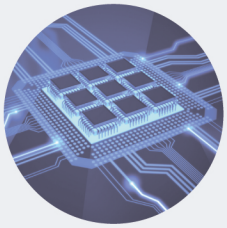
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Deployment of ML Mechanisms for Cybersecurity in Resource-Constrained Embedded Systems

PEDRO MIGUEL CASAL VICENTE
outubro de 2023

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Deployment of ML mechanisms for Cybersecurity in Resource-Constrained Embedded Systems

Pedro Miguel Casal Vicente

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: Pedro Miguel Salgueiro Santos

Evaluation Committee:
President:
Luís Miguel Pinho, ISEP

Members:
Sérgio Crisóstomo, FCUP
Pedro Miguel Salgueiro Santos, ISEP

Porto, October 4, 2023

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

The work presented in this document is original and authored by me, and performed in the scope of the Master's degree in Critical Computing Systems Engineering.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration, all references have been acknowledged and fully cited, and all text was originally produced by me (except when duly noted).

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

Porto, Porto, October 4, 2023

(Signature as in the official identification document, or, preferably, digital signature)

Abstract

The increase of low security devices in the Internet is being exploited by hackers to compromise data or use to use them as external agents to perform further attacks. As so, it is of crucial importance that networks posses a system that correctly assess the nature of incoming and outgoing packets to protect the local network and the overall Internet connected systems. To achieve this, Machine Learning is being broadly used due to his early success. Nevertheless, these mechanisms are better inserted at the entry point of local networks, an embedded system which has limited resources to train machine learning models and/or to perform inference tasks. Since Cybersecurity is a real-time problem, the embedded systems should perform these activities in a very restricted time interval. The time required to classify the packets depends on the overall system load, machine learning models complexity and desired accuracy. This thesis aims to assess the current support for ML in embedded systems, either through the interoperability of models or through their development in low level languages, and the relationship between the time required by different embedded systems, the different tools and models. This thesis explored one transpilation tool, *m2cgen*, two interoperability formats, *PMML* and Open Neural Network Exchange (ONNX) and one real time environment, *ONNXRuntime*, to deploy an already trained model in a device with limited resources. Results demonstrate that *ONNXRuntime* was the only machine learning tool with a perfect match regarding samples prediction's classification from the original models. An analysis on the time required to execute this task revealed that *ONNXRuntime* is faster than *Scikit-Learn* with the Isolation Forest (ISO), One Class Support Vector Machine (OCSVM) and Stochastic Gradient Descent One Class Support Vector Machine (SGDOCSVM) models and slower with the Local Outlier Factor (LOF) model.

Keywords: Embedded, Cybersecurity, Performance, Machine Learning, Accuracy, IoT, Time

Resumo

O aumento do número de dispositivos ligados à Internet com fracos níveis de segurança está a ser explorado por piratas informáticos para comprometer dados ou para os utilizar como agentes externos para realizar novos ataques. Como tal, é de importância crucial que as redes possuam um sistema que avalie corretamente a natureza dos pacotes que chegam e que saem para proteger a rede local e os sistemas conectados à Internet em geral. Para conseguir isso, *Machine Learning* é uma tecnologia que está a ser amplamente usada devido ao seu sucesso inicial. No entanto, estes mecanismos conseguem proteger melhor a rede local se forem inseridos no seu ponto de entrada, um dispositivo embebido que possui recursos limitados para treinar modelos *Machine Learning* e/ou executar tarefas de inferência. Como a Cibersegurança é um problema em tempo real, os sistemas embebidos teem realizar essas atividades num intervalo de tempo muito restrito. O tempo necessário para classificar os pacotes depende da carga do sistema, da complexidade dos modelos e da precisão desejada. Esta tese tem como objetivo avaliar o suporte atual desta tecnologia em sistemas embebidos, seja através da interoperabilidade de modelos ou através do seu desenvolvimento em linguagens de baixo nível, e a relação entre o tempo exigido por diferentes sistemas embebidos, diferentes ferramentas e modelos. Esta tese explorou uma ferramenta de transpilação, *m2cgen*, dois formatos de interoperabilidade, *PMML* e ONNX e um ambiente em tempo real, *ONNXRuntime*, para implementar um modelo já treinado num dispositivo com recursos limitados. Os resultados demonstram que *ONNXRuntime* foi a única ferramenta de *Machine Learning* com uma correspondência perfeita em relação à classificação das amostras dos modelos originais. Uma análise do tempo necessário para executar esta tarefa revelou que *ONNXRuntime* é mais rápido do que *Scikit-Learn* com os modelos ISO, OCSVM e SGDOCSVM e mais lento com o modelo LOF.

Acknowledgement

First of all, a special thank you to my family who despite all the adversities was always there to support me. A big thanks to my advisor Pedro Santos who guided me throughout my journey and who always gave me insightful support, always with a positive spirit. Also to thank the project that funded my work, POCI-01-0247-FEDER-069522 (MIRAI), co-financed by the European Regional Development Fund (ERDF), through the Operational Programme Competitiveness and Internationalisation (COMPETE 2020), under the PORTUGAL 2020 Partnership Agreement. I am also thankful about the team with which I had the opportunity to work with, to highlight professor Luís Almeida from Faculty of Engineering of the University of Porto. A final thank you to my closest friends, José Miguel, Carlos Daniel, Miguel Silva and Tiago Lino whose mere presence enlights my way.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

CPE	Costumer Premises Equipment.
Float	Floating Point.
ISO	Isolation Forest.
LAN	Local Area Network.
LOF	Local Outlier Factor.
ML	Machine Learning.
OCSVM	One Class Support Vector Machine.
ONNX	Open Neural Network Exchange.
pip	Pip Installs Package.
PMML	Predictive Model Markup Language.
RF	Random Forest.
SGDOCSVM	Stochastic Gradient Descent One Class Support Vector Machine.
SVM	Support Vector Machine.
TSTAT	TCP STatistic and Analysis Tool.

Chapter 1

Introduction

1.1 Problem statement

The evolution of machines has proven to a remarkable chapter in the human history as it plays a higher role in our everyday lives. New technologies and discoveries are made everyday and the achievements made in hardware have taken a huge leap since their creation as in instructions execution velocity, new components required to use the most recent technologies, etc. Nevertheless, the scaling of processors using Metal Oxide Semiconductor (MOS) transistors, the basic processing unit of Central Processing Units (CPUs) has reached a limit since their reliability is being compromised by well-known challenges [Igarashi et al. 2014].

1.1.1 IoT and raise in Cyber-attacks

In recent years, the world assisted to the beginning of the Industry 4.0, the digitization of the Industry. This *era* lead to inevitable increase of the number of devices connected to the Internet, making them reach new limits and one of the biggest contributors to this fact is Internet of Things (IoT) [Palandrani and Lucas 2022]. These devices are mainly used for data gathering and are one of the lowest security embedded devices available. This sudden raise in information circling in the Internet caught the attention of (potential) hackers to exploit these breaches to their interests. Cyber-attacks are not a new concept, as they have been used for decades and will be in others to come. Nevertheless, the number of cyber-attacks performed each year is increasing at a fast pace and "Global cyber-attack volume peaked in Q4 2022, with an average of 1,168 weekly attacks per organization" [News 2023]. The weak security features implemented in IoT devices and their wide range of applicability make them a very vulnerable device and can likely be the entry access point to many companies infrastructure. As so, overall IoT devices users and specially companies should update these device's security protocols to match their own in order to mitigate the possibility of an attack.

