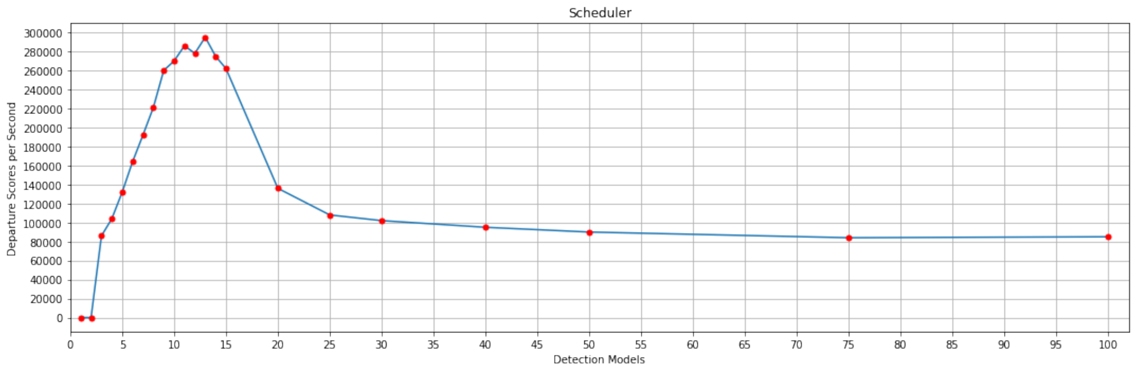


Figure 18: The dynamic user-level scheduler's throughput performance.

Just as in the previous graph, the vertical axis in Figure 18 represents throughput although now for the dynamic user-level scheduler. As can be seen in the graph, the scheduler has its peak throughput between nine and 15 PASAD instance with a peak at 13 PASAD models. Adding more instances causes the scheduler to decline in throughput.

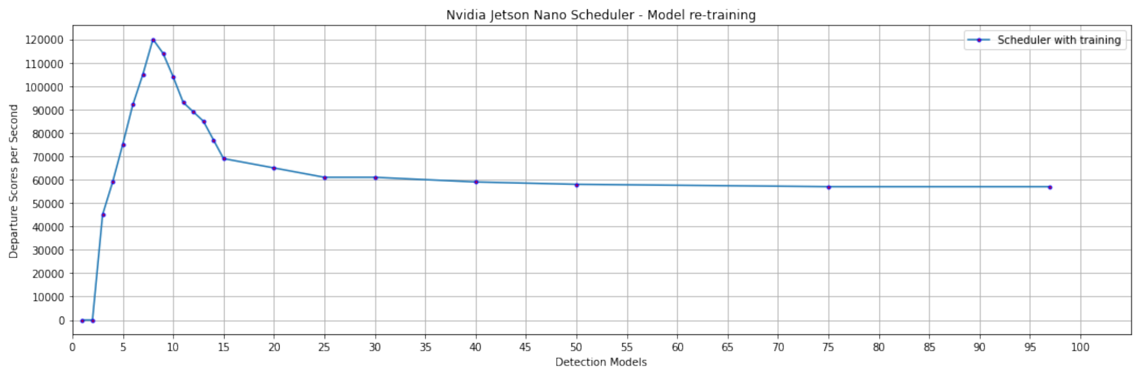


Figure 19: Throughput performance while conducting continuous re-training of models.

Unlike the Raspberry Pi, Nvidia Jetson Nano proved capable of executing model re-training and inference simultaneously if more than two PASAD models were running. Figure 19 shows the system's throughout while continuous retraining is taking place.

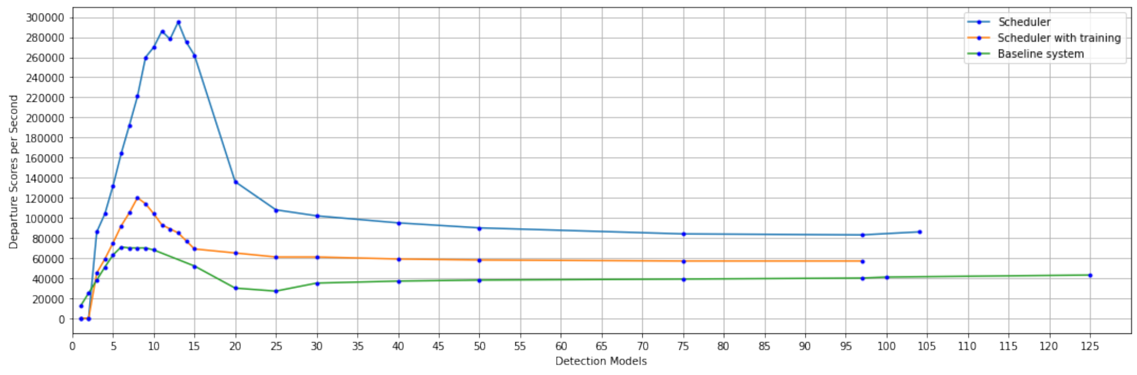


Figure 20: Performance comparison with re-training

Figure 20 depicts the difference in throughput for the baseline and the scheduler with or without model retraining taking place. We can see that the scheduler both with and without model re-training out-performs the baseline at all occurrences except when running only one or two PASAD models. As an added bonus here, we also tested how many PASAD models each system was capable of running before the whole process was killed by the OS, indicated by each line ending at different locations on the vertical axis.

As shown in Figure 20 the Nvidia Jetson Nano was able to run the scheduler with and without model re-training with no more than 97 and 104 Pasad instances respectively, whereas it could run with 125 Pasad models with the baseline. Linux killed the program when we attempted to go beyond these end points. The system monitoring utility ”jtop“ indicated that the RAM was completely full when adding more Pasad instances than the end points.

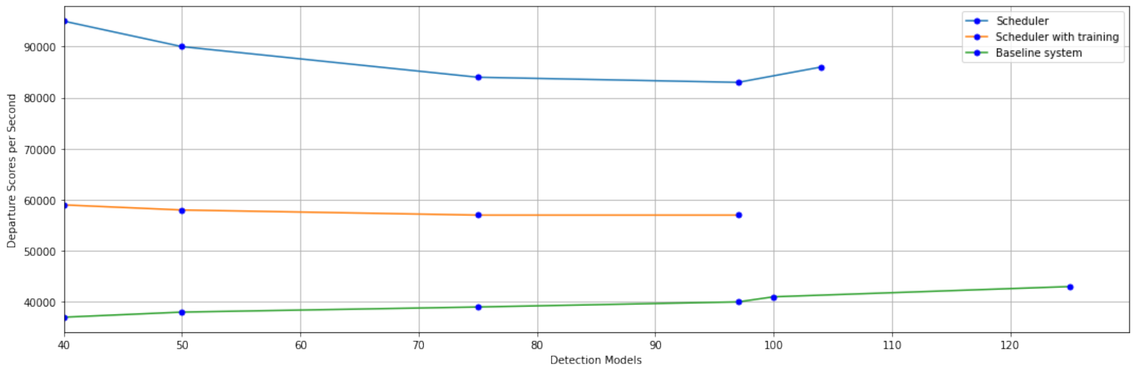


Figure 21: Performance comparison highlighted at 40 or more Pasad models

Figure 21 provides a clearer view of the throughput difference with 40 or more models.

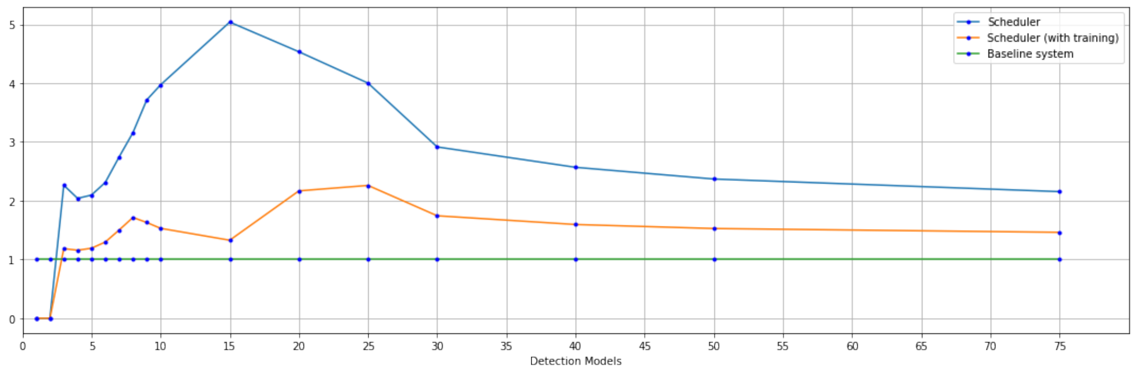


Figure 22: Performance comparison normalized on the baseline system.

Figure 22 displays the same information as Figure 20, however now normalized on the baseline’s throughput to demonstrate how our architectures scale in comparison. We can see that a performance boost is gained with the scheduler, and even with model re-training enabled it is still able to out-perform the baseline’s throughput. Note that the dip at 15 PASAD models when retraining is enabled derives from the fact that in Figure 20 the baseline’s performance does not decrease as fast as the scheduler’s.

6.1.3 Raspberry Pi 4 versus Nvidia Jetson Nano

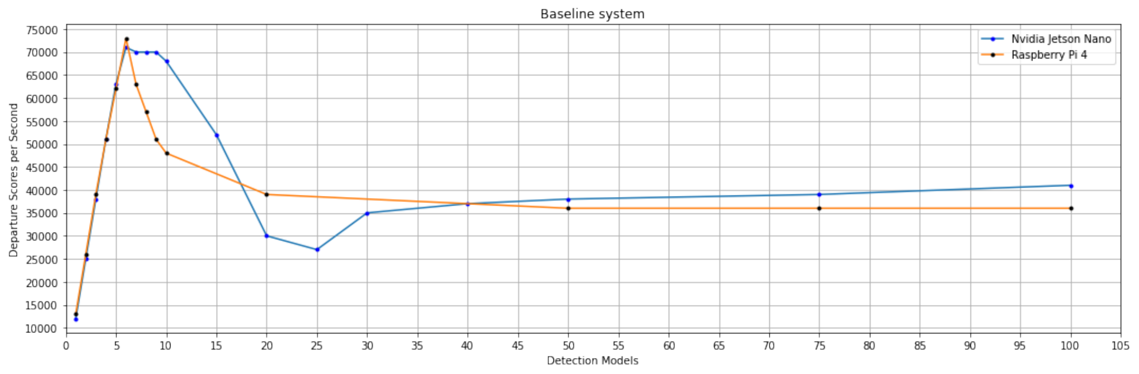


Figure 23: Throughput comparison between the Raspberry Pi 4 and Nvidia Jetson Nano running the baseline system.

Figure 23 indicates a similar performance for both hardware devices up to six PASAD models. The Nvidia Jetson Nano is able to stay at its peak throughput with a few more PASAD models, but just like the Raspberry Pi 4, eventually decreases in performance. Interestingly, while the Raspberry Pi 4 reaches a plateau after 20 Pasad models, the Nvidia Jetson Nano increases in performance again when executing with more than 25 Pasad models.

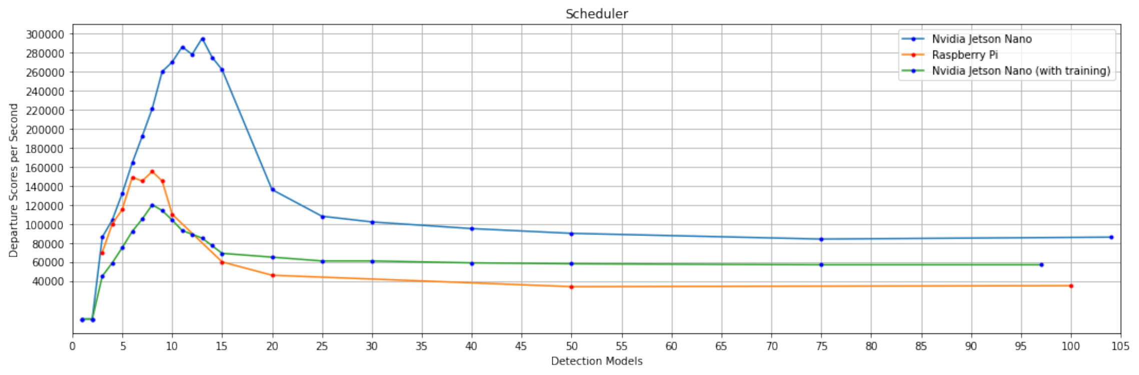


Figure 24: Throughput comparison between the Raspberry Pi 4 and Nvidia Jetson Nano running the scheduler.

Figure 24 demonstrates that the Nvidia Jetson Nano out-performs the Raspberry Pi 4 in the case of the scheduler without model re-training. Interestingly, even though retraining models is computationally intensive and thus decreases performance, the Nvidia Jetson Nano remains superior (green graphs) to the Raspberry Pi 4 (recall that Raspberry Pi 4 could not perform retraining). Recall from Section 6.1.1 that the Raspberry Pi 4 is not at all capable of executing retraining and inference simultaneously.

6.2 Model prioritisation

System type	Time until detecting alarm	Extra time taken
Scheduler	98.167 s	-
Scheduler (no prioritisation)	104.963 s	≈ 6.8 s
Baseline	187.252 s	≈ 89.1 s

Table 6.1: Time measurements averaged over three runs.

In Table 6.1, the time it took for each system version to detect an alarm has been listed. In this experiment, each system was processing data at their maximum capacity for 75 models, which for both versions of the scheduler is 84,000 anomaly scores per second. The baseline system however, can only produce 39,000 scores per second while running this number of models. We can see that the lower throughput has a significant impact on the attack detection speed.

System type	Time until detecting alarm	Extra time taken
Scheduler	180.643 s	-
Scheduler (no prioritisation)	180.742 s	≈ 0.99 s
Baseline	187.252 s	≈ 6.61 s

Table 6.2: Time measurements averaged over three runs.

Table 6.2 displays the results from a similar experiment as for Table 6.2. However this time the throughput for all system versions was adjusted to the baseline's maximum with 75 detection models, which is 39,000 anomaly scores per second. This test was conducted to see how the model prioritization feature would perform when the throughput is equal for all systems. We can see that running the scheduler at a much lower throughput than what it is able to handle causes the benefits of model priotisation to almost be eliminated.