Machine Learning

Today machines posses more processing capacity than ever before and are being used to perform overall more sophisticated operations and more complex problems. One area of development that certainly requires such computational power is Artificial Intelligence (AI), Machine Learning (ML), Deep Learning (DL) and Artificial Neural Networks (ANN). This set of activities originates in AI, a technology aimed at mimicking the human behaviour. The use of statistical methods originated whats called ML and allowed the standard AI models to improve their performance with experience. So, the models are able to learn from the

data they are given and prefect their tasks. After ML, ANNs emerged as a competitive tool to handle ML problems. Based on a sequence of rows of neurons, ANNs training procedure translates into weighted connections between neurons of different rows that better achieve the expected results. DL represents a subset of ANNs that have more than the minimum number of rows (three), hence the *deep* concept. All this different yet similar approaches have their applicability is highly dependent on the technology used to train the models, in the resources available at the targeted device and in the overall balance between technologies limitations, effort and performance. Some developers may also want to analyze the power consumption of the developed models but the literature isn't paying that much attention to this challenge.

1.1.2 Security Concerns

Unfortunately, most IoT users overlook their devices security issues or are unaware of the threats they represent when poorly secured. Once a hacker is granted access to a network connected device, all the machines in that same network become compromised as they can be monitored externally and their data corrupted. To mitigate attacks from reaching their destinations, security mechanisms have been developed and are mainly present at the end devices (consumer devices). Nevertheless, the access to the Internet is granted through an access point (i.e router) and this device can host more advanced security features. So, as a second level of security, Costumer Premises Equipment (CPE) can provide a first scan and assess the integrity of the coming packets. This new service is being developed by some Internet Service Providers (i.e NOS), companies investing in their costumer's local security to protect them from external threats as traditional security mechanisms fail to detect and prevent the most recent attacks.

1.2 Contributions of this Thesis

This thesis contributed to the study of Machine Learning (ML) technology in devices with limited resources by investigating and exploring some ML tools with potential to be implemented in resource scarce devices. It also tested the time required by *ONNXRuntime*, a tool able to use Open Neural Network Exchange (ONNX) ML models, to execute the prediction process. This thesis results led to the following paper:

Pedro Vicente, Pedro M. Santos, Barikisu Asulba, Nuno Martins, Joana Sousa, Luis Almeida, "Comparing Performance of Machine Learning Tools across Computing Platforms," In 1st Workshop on Distributed Edge AI (DE-AI), part of the 18th Conference on Computer Science and Intelligence Systems (FedCSIS 2023), Warsaw, Poland, 18 September 2023.

1.3 Document Structure

The first chapter of the document provides a brief explanation on topics related to the thesis, its contributions and the structure of this document. Chapter 2 describes in more detail the specific terms and concepts introduced in chapter 1 and some conclusions on works already made in domains related to this thesis's. Chapter 3 defines the problem to solve as well as the methodology used to achieve it. It also describes the MIRAI project requirements and target devices. This chapter depicts the different ML deployment strategies and the plan developed to guide the ML tools exploration phase. Chapter 4 outlines the ML tools attempted implementations, results and the tool chosen to be implemented. Chapter 5

describes the time implementation with the chosen tool and its results. The final chapter (6) resumes the thesis results and conclusions, and enumerates some examples of related works that can be performed to improve the ML use/understanding.

Chapter 2

State of the Art

This chapter describes concepts required to understand the work motivation and challenges. Firstly, an explanation of the raise of Cyber-attacks is given followed by the most common ones performed. Then, an introduction of Machine Learning technology and embedded systems is made. In last, some machine learning libraries/frameworks to train and/or deploy machine learning models in embedded devices are listed followed by some works related to this thesis.

2.1 Internet of Things and Cybersecurity

The Internet of Things (IoT) concept describes the connection of typical isolated devices (i.e doors, motors, etc...) to the network and therefore to the Internet, to ease their management and control. In addition, it allows for a real-time analysis of the involved systems status, performance, possible problems, warnings and overall data. An IoT network is a distributed system composed of several edge nodes and one central machine (normally cloud hosted) to collect and manage the system's activity. Figure 2.1 represents a possible IoT network.

Figure 2.1: IoT Simple network structure [Sovrin 2022]

IoT devices are in their essence, embedded systems connected to the Internet and posses poor security defense systems. To perform their activities, the vast majority of IoT devices have low processing capacity at their disposal and are integrated with sensors and actuators since their tasks usually interact with the physical world (i.e read the temperature or rotate a motor).

Although IoT seems to be a recent concept, the first use of it dates back to 1982 with a coke machine and has became popular in 1999 [Greenberg 2021]. Since then, the number of IoT devices connected to the Internet has been growing at a fast pace. Figure 2.2 shows a prediction of the number of IoT connected devices till 2030 FinancesOnline.

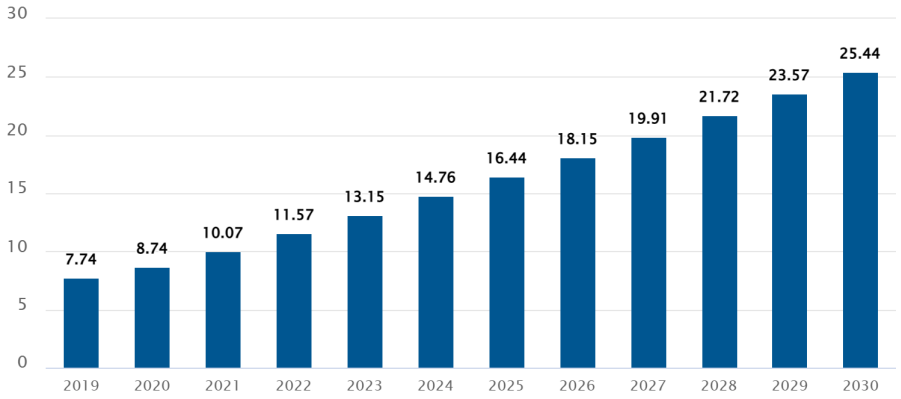


Figure 2.2: Expected growth of IoT devices connected to the Internet in billions adapted from [Jay 2022]

This huge rise in Internet connected devices is heavily increasing the amount of data circling in the Internet. So, hackers have more access than ever to sensitive information, the main reason why the number of attacks is still increasing by the year. "In the first six months of 2021, attackers caused more than 1.5 billion Internet of Things (IoT) breaches, up from only 639 million in 2020" [Howarth 2022].[1]

IoT lead to the introduction of typically isolated devices into communication networks, either in households (i.e smart fridges) or in industry contexts (i.e robotic arms). Through this integration of sensors and/or actuators , this systems can be monitored/controlled outside of the equipment's premises. Unfortunately, the majority of these devices lacks of security mechanisms able to prevent the most recent threats. Nevertheless, IoT devices should have the appropriate security levels since they can be used to:

- collect sensitive data - These devices may be monitoring personal information such as the heart rate of a hospitalized patient or the use of a power grid;
- perform activities in the real world - These devices can be accomplishing tasks in the physical world and an external agent can use them to damage their integrity or a business activity for instance;
- perform attacks - An infected device can be used by a malicious agent to perform all types of existing attacks to third-party machines.