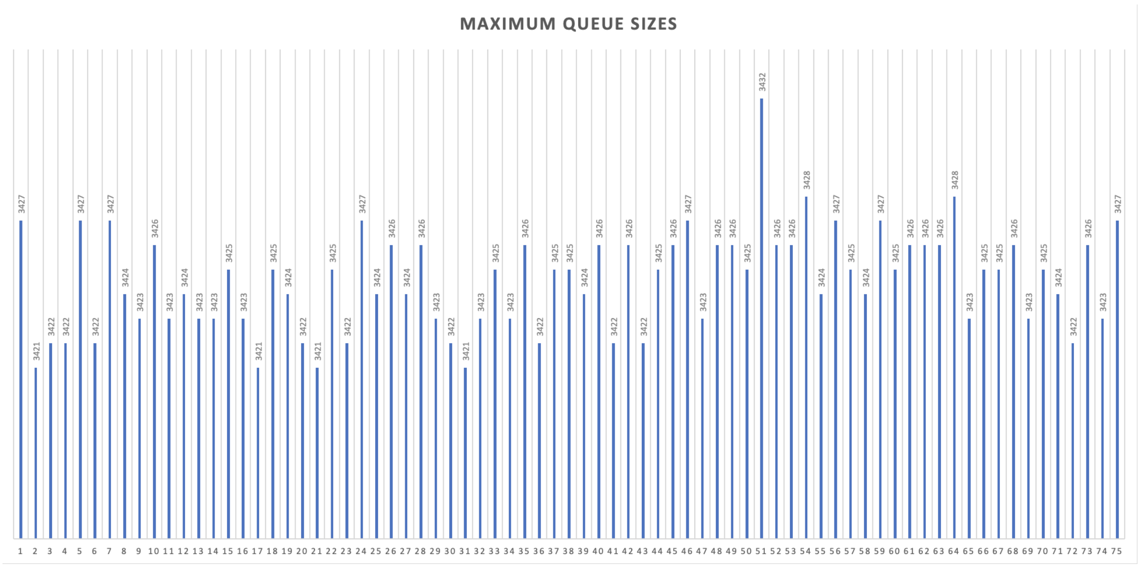


Figure 25: Each column displays the largest buffer size for each detection model throughout the complete runtime.

Figure 25 displays the maximum buffer sizes after running the same experiment as in Table 6.1 on the baseline system.

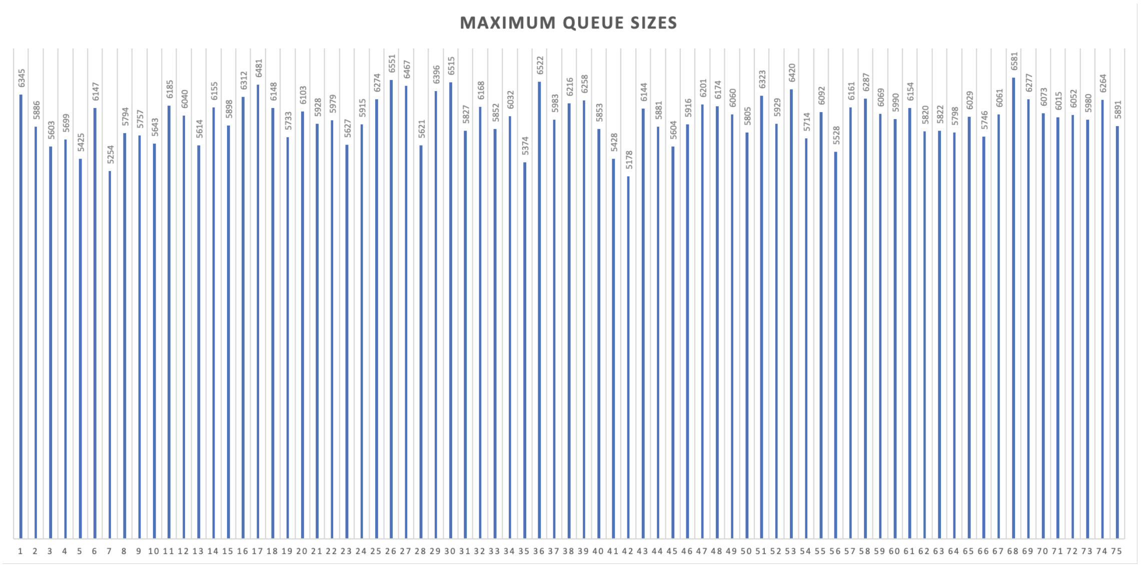


Figure 26: Each column displays the largest buffer size for each detection model throughout the complete runtime.

Figure 26 displays the maximum buffer sizes after running the same experiment as in Table 6.1 on the scheduler without the model prioritization feature activated.

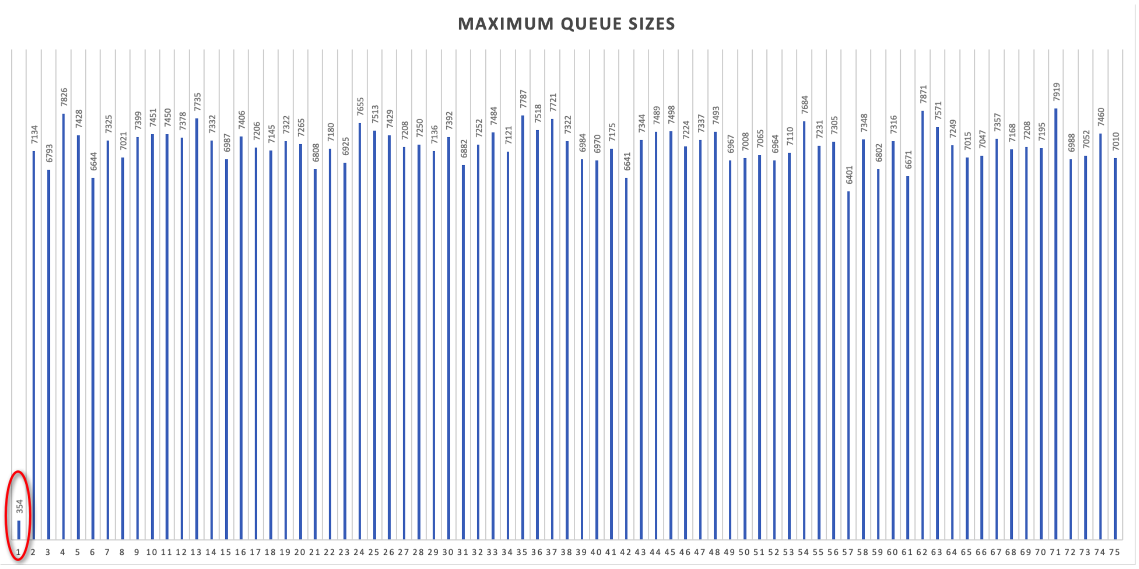


Figure 27: Each column displays the largest buffer size for each detection model throughout the complete runtime.

Figure 27 displays the maximum buffer sizes after running the same experiment as in Table 6.1 on the scheduler with the model prioritization feature activated. The metric for the single model which intentionally breached both the soft and hard thresholds has been highlighted using a red circle. We can see that the higher prioritisation has a significant effect on its buffer size, since consumer threads are ”choosing“ to perform work on this detection model more often.

6.3 The producer-consumer relationship

As mentioned in Section 5.7, we conducted experiments to gain a better understanding of the relationship between producer and consumer threads.

As seen in Figure 28, splitting up the data into various buffers does not affect the speed of the consumer. The consumer's execution time remains roughly constant throughout all the different buffer sizes.

Figure 29 displays performance tests on how accessing mutex locks affects the producer and the consumer. As seen in the graph, there is a significant difference when there only exists one buffer between the producer and the consumers.

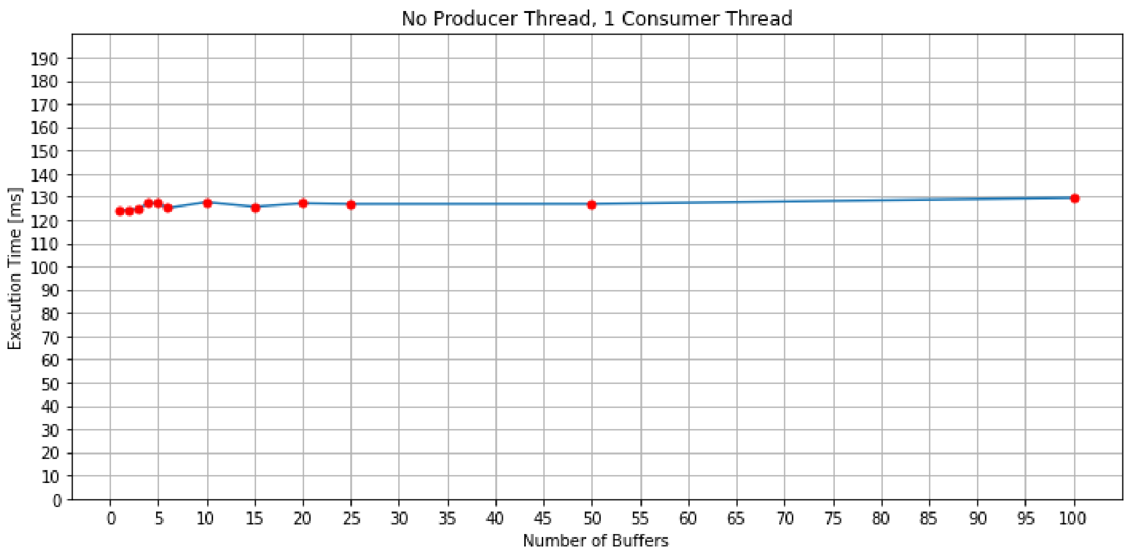


Figure 28: Experiment where the producer thread is eliminated, hence all buffers are pre-loaded during runtime initialization with the same amount of data in total.

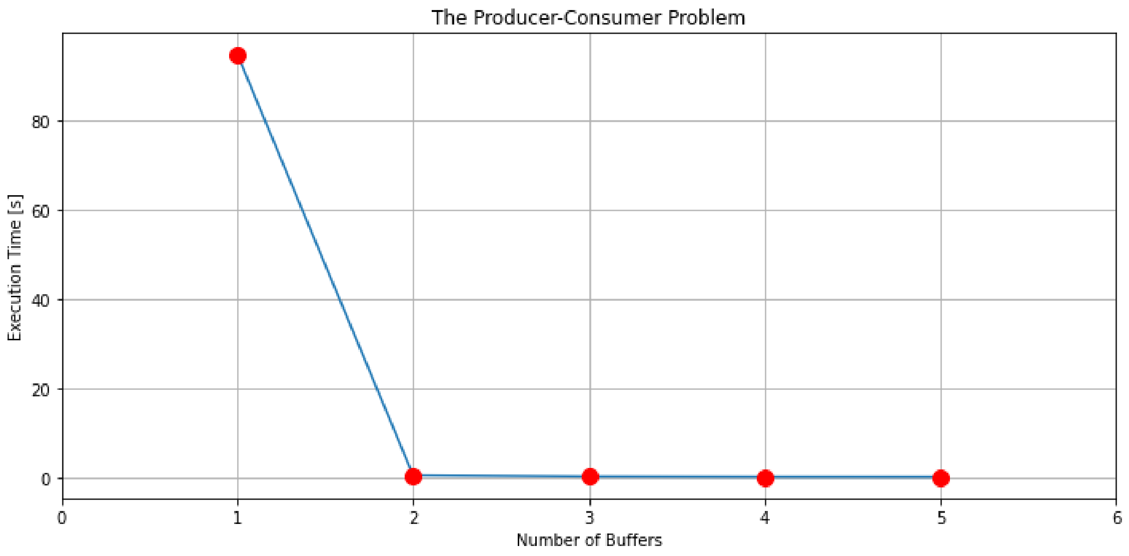


Figure 29: Experiment on a simple producer-consumer test-program.

6.4 PASAD's inference-related functions

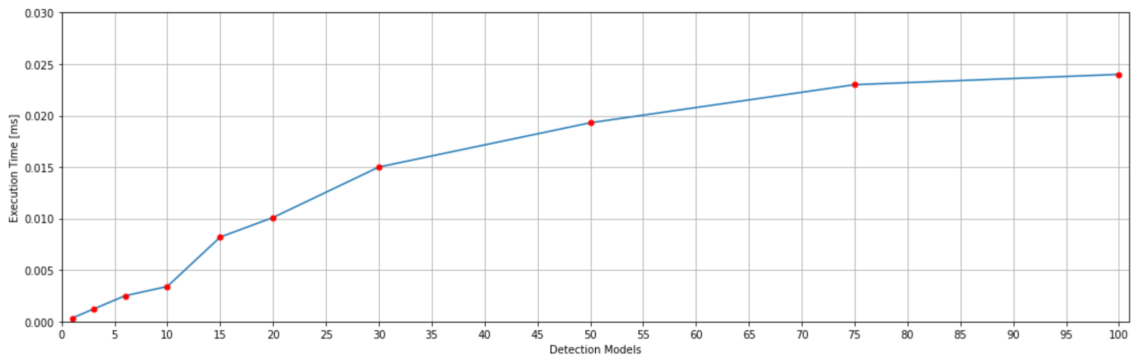


Figure 30: Average execution time of the function adding data to the internal test-set for different numbers of PASAD instances.

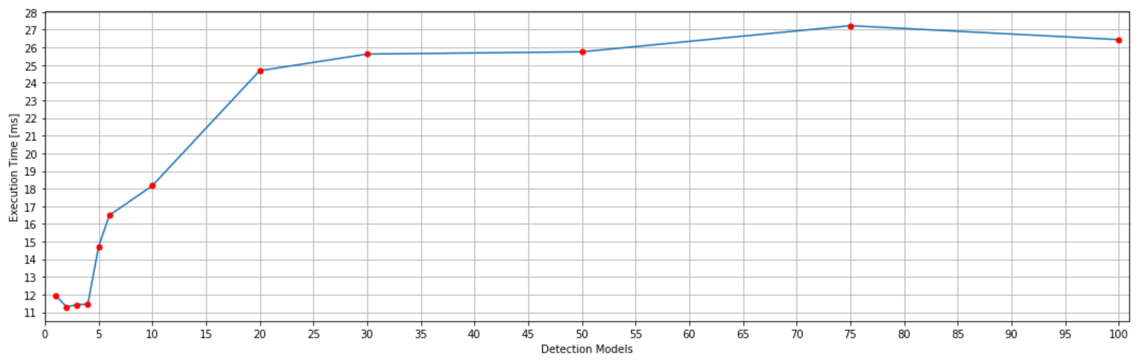


Figure 31: Average execution time of the function producing anomaly scores in relation to the number of PASAD instances.

Figures 30 and 31 shows the cost in terms of execution time for executing PASAD's internal inference-related functions. These are the only two function calls we need to make from PASAD's own header file. Especially interesting is that the graph in Figure 31, which represents the cost for calculating anomaly scores is nearly identical to the inverse of Figure 14 when the throughput decrease takes place.

6.5 Scheduler features

Figures 32 and 33 shows that the added features for dynamic resource prioritization, along with our efforts to minimize overhead and starvation (see Sections 4.2.1 and 4.2.2) are actually leading to an increase in throughput compared to letting multiple consumer threads iterate through their own part of the data-structure that contains all detection models. It can further be seen that the scheduler with all features enabled produces a smoother graph where as the more basic iterative version generates a graph with several declines before reaching its peak. Note that model re-training results in worse performance.

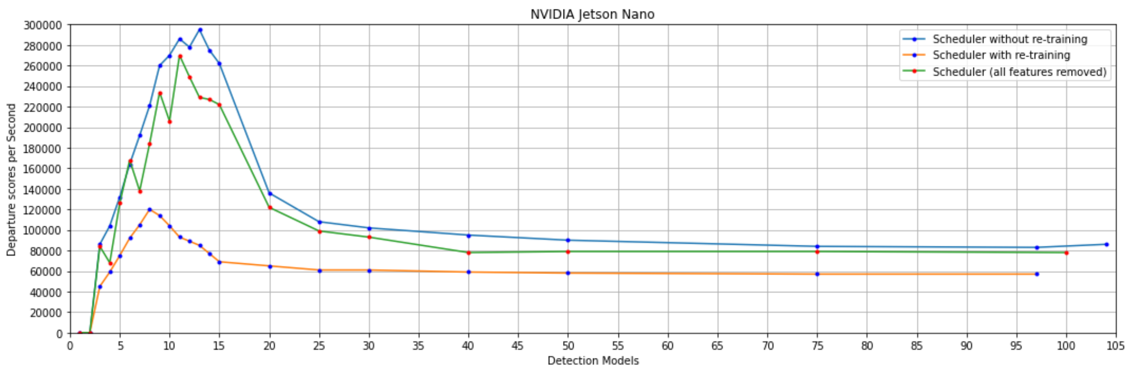


Figure 32: Comparing our multi-threaded scheduler with a more basic iterative version on the Nvidia Jetson Nano.