2.1.1 Types of Attacks

As IoT devices can be performing a lot of different activities, the attention they are getting from the hacker community is bigger, threatening their integrity. There are a lot of different attacks that IoT devices can be subjected to and the following list categorizes and briefly describes some of the attacks mentioned in [Abdul-Ghani, Konstantas, and Mahyoub 2018], a comprehensive survey on IoT attacks:

[1] The live number of cyber-attacks can be consulted at Check Point thread Cloud a security company focused in protecting their costumers from cyber-attacks.

- Physical-based attacks - Attacks targeting the IoT devices hardware
  - Object replication attacks: An attack where a new device is added locally to the network and identity replication performed to gain access to the network;
  - Malicious code injection: This attack is performed when malicious code is manually injected into a IoT device;
- Protocols-based attacks - Attacks targeting information circulating on the Internet
  - Sniffing attack: An attack where a device is monitoring the traffic in order to capture sensitive information which application protocols don't posses security mechanisms;
  - TCP-UDP Port scan: An attack performed to discover listening ports in reachable devices to compromise them;
- Data at rest-based attacks - Attacks targeting IoT devices in order to access the stationary data on them or in the cloud
  - Data exposure: An attack where the lack of encryption between IoT devices and the data centers leads to external access to the latter through the first one;
  - Brute-force attack: An attack where a automated program tries to decrpypt ciphertext to discover sensitive information;
- IoT Software-based Attacks - Attacks targeting the IoT device's application, operating-system or firmware
  - Distributed Denial of Service (DDoS) - An attack where a resource is compromised by an overflow of requests;
  - Phishing attack - An attack where sensitive information is gathered through social media, email or phones

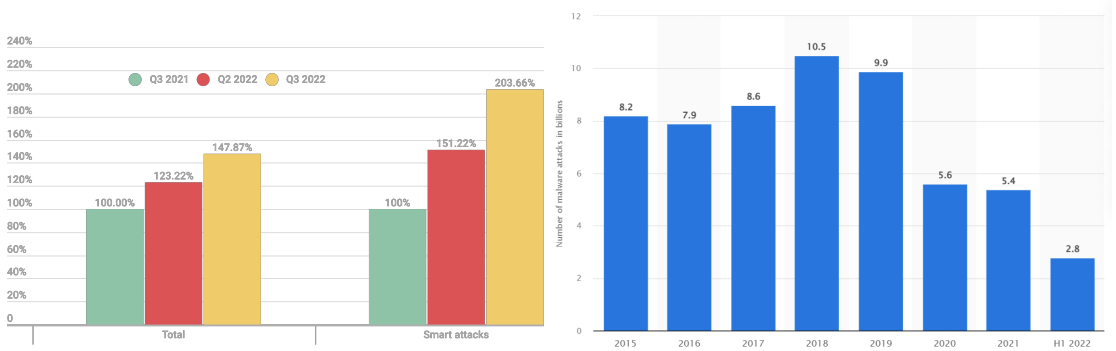
2.1.2 Selected Attacks

Since nowadays there are multiple attacks to achieve the same goal, the attention given to each of them relies on their efficiency (effort versus reward) and some attacks are more difficult to perform then others. The following list presents and briefly describes the most common cyber-attacks performed in recent years according to [Baker 2021; Tietsort 2022]:

- Ransomware - Access to user's system is denied and a ransom required by hackers to eliminate the restrictions that the latter created.
- Malware - User's machine is infected with a malicious program designed to harm the device.
- Malware as a Service (MaaS) - Anyone can perform a Malware attack by hiring hackers to execute them against a third-party (no exposure technically).
- DoS and DDoS Attacks - Online resources or services are flooded with false requests to deny access to legitimate users. While DDoS attacks are performed with multiple devices (strengthening the attack), DoS attacks use only one.
- Phishing - Attacks that recur to communication mechanisms to entice a victim to grant access to sensitive data or download malicious files.

- Man in the Midle (MITM) - An attack were a malicious actor is listening to the traffic between a web application and a network user to possibly discover confidential information. It might even impersonate one party to extract more information.
- Cross-Site Scripting (XSS) - Malicious code is inserted into websites. Then, when users access the infected website, hackers might have access to personal information or steal the user's session.
- Structured Query Language (SQL) Injections - A data-driven application is injected with mischievous SQL statements that will later provide the hacker with access to the database.
- DNS Tunneling - Domain Name System (DNS) queries and responses are used to evade the security mechanisms and transmit information in the network. An hacker can then use DNS responses to perform malicious activities.
- Password Attack - A program designed to brute force authentication interfaces to discover user's passwords. Attack motivated by the users' lack of concern about the security of their passwords.
- Drive By Attack - Explores applications vulnerabilities to run a malicious script that will download and install a program without any permission from the user.
- Cryptojacking - Attack where one's resources are used without permission to mine cryptocurrency.

The following images show some statistics about the occurrence of some of the attacks mentioned in the previous list in recent years:



(a) DDoS reported attacks [Kaspersky 2022]

(b) Frequency of Malware attacks [Department 2022]

Figure 2.3: Growth and volume of cyber-attacks

Apart from the attacks that are detected and eliminated in their target's defense systems, their impact can be translated into monetary losses. Following the increase frequency of attacks is the cost associated with them. The following image shows the evolution of the cost of cyber crime in recent years.

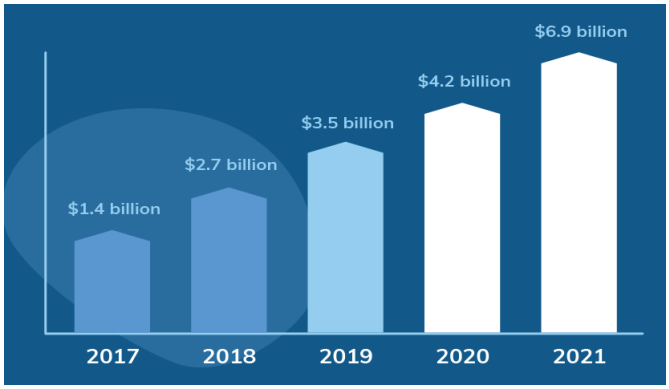


Figure 2.4: Cost of cyber crime from 2017 to 2021 [Lukehart 2022]

2.2 Machine Learning

In the 1950s, Artificial Intelligence (AI) emerged, a technology with the aim of creating machines that could mimic human behavior. Thirty years later, a new concept was created, Machine Learning, a subset of AI techniques which use statistical methods to enable machines to improve with experiences, as described in figure 2.5. In other words, ML based systems can learn with time to better execute the tasks they were given.