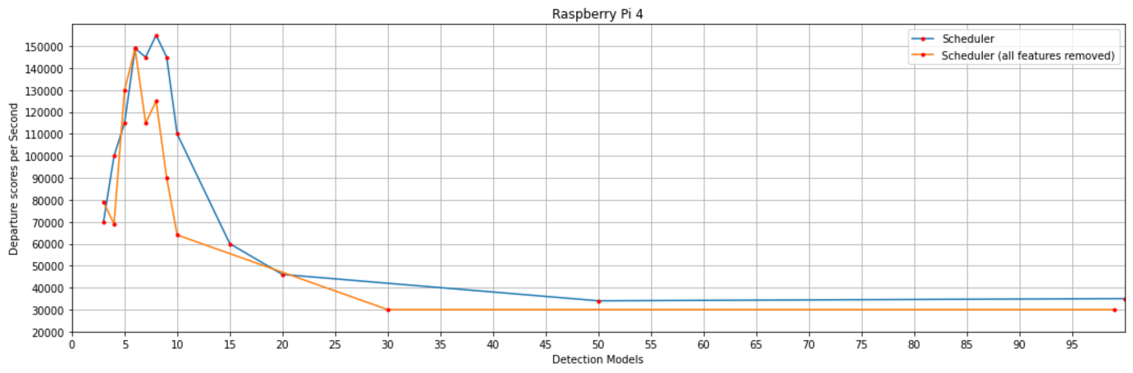


Figure 33: Comparing our multi-threaded scheduler with a more basic iterative version on the Raspberry Pi 4.

6.6 Runtime & Scale-up measurements

Nvidia Jetson Nano

System	Throughput [anomaly scores/s]	#PASAD models
Baseline	37,000	75
Scheduler w/o retraining	80,000	75
Scheduler w/ retraining	53,000	75

Table 6.3: Throughput values chosen from Figure 20.

System	Execution time 50 loops [s]	Improvement faction
Baseline	290	-
Scheduler w/o retraining	163	≈ 1.78
Scheduler w/ retraining	216	≈ 1.34

Table 6.4: Runtime measurements with 75 PASAD models.

The runtime measurement indicates how much faster the scheduler is compared to the baseline. As seen in Table 6.4, with retraining disabled the scheduler is 78% faster and with re-training enabled the scheduler is 34% faster than the baseline.

System	Data points analyzed	Scale-up
Baseline	$\approx$ 10.000.000	-
Scheduler w/o retraining	$\approx$ 16.400.000	1.64
Scheduler w/ retraining	$\approx$ 12.400.000	1.24

Table 6.5: Scale-up with 75 PASAD models given 290 seconds of execution time.

The scale-up factor indicates how many more data-points are analyzed by the scheduler during an equal execution time compared to the baseline. As seen in Table 6.5, with retraining disabled the scheduler is able to perform anomaly detection on 64% more data points while with retraining enabled it can analyze 24% more data points compared to the baseline.

Note that the throughput of each system was slightly lower in this experiment than their stated maximum in Figure 20. This is discussed in Section 7.6.

Raspberry Pi

As seen in Figure 16, the scheduler achieves an identical throughput as the baseline when running 75 PASAD models. Due to this, there are no noteworthy results regarding runtime and scale-up.

7
Discussion

This chapter deals with analyzing and reflecting upon the findings in Chapter 6. We mainly discuss the results and behavior in regards to the baseline and scheduler. In particular, we examine why the Raspberry Pi and Nvidia Jetson Nano produced so drastically different results with the scheduler.

7.1 Linear behavior in the maximum throughput graphs

All graphs related to throughput performance for the baseline system and the scheduler in Chapter 6 show a linear behavior before reaching their peaks. This is due to a common problem for producer-consumer architectures and how the threads inside them access the mutex locks. Recall that in our architectures the producer must continuously capture communication traffic, push data from it into buffers while the consumer threads are continuously producing anomaly scores whenever they find any element stored in those buffers.

When we eliminated the producer however, such as in the simple test seen in Figure 28, the speed of the consumer remains fairly stable even when the data it is accessing gets split among an increasing number of buffers. This points to the fact that execution time on the consumer side does not necessarily need to change drastically just because more buffers are added or removed inside the system.

Furthermore, in Figure 29 we see that execution time is much higher when there is only a single buffer between the producer and consumer. This occurs when either the producer or consumer thread is working so fast that one of them are able to lock a mutex, finish the task at hand and unlock the mutex. However, the thread then immediately locks the mutex again (as its work is encapsulated inside a continuous loop) all while the other thread is still waiting for access to the same mutex lock. Upon closer inspection of this experiment we realized that not only could the producer lock out the consumer, or vice versa, which of the two threads that were locking out the other could also randomly change during runtime. However, when adding more buffers to the system, the threads started to co-operate in ”harmony“ and the overhead caused by accessing mutexes quickly dropped.

Recall that in our architectures the producer thread and the consumer do not operate at the same speed. The consumer is much slower due to having to produce anomaly scores, an event we have very little actual control over as well as insight into, since it was conducted through calls to PASAD's own header file. With all of the observations above in mind, the linear behavior in the beginning of the performance graphs for both hardware devices can likely be explained by the producer and the consumer not having reached an equilibrium yet in terms of their speed in relation to the number of mutex locks present. Note that this point of equilibrium is unique for both the baseline and the scheduler as it appears when different amounts of PASAD models are running.

7.2 Raspberry Pi's performance issues

Figure 16 is a particularly important figure to analyse, due to the peculiar performance behaviour of both systems there. Especially important to address is the fact that beyond a certain amount of detection models, our fully fledged scheduler shows zero performance improvement compared to the baseline. This is a direct threat to the purpose of our thesis, which to a large part intends to improve performance through a user-level scheduler that utilizes multi-threading. Most importantly, the graphs states that executing a system with triple the threads (on the consumer side) results in no performance gain when scaling the system which does not fit within reason.

After an extensive period of analyzing and pondering however, we realized that processing power was actually not the bottleneck causing this, as previously assumed. The reason why both systems appear to hit some sort of "performance wall", and then remain at the same throughput, is actually most likely due to them experiencing a variant of being memory bound, happening between the CPU and Random Access Memory (RAM). This occurs when an application is dependent on data in order to perform its computations, but has already reached the limit of how fast that data can be retrieved from the RAM. The graph in Figure 16 shows a drastic decrease already before ten PASAD models are running. We do not have enough information regarding PASAD to calculate how much memory each model occupies (running "sizeof()" is not sufficient), but Figure 16 suggests that from ten models and beyond we are exceeding what fits inside the L2-cache and begin accessing the RAM. Add to this the fact the hardware device itself only has a single memory channel between the L2 cache and RAM and the plateau becomes much more logical.

The downwards slope can then be described by the consumer thread(s) starting to access the RAM more frequently as more and more models are added, and when we are going beyond roughly 20 modes they are accessing the RAM constantly. At this point, the memory bandwidth, meaning how fast data can be retrieved from RAM becomes a limiting factor. Additionally, when having multiple consumer threads as with the scheduler, the single memory channel causes significant idleness. That is because only one consumer thread is able to go through the channel at the time, while the others can not do anything but wait.

At that point, throwing additional processing power at the problem will not be of any aid, which is precisely why the scheduler's throughput results get bottlenecked at the exact same level as the baseline system. More powerful cores would only aid in making the tasks of the consumer threads finish faster once they actually have the data they need in possession.

This performance issue is not caused by any functionality of the scheduler itself however, but rather that we have employed PASAD as our detection algorithm. As can be seen in Figure 31, the cost of producing anomaly scores increases drastically as soon as we deploy more than approximately four models. The other function that has been supplied to us for adding data-points to each PASAD object's internal buffer, seen in Figure 30 also increases in cost, however it always remains at a significantly lower level comparatively. As was stated back in Section 1.6, we treat the detection algorithm itself as a black box and hence do not possess full knowledge of what exactly is occurring internally inside PASAD whenever these two functions are called. Important to point out is that Figure 31 is suspiciously similar to the inverse of the baseline's throughput graph in Figure 14 right after the downward slope has begun. As we increase the number of models, the cost for calculating anomaly scores increase as well (most plausibly due to accessing RAM more often), significantly hurting overall performance.

We believe that the single memory channel is also why the Raspberry Pi was completely unable to perform model retraining and inference simultaneously. Retraining is not only a computationally heavy task, it also requires a lot of data (30,000 data-points in our case as stated in Section 5.1). What is occurring then is that as soon as the training phase begins, the memory channel gets fully occupied by the training thread while all consumer threads are waiting idly, putting a halt to all their tasks.

7.3 The dynamic user-level scheduler

In the case of running the dynamic user-level scheduler, the Jetson Nano proved to be vastly superior than the Raspberry Pi despite having less capable CPU cores. As seen in Figure 24, the scheduler running on the Raspberry Pi has the worst scaling performance. We explained in the previous section that the reason why the Raspberry Pi was performing poorly was not due to computational power, but rather the structure of its memory hierarchy.

Firstly, since the Jetson Nano has an L2-cache size of 2MB (double the Raspberry Pi's capacity) it is able to fit a higher number of PASAD instance inside its cache memory. This is the reason why its throughput decline occurs later, and the higher memory bandwidth aids performance as well since the constantly required data can be accessed faster. But even more significant for this thesis, is that the Jetson Nano is equipped with dual channel memory. This significantly reduces idleness as there are now two ways instead of just one that the threads can take to access the RAM.

As can be seen in Figure 16 and 20, this leads to the Jetson Nano's performance never dropping to the exact same throughput as the baseline system, contrary to the Raspberry Pi. It also enables for simultaneous training and inference to take place, as one memory channel may be occupied for each of these data-heavy tasks.

In Sections 4.2.1 and 4.2.2, we described how the scheduler dynamically prioritizes its resources as well as minimizes starvation and overhead. Figure 32 and 33 show that these features enables the scheduler to reach slightly greater throughput levels (when not retraining models), compared to simply letting each consumer thread iterate through their own part of the data-structure that stores all detection models. Especially noteworthy is the jagged behaviour seen in the graph for the iterative version. This takes place when the number of PASAD models in the system cannot be evenly split between the consumer threads for optimal load balance.

Moreover, with our added features, the scheduler seems to achieve better load balance across the consumer threads since we are not experiencing the same performance drops as the iterative version whenever the number of PASAD models cannot be evenly divided among them. Our explanation for this is that allowing models to be prioritized results in models already brought into cache to occasionally be re-selected, while each consumer thread in the iterative version has to go through all its models before performing work on the same one again, forcing it to constantly access the RAM.

It is important to add however that these features do still add some overhead, especially when a large number of models need to be checked before a decision regarding which one to execute is taken. This can be seen in the runtime measurements from Section 6.6 that are not scaling linearly between the baseline system and the scheduler (although several additional factors are likely also contributing to this which will be discussed).

We argued in section 5.10 that a perfect runtime improvement would be of the value ”3“, indicating a performance tripling. But our results are showing an improvement of 1.78 for the scheduler system compared to the baseline, and drops even further to 1.34 when retraining is enabled. Both of these scheduler versions could also perform 64% and 24% more anomaly detection respectively during the same execution time compared to the baseline, judging from our scale-up results.

Figure 22 is also showing some interesting variances where the throughput performance has been normalized on the baseline system. Without retraining, the scheduler even reaches upwards of five times higher throughput when running a low number of models, then two times the performance once the slope evens out.

Remember though that one of the most important contributing factors for us not reaching closer to a perfect speed-up result is the chosen detection algorithm PASAD. Improvements to its implementation would reduce the overhead when calling on its functions.

Another factor can be that the mutex locks for guarding buffer accesses are having an effect even when running a large amount of detection models. In Figure 26 we can witness this through the fact that there are large differences in terms of how sizeable each model's buffer got, even though the consumer is instructed to pop elements from them one by one. This is most likely due to the consumer thread occasionally being denied the mutex and therefore carries onto the next buffer. What is even more interesting is that the differences seen in this diagram appear to be significantly greater than in Figure 27, where our model prioritisation feature handles the choice of which buffer to remove an element from.

Lastly, we need to mention that our baseline's architecture (and to some extend the scheduler built upon it) is likely far from the perfect implementation of a two-threaded IDS. Our approach when developing it was to satisfy the requirements in Table 4.1, which mainly entailed the creation of a simple producer-consumer architecture with one producer and consumer thread. Because of this, there still exists room for optimization work to be made and the results from Figure 22 would differ if normalized on a "perfect“ two-threaded IDS.

We are without a doubt sure that there are several additional factors contributing to our performance results, which we were unable to fully uncover within the time-frame of this thesis. Embedded systems like the IoT-devices we have utilized are complex and they differ from usual computer systems as there is a wide range of components fitted in to interact together on a single small chip.

7.4 Attack detection

The reason as to why the system throughput and our model prioritization feature are important for attack detection as well as from a security perspective in general has been thoroughly explained in Section 5.5 and 5.6 respectively. Therefore, in this section we will focus on the results obtained concerning the latter in Section 6.2.

The most important takeaway from Table 6.1 is that model prioritization allowed the scheduler to detect an alarm almost seven seconds faster, even though all other parameters of the experiments were the same. A perhaps more appealing way to visualise the reason for this would be to compare Figures 26 and 27. Here, we clearly see that the consumer threads have performed much more work on the single model that was set to breach our thresholds judging from how much smaller its queue size got. We do not believe however that the much longer time it took for the baseline system to detect an alarm in Table 6.1 is particularly interesting, since that is certainly mainly due to it being unable to run at an equal data input rate.

Therefore, in order to obtain some sort of comparison with the baseline we ran the tests again, however this time while all of them were getting the same input rate as what the baseline can maximally handle. We can from the results of this experiment shown in Table 6.2 see that the scheduler is able to detect the alarm over six seconds faster. But what is also interesting about these results is that the benefits of activating model prioritization on the scheduler almost completely disappeared. That is because in order to witness an effect of model prioritization, there must actually exist some margin of data within the buffers for the consumer threads to work with. But since the scheduler is running here on a much lower rate of input than what it is able to handle, the data stored inside each buffer gets processed almost immediately anyway, hence the benefit of prioritizing vanishes.

Lastly, we wish to briefly mention our thoughts on that it might be possible for an adversary to take advantage of our scheduler's model prioritization. This entails however that the adversary is informed about the existence of these logistics inside the IDS guarding the targeted node. The attacker could for example intentionally cause alarms to happen on insignificant nodes, in order to make the scheduler allocate its resources there, hence distracting it from the real target. Anyone intending to take inspiration from this thesis work for their own implementation should take this possibility into account.