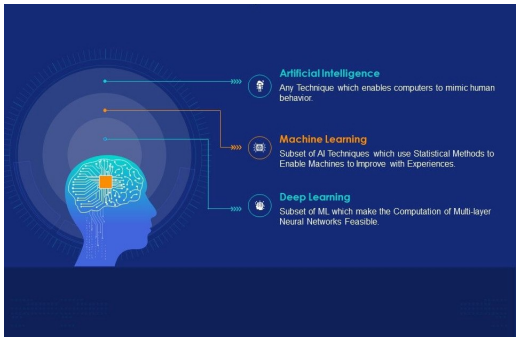


Figure 2.5: Machine learning context [Bahri 2020]

ML technology is based in models that are trained to accurately accomplish the tasks they were given. Figure 2.6 represents the typical procedure of ML. First, a good representation of the possible information it might be analyzing must be gathered. Then it is a good practive to divide the dataset into two different datasets. One for training the ML models called *training dataset* and one for testing the trained model called *test dataset*. As the accuracy of the ML models depends the most in the quality of the *training dataset*, a special care in the data used for model training is required, thus many ML developers insert refining stages in the ML process with the name *Data pre-processing*. There are a lot of different ML models implemented and their choice also has a great impact in the model's performance. The working principles of the models differ from each other, although some of them derive from others, and must be studied. After training the model, it is deployed in the intended machine and the *test dataset* inserted to assess the performance of the trained ML model. There are a lot of results and parameters that can be analyzed from the execution of a ML model, but the attention resides mainly in the accuracy of the model. This specific parameter ideal value would be 100%, meaning that it was capable of correctly predict all the given instances. Due to the fact that most ML models first accuracy test doesn't correspond

to the expectations, they can be retrained with new or improved data till they provide the expected results.

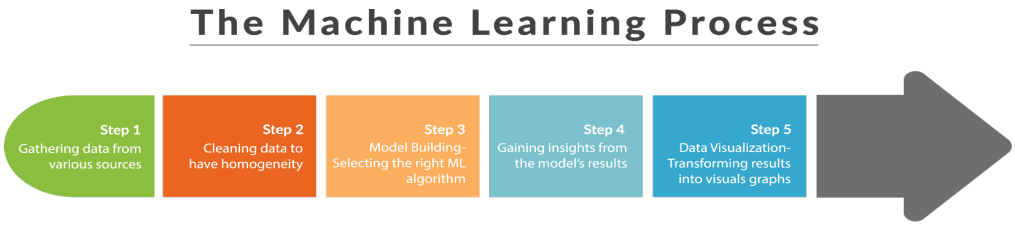


Figure 2.6: Typical procedure for ML model creation [Diskin 2020]

2.2.1 Types of ML algorithms

There are a lot of ML algorithms already implemented and they can be distinguished by their strategy to achieve the best possible outcome and their need for labelled data to properly distinguish them in classification algorithms.

Supervised learning represents a subset of ML models that require labelled data. This group of algorithms can be further divided into two categories: classification algorithms thrive to create a function that best separates all the different data based on their labels; regression algorithms attempt to make predictions considering the values they were given.

Unsupervised learning contains algorithms that don't require data labelling. The utility if this type of models lies on their ability to discover patterns/relationships between the given data. Their results might report interesting features that might provide an improvement on the comprehension of the *training data*.

Semi-supervised is a combination of supervised and unsupervised algorithms and so can be considered a two step process. The first step is the use of an unsupervised model to associate similar data into clusters. Then, the presence of at least one labelled input in each cluster (hence being supervised as well), can be used to generalize the classification to all data points in the same cluster. This strategy has a lower execution time than supervised learning as the majority of the input data is unlabelled, reducing the processing required.

Reinforcement learning is a reward based type of ML. This subset of ML models predictions are made in order to maximise the reward or to minimise the penalty.

2.3 Machine Learning for IDS in Embedded systems

The embedded system concept describes a system which has a defined set of hardware components available to use. These components must allow the system to perform its activities in the expected way. "Embedded systems are information processing systems that are embedded into an enclosing product" [Marwedel 2021]. Many of these devices can be further classified as real time devices as their activities must be completed by a certain deadline. To better manage these system's tasks, Real Time Operating Systems (RTOS) have been developed to meet the imposed restrictions. In addition, this devices can be placed in harsh environments with to connection to any power grid for a continuous energy supply and often rely on a small battery as a power source to perform all their activities. As so,

the development of the sequence instructions of the programs to be inserted in these type of systems should also be energetically optimized as possible. There are endless applications of Machine learning technology and the main problem of reaching an external device for inference tasks is *time latency*. As stated before, embedded systems are widely used with real time constraints and outsourcing a device to perform the inference tasks would require a constant and reliable connection. Unfortunately, Internet connections aren't stable and the time required for the request to reach its target, to be processed and the response sent back, might be too long to meet the embedded device's deadlines. Even if the elapsed time in a outsourcing activity is acceptable, critical applications would require a warranty that the worst possible time to perform this operation would still meet the setted deadlines. As this warranty can't be granted, it is impractical to rely on cloud computing.

2.3.1 Challenges of ML in resource constrained devices

- Accuracy represents possibly the most important parameter of a ML model since it represents the balance between the correct and incorrect predicted data. With it, one can assess how well the model performed against the new information.
- Memory Footprint is also important to analyze. As embedded systems have low storage capacity, ML models can be subjected to an optimization process to reduce their memory size. Nevertheless, this optimization can result in accuracy losses.
- Execution time simply states how much time the inference process required to perform the predictions. This time is dependent on a set of features like *clock frequency*, *CPU Architecture*, *Programming languages*, etc.
- Power Consumption has less attention than the three previously mentioned features as embedded systems consume less energy than others devices (i.e desktops).

2.3.2 ML Libraries in Low Level Languages

Machine learning is most commonly used in high-level programming languages (i.e Python). This subset of languages allows for a rapid program development but it's instructions are further apart from the CPU ones requiring more translations along the way. Translating instructions from one language to another is a time consuming task and as so, low-level programming languages represent a more attractive solution to efficiently run machine learning models in resource constrained devices.

There are a few free use libraries publicly available to create machine learning models and the following list describes some interesting libraries discovered to develop machine learning models in the languages C/C++ or to embedded devices:

- TensorFlow is an open-source library for Artificial Intelligence (AI) and Machine Learning (ML). This library is integrated with:
    1. Datasets and pre-treined models developed and released by the Tensorflow Community;
    2. Tools to help the community to use Tensorflow. Colaboratory (Colab) for instance, is a free jupyter notebook environment and runs in the cloud so the user doesn't need to setup anything in his local machine;

3. Libraries and extensions with advanced features to help the Tensorflow users to achieve their goals;

This library is also supported in Haskell, C#, Julia, R, Ruby and Scala languages by the community and in Javascript by Tensorflow. This library also has a lite version *Tensorflow Lite* to allow for ML models to run on edge devies (i.e resource constrained systems). This tool allows for a model optimization, easing the computational requirements of the target system but its accuracy can be compromised if it uses operations not supported by the *Tensorflow Lite*. Cross-compilation for specific operating systems (OP) is also possible, allowing the models to be trained in the cloud, where there's no lack of processing capacity, and further deployed in the edge devices. For running *Tensorflow Lite* models the target devices don't require the full library installation, just the *tflite_runtime* to be able to make data inferences.