7.5 Model retraining

Model retraining has been one of the most interesting parts to have been investigated in this thesis, and as shown in Chapter 6, it was only possible to perform on the Jetson Nano. As was gone over thoroughly in Section 7.2 and 7.3, we strongly believe that the reason for this is that retraining fully occupies one of the memory channels between the CPU and RAM while it is happening.

On the Raspberry Pi then which is only equipped with one such channel, the idleness caused by consumer threads waiting to use it freezes all inference-related tasks until retraining is concluded. This leads to packets being missed and therefore failing our project goal of achieving simultaneous inference and training.

We should mention that although the Jetson Nano succeeded very well in this regard, retraining of machine learning models still remains the most heavy task our scheduler has to perform and leads to a noticeable performance drop as can be seen in graphs such as Figure 22.

It is our belief that simultaneous execution of inference and training of machine learning models on an IoT device is the true highlight of this thesis. It is the aspect that delivers the most research value in terms of future work for all communities that are passionate about performance as well as security. Our thesis has demonstrated that by using our scheduler, it is possible to retrain one detection model in parallel with 96 other active detection models producing 57,000 departure scores per second.

Both the level of anomaly score throughput as well as performing continuous training is an overkill of what would actually be needed in a realistic scenario. This is because there is a low likelihood of needing to continuously retrain detection models one after the other. In a realistic scenario, the detection models would also not receive such large quantities of data. However for this thesis, we always aimed to stress the hardware to its limit during performance tests, meaning that there is still a good amount of headroom left in case one would like to allocate resources for something else.

This ensures that there is not a necessity for anomaly-based IDS, based on lightweight and data-driven ML-algorithms running on IoT devices to transfer sensitive data over networks in order to offload heavier tasks. Everything remains at the edge layer close to the monitored nodes.

Simultaneous inference and training still remains a significantly unexplored research area however, and we believe to have only scratched its surface. We therefore hope that this thesis will act as an inspiration for further research, as there is still a lot of potential to be gained within the domain of security and elsewhere.

7.6 Limitations in methodology

As in most research, certain limitations that influence the quality of the results are present in this work as well. One of these limitations is that we measure a system's throughput while dispatching data to it in bursts from a single source. The realistic scenario that a system like what we have developed would encounter is a lot more complex, and difficult to recreate.

As already explained in Section 1.4, each node that the IDS is monitoring is represented by a unique instance of the detection model. In reality, each of these instances would have a separate stream of incoming data, meaning not from as single producer as in our system although we have tried to simulate this behavior by allowing the user to alter the distribution pattern. Each node would also often produce different event streams from one another, which in addition could be highly irregular, meaning that the rate of incoming data varies with time. Some nodes might only produce bursts of event streams for a short period of time, while others might produce them with no apparent pattern and be nondeterministic.

Important to discuss next are the runtime and scale-up measurements from Section 6.6. To perform these tests, the throughput in all three cases had to be lowered a bit from what had been stated as their maximum values earlier in Section 6.1, as can be seen in Table 6.3. The reason for this is because the execution time was now being monitored using the ”Chrono“ library included in C++ as well as the ”time“ command in Linux. Profiling tools such as these come with their own contributions of overhead. This caused our systems to miss packets if they were trying to achieve their maximum throughput levels.

Perhaps the largest setback and cause for confusion in this thesis was that the hardware devices utilized in our experiments prevented us to from seeing important metrics needed to further validate our theories. This is mainly due to the manufacturers of their ARM-based chipsets having chosen to not provide many hardware supported events for profiling tools. Being able to view statistics such as ”Last Level Cache Misses“ or ”Stalled Cycles“ would have been a valuable asset when explaining the consumer thread idleness in Section 7.3 and the Raspberry Pi's memory boundedness in Section 7.2.

Lastly, we do not have control over how the Linux kernel chooses to prioritise processes and what it is working on in the background. The background processes can impact the memory and computational resources available to us, and therefore influence our results when conducting experiments. Additionally, the kernel and the activities it performs can change over time.

7.7 Hardware requirements for the scheduler

In order to be able to run our scheduler on an IoT device one must choose a device capable of satisfying certain requirements, in order to not run into the same issues as the Raspberry Pi did in our experiments.

Deploying the scheduler without retraining models requires a CPU containing at least two cores, so that the producer and consumer threads can work in parallel. Enabling simultaneous retraining will require an additional core, as this is executed on a separate thread. Retraining will also require at least two memory channels between the RAM and CPU to allow for the needed data transfers for all the tasks executed by the scheduler.

Beyond what is stated above, the scheduler is relatively hardware agnostic towards the number of cores, and users are free to make engage additional ones to enable more consumer threads. The current implementation does not support GPU utilization however.

One should also remember that the number of detection models that can be run is highly dependent on how much memory each of them require and how much the IoT device grants. In our case, 4GB of RAM was enough to run roughly 97 PASAD models while retraining was taking place.

7.8 Ethics & Sustainability

Back in Section 1.2, we outlined some of the existing concerns regarding IoT devices and their vulnerabilities towards cyber attacks, as well as what damaging consequences such events can have. The work of this thesis may lead to a reduction of those consequences, by allowing attacks to be detected earlier. Faster detection paves the way for a faster response, preventing the devices from being misused and thereby making this thesis important for sustainability.

We deliver on this point by having created a scheduler which can take advantage of the processing cores available on the device it is being deployed on to increase the system's throughput, as well as with its capability to allocate resources towards nodes needing it.

Another contributing element for preventing the misuse of IoT devices is the work made to keep the execution of model retraining local and without interrupting any other important tasks of the IDS. As described in Section 1.4, this is an important outcome as the risk for blind spots where an attacker may operate freely is eliminated.

There is also an ethical component for not needing to offload the training-related tasks. Information read by an IDS is often sensitive, and transferring that data over networks could lead to it being leaked in case an attack should occur. In order to also ensure that we ourselves do not expose any personal information during this project, our partner company Clavister provided us with traffic data that had already been anonymized and pre-processed in a way that makes it suitable for creating AI/machine learning models intended for outlier/attack detection.

7.9 Goals reached

In Section 1.5 we listed our goals for the thesis. They consisted of developing a simple anomaly-based IDS capable of capturing communication traffic and feed data from it to a machine learning model for anomaly detection. We would thereafter incorporate further enhancements to this IDS using parallelism techniques from Section 3.1, such as mainly scheduling to increase the system's throughput and performance from a security perspective.

An analysis using primarily the metrics described in Section 3.3 needed to be done afterwards for drawing plausible conclusions regarding the project's level of success. Lastly, we stated that we would investigate whether it is possible to retrain a machine learning model locally on the IoT device in parallel with ongoing intrusion detection.

Following the self-evaluation of our thesis in this chapter's previous sections, we feel confident with stating that we have successfully accomplished all of thesis goals from Section 1.5.

8 Conclusion

8.1 Future work

In order to fully optimize the scheduler, one would need to evaluate it on an IoT device that supports cache and CPU metrics. Since the available hardware in this thesis did not provide support for those metrics, we were unable to fully examine how memory efficient the scheduler actually is. For example, analyzing the scheduler through metrics such as ”last level cache references and misses“ would clearly indicate the cache hit rate for the scheduler. A high hit rate on the last level cache means that the scheduler is less often in the need of accessing data from the RAM. This would improve the performance of the scheduler since accessing the RAM is much slower than the cache. There are of course more metrics that would be beneficial to analyze in order to make the scheduler as efficient as possible.

Additionally, to make the scheduler memory efficient one should investigate how the buffer system between the producer and consumer(s) can be improved. In the current implementation, many elements are constantly being pushed and removed from multiple buffers simultaneously, resulting in extensive memory copying. A reduction of memory requirement for the buffers could also enable the scheduler to run more detection models without reaching the RAM's maximum capacity.