- Armadillo is a library in C++ for linear algebra and scientific computing. It tries to reach a good balance between ease of use and speed and can automatically use Open multi-processing (OpenMP), a free easy to use library for parallel computing. It allows for the development of algorithms in C++ and thrives for programs efficiency through dynamic evaluations;
- mlpack is a C++ ML library focused in providing fast and extensible implementations of ML models. This library is the combination of *Armadillo* (previously described), *ensmallen*, a library for numerical optimization and *cereal*, a serialization library.
- Shogun is an open-source library in C++ for machine learning development. It provides interfaces for C++, Python, Octave, R, Java, Lua, C#, Ruby and implements all the standard ML algorithms and some advanced as well. It has a good execution speed and is available for the most used operating systems;
- SHARK is an open-source machine learning library implemented in C++. It provides neural networks, kernel-based learning algorithms, linear and nonlinear optimization methods and is available for the most common operating systems.

2.3.3 Interoperability of ML models

Interoperability of ML models refers to their portability between different libraries. In the ML domain this is a challenge since the majority of the frameworks and tools developed to build and train such models weren't built to be understandable by other libraries. Despite this, the adoption of a global format that describes ML models is natural in the goal of accomplishing the interoperability of the libraries trained ML models as the development of tools to translate each model's processes for each pair of ML libraries is just unpractical and unfeasible. There are already some developed mechanisms to allow models that were trained in one framework to be used by another one and the following list describes some of these mechanisms.

- Open Neural Network Exchange (ONNX) is a community project focused on building an open standard representation of ML models. ONNXRuntime is a tool developed to use ONNX models and was created to improve the overall ML developer experience by reducing the time required for model training and model inference. By being mostly written in C++ and C, low level languages, it already takes less time to perform these type of operations compared to implementation focused libraries. These second libraries are usually developed in high level languages like Python to ease the user's

implementation process so results can be achieved faster. In addition, ONNXRuntime is able to use hardware accelerators and perform graph optimizations to further increase its performance. It is adopted and supported by a considerable amount of different ML tools: at least 29 frameworks and converters and 30 inference runtimes like Scikit-learn, Tensorflow, OpenVINO, Keras, and so many others.

- Sklearn-porter is a python library specifically developed to transpile ML models built with Scikit-Learn to other programming languages such as *C*, *GO* and *JavaScript*. Transpile refers to the process of translating a program to a language different from the one used originally.
- Predictive Model Markup Language (PMML) is a document format based on the Extensible Markup Language (XML) that can be used to described machine learning algorithms[Guazzelli et al. 2009]. There are some libraries developed to work with PMML ML models like cPMML in C programming language and sklearn2pmml in Scikit-Learn library.
- Model 2 Code Generator (m2cgen) is a library mainly developed in Python, that transpiles ML models implemented with Scikit-learn or lightning libraries to other languages. This tool can transpile models to at least 16 different programming languages like *R*, *Visual Basic*, *Haskell* and *C#*. Information available on the Github repository m2cgen.

2.4 Related Work

The use of ML algorithms in embedded devices isn't the most explored application of ML technology, mainly due to their few resources. As such, this section presents some works developed in recent years to assess the performance of embedded systems in activities besides ML. The works address the performance of the systems taking into account the time required to perform the tasks and the memory usage for non ML works and the models accuracy and speed for ML related works.

2.4.1 Time performance of Embedded Devices

In [Magnani et al. 2021], the authors investigated the effect of precision tuning arithmetic operations in motors drive control. The authors compared the execution of the full code with all the platform-specific libraries against a reduced version with a Field Oriented Control (FOC) application. The results show a speed improvement of 278% with a less than 0.1% output error.

In [Silva et al. 2016], the authors studied the performance of cryptography algorithms in embedded systems. This study was performed in a Sony VAIO Laptop and in an Overo EVM pack as a resource constrained device. They analyzed the memory, CPU and time consumption of several algorithms. The results show that the embedded system required a lot more time and less memory to perform the same encryption and decryption tasks.

In [Cifredo-Chacón et al. 2019], the authors compared the speed performance of FPGA and processor based solutions to implement a spectral kurtosis (SK) function, an example of a complex digital signal processing function. XC6LS16 and XC7A100T are the two FPGAs used, implemented in Nexys3 and Nexys4 evaluation boards. Intel Atom N270, ARM Cortex-A7 and an Intel i7-3517U are the processors used as a comparison. The results reported

show that a FPGA based solution can perform a typical SK function between 34 and 1740 times more efficiently than processors. So, for SK operations, even working at a lower frequency, the FPGA technology requires less time than general-purpose processors.

2.4.2 Energy Efficiency of Embedded Devices

In [Viegas, A. O. Santin, et al. 2017a], the authors proposed a tool to predict the best voltage and frequency of the CPUs power source to achieve the desired performance. This study was conducted on a multi-core NVIDIA JETSON Tegra K1 platform running the Linux operating system and the results show that with a 10% loss in time efficiency, the system can reduce its energy consumption from 4.1% to 6%.

2.4.3 Recent Application of ML in Embedded Systems

In [Society et al. 2017], the authors successfully implemented a real-time system to detect pedestrians based on thermal images on a Raspberry Pi 3 Model B containing 1.2 GHz quad core processors. Through the use of foreground segmentation based on histogram equalization in half-size images and further identification of zones of interest, the authors reported a time reduction about 3.4 times compared to the traditional Histogram of Oriented Gradients (HOG) implementations. With the optimizations implemented, the Raspberry was able to process more than 10 frames per second (fps) "while keeping the high accuracy" [Society et al. 2017].

2.4.4 Performance of ML models in Cybersecurity

In [Vinayakumar et al. 2019] the authors compared Deep Neural Networks (DNN) performance against classical machine learning approaches. The DNN was implemented with Keras and GPU enabled Tensorflow as backend and the machine learning algorithms with Scikit-Learn. Both implementations were deployed in Python programming language. Network Intrusion Detecting system (NIDS) was trained and tested with KDDCup 99, NSL-KDD, UNSW-NB15, Kyoto, WSN-DS, CICIDS2017 datasets. DNNs were tested with 1 through 5 layers of neurons. It was concluded that DNNs surpassed the performance of classical machine learning approaches.

In [Eskandari et al. 2020] the authors used Isolation Forest and Local outlier Factor, two one-classification machine learning algorithms to evaluate the network traffic. A local testbed was built and network traffic collected to train the models (assuming it wasn't under attack) and Metasploit used to perform attacks to the IoT devices. The attacks performed were Port Scanning, HTTP Brute force, SSH Brute Force and SYN Flood. the results show that Isolation forest model performed better then Local outlier factor and the F1 score oscilating from 0.79 to 0.99.

In [Szakál et al. 2017] the authors used KDD dataset to train and test the performance of several machine learning algorithms (J48, Random Forest, Random Tree, Decision Table, MLP, Naive Bayes, and Bayes Network). The test was performed on a Ubuntu 13.10 platform with a Core(TM) i5-4210U CPU at 1.70GHz (4CPUs) and 6 GB RAM. Results showed that none of the algorithms could detect all types of attacks. Bayes Network classifier had the highest TN rate and the Random Forest had the best accuracy with 93.77%.

In [Viegas, A. Santin, et al. 2018; Viegas, A. O. Santin, et al. 2017b] similar studies were made to asses the energy efficiency of resource constrained devices. The data was generated with HoneyD, a honeypot tool, hosted in a single machine to generate real service responses and one hundred automated hosts were deployed to generate the service requests. The responses were then stored in a pcap file. The [Viegas, A. O. Santin, et al. 2017b] study focused in DDoS attacks, namely in SYNFLOOD, UDPFLOOD, ICMPFLOOD and SLOWLORIS attacks, while the [Viegas, A. Santin, et al. 2018] study focused in scan attacks (i.e UDPScan, TCPConnect, ...) both generated through well-known tools. The model training-phase used 25% of the generated attacks, 25% for model validation and 50% for testing.

To deploy the stored packets the authors used TCPReplay tool. The system was tested with a DN2800MT Atom motherboard with an Intel N2800 CPU (4 GB DDR3 RAM and a 500 GB hard drive) as a control test and in a FPGA board as comparison. The power consumption of the non-restrained system was calculated through the sum of the time the CPU was performing network traffic classification tasks by the means of an external signal sent to a data acquisition device. In the FPGA the power consumption was assessed through the Altera's Power Monitor tool.

Both works show a good accuracy of the used machine learning models for the attacks they were trained with (> 99%). The [Viegas, A. Santin, et al. 2018] further explored this topic using similar and unknown attacks of the ones used in the models training-phase and results report a good detection rate for similar attacks (> 98%) and for unknown attacks (> 99% only with a high rejection rate).

Both implementations of the feature extractors in the FPGAs are faster than the control board and require less energy. All the classifiers used in both studies reported a lower consumption in the FPGAs when compared to the control values. Overall, the FPGAs consume less energy than the DN2800MT Atom motherboard to perform the same tasks. In [Viegas, A. O. Santin, et al. 2017b], the lowest consumption implementation in the control board required approximately 17.5 times the energy of the FPGA equivalent and in [Viegas, A. Santin, et al. 2018] the FPGA required 42,6% of the energy required by the control board.

2.4.5 Limitations found

Timely response is a very important if not critical aspect of embedded devices. The works developed about the performance of ML technology in the cybersecurity domain reside mainly on the classification models accuracy for intrusion detection tasks. The application of IDS in embedded devices merges the two concepts and so, a careful analysis of the time required for embedded systems to achieve a certain accuracy is a must and is lacking in the literature. This thesis aims to fulfill this gap by evaluating the relationship between time and accuracy in embedded systems.

Chapter 3

Goal Description and Initial Analysis

This chapter describes in more detail the proposed approach to reach the end goal and reports the requirements imposed by the MIRAI project. It also describes the target devices main characteristics as well as the different deployment strategies used by ML technologies. It also presents a study on the presence of the MIRAI's ML models of interest in the tools introduced in chapter 2 and the order in which the identified ML tools will be explored.

3.1 Goal and Approach

Cybersecurity is a worldwide problem and mechanisms to identify and protect devices from external attacks has been studied for decades. Machine learning is proving to be a promising technology to implement these functionalities but it is a computationally demanding activity that wasn't designed for hardware constrained platforms. Cybersecurity can be classified as a real-time activity and the reliability of resource-scarce devices performing these tasks requires further study. This thesis aims to assess the feasibility of such tasks in resource scarce devices by comparing the time required by different systems.

In order to assess the feasibility of ML technology for network analysis in low-resource devices, it is important to correctly quantify the time required by the implementation to classify any packet received since its arrival. Since the tool to be used for the packets analysis is already defined, this time interval will be heavily affected by the prediction time required by the ML implementation used. So, an evaluation on the ML existing tools for scarce devices will be mandatory to then proceed with the real time implementation of the system. The figure 3.1 represents the implementation workflow of the proposed system.

Figure 3.1

While the network traffic is arriving at the Local Area Network (LAN), its packets are being analyzed by a tool that reports the type of flows[1] identified and their characteristics. These values must be available as soon as possible as they need to be processed and inserted in the ML model to obtain a classification.

[1] In the Network domain *flow* refers to a sequence of packets exchanged between a defined set of devices resembling a logical connection[Wikipedia 2023b]

3.2 MIRAI Requirements

This section describes the requirements to be taken into account in the development of this work.

3.2.1 ML Models selected under the MIRAI project

The MIRAI project is an on going project for some years and ML models have already been selected and implemented in *Scikit-Learn*. This library is implemented in Python programming language and is simple to use and supports a wide range of known ML models. The list bellow describes the implemented models:

- One Class Support Vector Machine (OCSVM) - A unsupervised model that uses an hypersphere to distinguish between two types of data. This model tries to fit the X number of points requested that translates to the least volume in the hypersphere. A variation of this method, the Stochastic Gradient Descent (SGD) OC SVM, was also tested.
- Isolation forest (IF) - A model where the data is asked a fixed set of *yes* or *no* random questions from the features group. This process is usually referred to as tree.
- Elliptic Envelope (EE) - A unsupervised model that creates a ellipse around the data. The data used to train the model has to be though through as the predictions that lie inside the ellipse are considered normal and the rest as abnormal.
- Local Outlier Factor (LOF) - A model used in anomaly problems that uses the points density to assess its classification. This model compares the density of the k nearest points. If the density of the test data is smaller than the nearest points, it is considered to be an anomaly. This model requires a good amount of normal points closed together to correctly perform predictions based on this method.

3.2.2 Target Device

The low resources of the target device, NOS Costumer Premises Equipment (CPE), represent a major aspect to be taken account for in the system development process. As so, a special care was required to verify if the target system could actually comply with all the requirements necessary to install and use the chosen ML technology.

3.2.3 Packet Analysis Tool

The tool selected to capture the network traffic and provide measurements about the connections identified is TCP STatistic and Analysis Tool (TSTAT). This tool provides a set of features regarding each network flow that define the flows characteristics. This information is saved by TSTAT as logs, a simple text file with the active features followed by their values for each new entry. TSTAT as some flags in *runtime.conf* file in *tsta-conf* folder that allow the manipulation of the sets of features reported by the logs. Each flag is associated with a set of features and their activation or deactivation will define which features are to be printed.

3.3 ML Deployment Strategies

The ML behaviour can be implemented in various different ways and the tools mentioned in 2.3.2 and 2.3.3 can be distinguished by their working principle:

- Interoperability (*ONNX* and *PMML*) - This implementation section describes the translation of ML models from one format to another. This type of implementation is very interesting as it allows performance comparisons of different libraries and runtimes (as long as they understand one of the supported formats). Nevertheless, tools are required to execute the translation process that will limit the usable ML models. The Figure 3.2 displays the process to convert a ML model from one technology to another and their use. The original ML model is first converted to a widely adopted type format and then translated to the target technology. The data can then be given to the newly converted ML model and the predictions calculated.