Currently, the scheduler is coded as a single large program, however we realized that splitting it up into separate processes would provide us with the benefit of easily being able to instruct the Linux kernel to assign a different priority level for each one of them. This would eliminate our fear expressed in Section 5.3 regarding the producer thread being time-sliced, as we could assign it the maximum priority hence allowing its execution to never be compromised by other processes. This would in turn lead to us being able to declare more consumer threads, making the total amount of threads exceed the number of available cores, in order to squeeze out the CPU:s last remaining computing resources.

While the scheduler is capable of re-training models and run inference in parallel, it would be intriguing to explore the performance gains from a GPU. GPU:s are better designed than CPU:s for computationally intensive operations since the purpose of a GPU is precisely to maximize throughput. As we move forward in technology, better GPU:s are incorporated within IoT devices and in the case of the NVIDIA Jetson Nano, there exists a GPU with 128 CUDA cores that we have not even touched yet.

Offloading the retraining of models to the GPU could make that task execute faster, as well as freeing up those resources on the CPU for additional inference monitoring.

Another aspect to investigate for future work is to analyze PASAD's internal functions and how PASAD works. We chose to view the chosen detection model algorithm as a black box but one would be naive to assume that its implementation is perfect. It is our function calls to PASAD that costs us the most in terms of memory and computational resources, therefore improving it internally would benefit the whole system. Lastly, it would be interesting to observe the scheduler's performance when using other lightweight and data-driven machine learning models. These assessments would aid the scheduler to be even more agnostic to the chosen detection model.

8.2 Conclusion

We have constructed an anomaly-based IDS for IoT devices, which utilizes PASAD as its detection algorithm by developing our own dynamic user-level scheduler. The results show great promise as our solution is able to calculate 57,000 departure scores per second for roughly 97 detection instances while simultaneously training models using the NVIDIA Jetson Nano. Additionally, our scheduler performs ≈1.46 times better than the baseline even with retraining enabled.

In order to run the scheduler with all of its features enabled the IoT device needs to be equipped with a multi-core processor (containing at minimum three cores), as well as having at least two memory channels for RAM access.

Bibliography

[1] N. Chaabouni, M. Mosbah, A. Zemmari, C. Sauvignac, and P. Faruki, "Network intrusion detection for IoT security based on learning techniques," *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2671–2701, 2019.

[2] F. Hosseinpour, P. Vahdani Amoli, J. Plosila, T. Hämäläinen, and H. Tenhunen, "An intrusion detection system for fog computing and IoT based logistic systems using a smart data approach," *International Journal of Digital Content Technology and its Applications*, vol. 10, no. 5, 2016.

[3] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, pp. 2435–2463, 1999.

[4] McAfee. Trending: IoT malware attacks of 2018 [Online]. Available:. https://www.mcafee.com/blogs/mobile-security/top-trending-iot-malware-attacks-of-2018/.

[5] Nokia's threat intelligence report 2019 warns on the fast-growing and evolving threat of malicious software targeting internet of things (IoT) devices [Online]. Available:. https://www.nokia.com/about-us/news/releases/2018/12/04/nokias-threat-intelligence-report-2019-warns-on-the-fast-growing-and-evolving-threat-of-malicious-software-targeting-internet-of-things-iot-devices/.

[6] J. S. Perry. (2017) Anatomy of an IoT malware attack [Online]. Available:. https://developer.ibm.com/articles/iot-anatomy-iot-malware-attack/.

[7] The OWASP IoT Security Team. (2018) Owasp internet of things (IoT) project [Online]. Available:. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project.

[8] Microsoft Threat Intelligence Center (MSTIC). (2019) Corporate IoT – a path to intrusion [Online]. Available:. https://msrc-blog.microsoft.com/2019/08/05/corporate-iot-a-path-to-intrusion/.

[9] Cloudflare. What is the Mirai botnet? [Online]. Available:. https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/.

[10] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai botnet," in *26th USENIX Security*

Symposium (USENIX Security 17). Vancouver, BC: USENIX Association, Aug. 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[11] C. Benzaïd and T. Taleb, "AI for beyond 5g networks: A cyber-security defense or offense enabler?" *IEEE Network*, vol. 34, no. 6, pp. 140–147, 2020.

[12] P. Wheeler and E. Fulp, "A taxonomy of parallel techniques for intrusion detection," in *Proceedings of the 45th annual southeast regional conference*, 2007, pp. 278–282.

[13] Snort. Available:. https://www.snort.org/.

[14] P. S. Wheeler, "Techniques for improving the performance of signature-based network intrusion detection systems," Ph.D. dissertation, University of California Davis, 2006.

[15] U. A. Sandhu, S. Haider, S. Naseer, and O. U. Ateeb, "A survey of intrusion detection & prevention techniques," in *2011 International Conference on Information Communication and Management, IPCSIT*, vol. 16, 2011, pp. 66–71.

[16] A. Khraisat and A. Alazab, "A critical review of intrusion detection systems in the internet of things: techniques, deployment strategy, validation strategy, attacks, public datasets and challenges," *Cybersecurity*, vol. 4, no. 1, pp. 1–27, 2021.

[17] H. Debar, M. Dacier, and A. Wespi, "A revised taxonomy for intrusion-detection systems," in *Annales des télécommunications*, vol. 55, no. 7. Springer, 2000, pp. 361–378.

[18] H. Muccini and K. Vaidhyanathan, "Software architecture for ML-based systems: what exists and what lies ahead," in *2021 IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI (WAIN)*. IEEE, 2021, pp. 121–128.

[19] L. de Souza Cimino, J. E. E. d. Resende, L. H. Moreira Silva, S. Queiroz Souza Rocha, M. de Oliveira Correia, G. S. Monteiro, G. N. de Souza Fernandes, S. G. Moreira Almeida, A. L. Barroso Almeida, A. L. Lins de Aquino, and J. de Castro Lima, "IoT and HPC integration: Revision and perspectives," in *2017 VII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2017, pp. 132–139.

[20] W. Aoudi, M. Iturbe, and M. Almgren, "Truth will out: Departure-based process-level detection of stealthy attacks on control systems," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 817–831.

[21] M. Almgren, W. Aoudi, R. Gustafsson, R. Krahl, and A. Lindhé, "The nuts and bolts of deploying process-level IDS in industrial control systems," in *Proceedings of the 4th Annual Industrial Control System Security Workshop*, 2018, pp. 17–24.

[22] M. Karsten and S. Barghi, “User-level threading: Have your cake and eat it too,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 1, pp. 1–30, 2020.

[23] T. Rauber and G. Rünger, *Parallel Programming for Multicore and Cluster Systems.* Springer, 2013, vol. 2.

[24] N. Chaabouni, M. Mosbah, A. Zemmari, C. Sauvignac, and P. Faruki, “Network intrusion detection for IoT security based on learning techniques,” *IEEE Communications Surveys Tutorials*, vol. 21, no. 3, pp. 2671–2701, 2019.

[25] J. Delua. (2021) Supervised vs. unsupervised learning: What’s the difference? [Online]. Available:. https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning.

[26] R. Samrin and D. Vasumathi, “Review on anomaly based network intrusion detection system,” in *2017 International Conference on Electrical, Electronics, Communication, Computer, and Optimization Techniques (ICEECCOT)*, 2017, pp. 141–147.

[27] V. Jyothsna, R. Prasad, and K. M. Prasad, “A review of anomaly based intrusion detection systems,” *International Journal of Computer Applications*, vol. 28, no. 7, pp. 26–35, 2011.

[28] GadgetVersus.com. Available:. https://gadgetversus.com/processor/broadcom-bcm2711-specs/.

[29] Raspberry Pi. (2019) DATASHEET - Raspberry Pi 4 Model B [Online]. Available:. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf.

[30] NVIDIA. (2014) DATASHEET - NVIDIA Jetson Nano [Online]. Available:. https://developer.nvidia.com/embedded/jetson-nano.