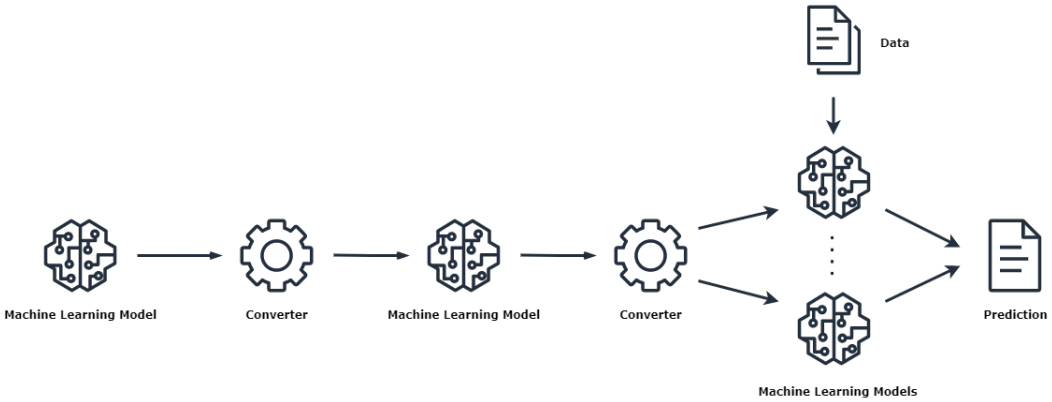


Figure 3.2: Inference process of a interoperability solution

- Runtime (*Tensorflow* and *ONNXRuntime*) - A ML runtime implementation describes a tool were resources are allocated to run code in real time [Rouse 2020]. It is described as an environment were resources are being correctly allocated when needed. The trained ML models are represented in a known format type (like binary) and the engine will just perform the expected operations that describe the model behaviour on the given data and return the predicted values. This implementation strategy can be more efficient than library based ones if implemented in low level programming languages. Figure 3.3 displays this implementation flow. After training a ML model, the runtime engine is given the model's characteristic processes, the dataset and proceeds with the given instructions which will result in the expected predictions.

- Transpiler (*Sklearn-porter* and *m2cgen*) - In the ML domain, transpilation refers to the process of converting ML models into a code representation in another language than the one used to train the model in the first place. Since it can be transpiled to lower level programming language, the transpiled code will take less time to execute than a ML model using a library implemented in a high level language. This implementation requires almost no memory compared to the previous implementations. Figure 3.4 represents the general process to convert and use a ML model transpiled code. A transpilation tool is used against a ML trained model resulting in a code description of the ML model's behaviour. This code can then be given data and print the calculated predictions.

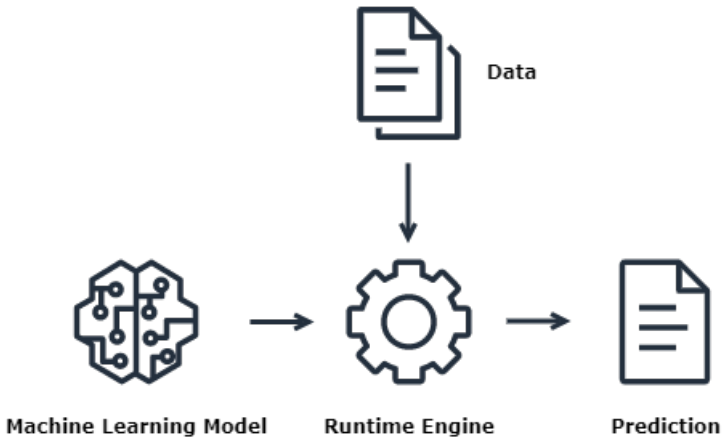


Figure 3.3: Inference process of a runtime engine based solution

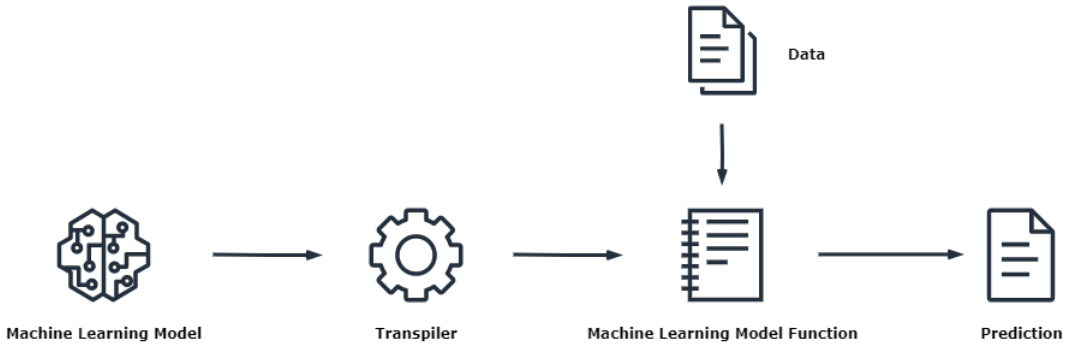


Figure 3.4: Inference process of a transpile based solution

- Libraries (*mlpack*, *Shogun*, *SHARK*, *Armadillo* and *Tensorflow*) - A ML library can be very inconvenient in resource constrained devices. Besides the ML model size, the library itself can prove to occupy more memory than available making it unpractical to implement. Nevertheless, libraries size can be reduced by performing a refinement process on the model's required functions. By tailoring down the exact steps used by a specific ML model, one can only implement the necessary functions and still achieve the same goal and performance. The Figure 3.5 represents the standard flow of information in a library implementation. The data is given as input to the already trained model and the predictions obtained.

3.4 Initial Analysis of Tools and Definition of Exploration Strategy

The MIRAI project ideal solution is to deploy the already trained ML models in Scikit-learn through a lighter yet reliable ML tool. Despite the importance of the choice of libraries and tools working principle described in Section 3.3, it is also crucial that the selected tool(s) can implement or execute inference processes on the ML models selected under the MIRAI project. The following tables present the results of the study made to understand if the low level libraries listed in Section 2.3.2 implement any of the ML models of interest (1) and the results of a similar study but for the interoperability tools (2).

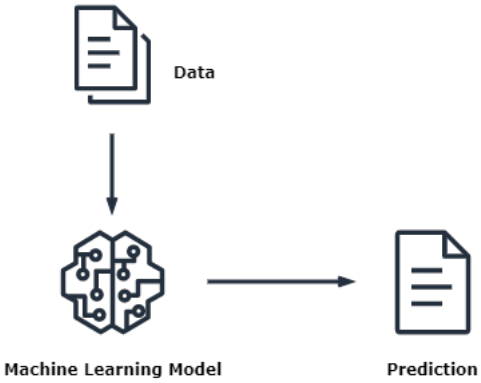


Figure 3.5: Inference process of a library solution

Table 3.1: Presence of the ML models of interest on the low level libraries mentioned

Library \ML Model	SVM	OCSVM	Random Forest	Isolation Forest	Elliptic Envelop	Local Outlier Factor
TensorFlow	Unspecified					
mlpack	Linear SVM	Not found	Yes	Not found		
Shogun	Yes	Not found	Yes	Not found		
Shark	Yes	Yes	Not found		Ellipsoid	Not found
Armadillo	Yes (git project)	Not found	Yes (git project)	Not found		

Interoperability Formats

From the tools present in Table 3.2, the deployment of the ML models through the use of a Predictive Model Markup Language (PMML) ot ONNX intermediate description was explored. The first solution requires a tool to convert the *Scikit-learn* trained models to *PMML* and a tool developed in a low level language able to understand and use the model's *PMML* description. The sklearn2pmml tool was developed to specifically convert *Scikit-learn* models into *PMML* and cPMML to read and execute the prediction process on ML models serialized with PMML. The apparent ease of use demonstrated in both tool's github repositories was the main reason to explore this solution in the first place. The second solution used *Tensorflow Lite* runtime as the target ML tool to test the model conversion from ONNX to a different format as there are tools that provide this process. To achieve this, first the Scikit-Learn ML models need to be translated to ONNX with the Sklearn-ONNX tool and then converted to the Tensorflow supported model. Protobuf (Protocol Buffers) is the file format used by TensorFlow to save and load ML models [Cloud 2023].

Transpilers

As transpilers create a code script that would ideally produce the same result as the original ML model requiring less time then other solutions, they were tested as well. In Table 3.2 two transpilers are mentioned, *Sklearn-porter* and *m2cgen*. From these two tools, *m2cgen* supports more programming languages, more ML models and appears to be easier to use. As such, *m2cgen* was the transpilation tool explored.

Runtime Environments

ONNX also has its own tool to run ONNX ML models, *ONNXRuntime*, and provides an optimizer that can, depending on the model, perform model optimizations and therefore

Table 3.2: Presence or capacity to use the ML models of interest on the mentioned interoperability tools

Tool \ML Model	SVM	OCSVM	Random Forest	Isolation Forest	Elliptic Envelop	Local Outlier Factor
ONNX	Depends on the version of itself and *skl2onnx*					
Sklearn-porter	SVC, Nu-SVC and Linear SVC	No	Yes	No		
m2cgen	SVC, Nu-SVC and Linear SVC	Yes		No		
PMML	Unspecified		Yes	Unspecified		

needing less time to execute. *ONNX Runtime* tool was used to test the ONNX models, alongside the ONNX optimizer to assess the impact on time performance of the optimized models when compared to the original ONNX models.

ML Libraries

From the tools present in Table 3.1, it is clear that none of the libraries cited implement all the ML models selected. Between them, TensorFlow is the library with most support from the community and has an Real Time Environment for low resource devices (TensorFlow Lite). *SHARK*, *Armadillo* and *Shogun* latest releases date back to 2018 so, it seems that not much work is being done (or at least available for the public). In addition, *mlpack examples* folder provides several examples on how to use the supported models easing the implementation process. Between the three remaining libraries, all of them implement the Support Vector Machine (SVM) model and one more (either One Class Support Vector Machine (OCSVM) or Random Forest (RF)).

The ideal solution would be to transfer the original *Scikit-Learn* ML model to the target libraries mentioned above. So, an investment was made to investigate the existing interoperability formats. Since they showed a good potential, these ML libraries ended up not being explored.

Chapter 4

Assessment of Potential ML Tools

This chapter describes the effort made to implement the tools identified in chapter 2. The time available to explore these tools was limited so, the results and conclusions reported in this chapter are based on a light research. Some of the conclusions extracted from the achieved results may not correspond fully depict each tool's potential, but surely are a pre-diagnosis for other developers interested in the tools explored.

4.1 Python implementation to C++ through PMML

PMML is a document format able to describe ML models. This subsection depicts the attempt made to have a *Scikit-Learn* ML model being used in a lower level programming language and assess if the predictions made by each programming language match. The Figure 4.1 describes the ML model transformation process from Scikit-Learn to C++ programming language.

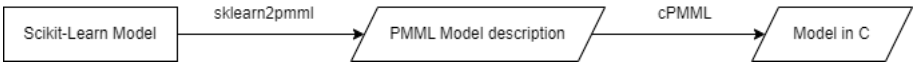


Figure 4.1: ML Model conversion process from Scikit-Learn to C

4.1.1 Scikit-learn to PMML

First, the Scikit-Learn trained models need to be transcribed into a PMML description. The code snippet in listing 4.1 displays a simple example available at the *sklearn2pmml* Github repository used to save a *Decision Tree* pipeline[1] as a PMML file.

[1]In the software domain *pipelines* consist of a sequential application of processing units [Wikipedia 2023a](transforms and estimators in Scikit-Learn)

```
1  import pandas
2  import numpy as np
3  from sklearn.tree import DecisionTreeClassifier
4  from sklearn2pmml.pipeline import PMMLPipeline
5  from sklearn2pmml import sklearn2pmml
6
7  iris_df = pandas.read_csv("Iris.csv")
8  iris_X = iris_df[iris_df.columns.difference(["variety"])]
9  iris_y = iris_df["variety"]
10
11 pipeline = PMMLPipeline([
12   ("classifier", DecisionTreeClassifier())
13 ])
14
15 pipeline.fit(iris_X, iris_y)
16
17 sklearn2pmml(pipeline, "DecisionTreeIris.pmml", with_repr = True)
18
19 # Test the model
20 data = {'petal.length':[4.6],
21         'petal.width':[1.3],
22         'sepal.length':[6.6],
23         'sepal.width':[2.9]
24         }
25 df = pandas.DataFrame(data)
26
27 Prediction = pipeline.predict(df)
```

Listing 4.1: Simple algorithm to convert a Decision Tree pipeline to PMML format (Python)

Since the previously mentioned repository doesn't provide the dataset mentioned in the example, a similar public dataset was used (*Iris.csv*). To load the selected dataset into the program, the *read_csv* function from the *pandas*[2] python package was used. This function main parameter is the path to the file of interest, "filepath_or_buffer" and returns the input data as a Dataframe, a two-dimensional data structure with labeled axes.

The dataset used possesses five different features: Sepal length, Sepal width, Petal length, Petal width and Variety. So, it is necessary to separate the training features from the ground truth to be predicted by the ML model. This feature distinction is saved in "iris_X" and "iris_y" variables in lines 8 and 9 respectively. The first one holds the training features and the second the ground truth. Since the dataframe is labeled, the feature selection process is easily done by selecting the names of the columns of interest. "iris_X" uses the *columns* method to select all the available features followed by the *difference* method to hold the non passed features (Stores every feature but "variety"). The "iris_y" only holds the "variety" feature.

Besides the preparation of the data to be used by the ML model, it is also essential to define the pipeline structure. To achieve this, *sklearn2pmml* package has a function named "PMMLPipeline" to define its parameters. The only required input of this function is named "steps" and corresponds to a list of tuples, where each tuple has a *name* and a *transform*. This list is interpreted sequentially, so the first tuple on the list contains the first *transform* to be applied in the pipeline and the last tuple contains the final *transform*, which must

[2] *Pandas is a tool written in Python programming language for data analysis and manipulation [Official Site]*

be an *estimator*, to be imposed. *Transforms* and *estimators* are two concepts mentioned by Scikit-learn documentation. So, whilst *transforms* are only able to make data transformations, $X \xrightarrow{Transform} F(X)$, *estimators* can make predictions based on the input values, $X \xrightarrow{Prediction} Y$.

In the attempt made, the list has only one tuple element, defined directly in the calling of *PMMLPipeline* function. The *transform* used was the Scikit-Learn *estimator* for the *Decision Tree* ML algorithm, *DecisionTree*, with the name "classifier".

After creating the pipeline, it is possible to train the ML model with data. In Scikit-Learn, this process is mentioned as *fit*. This function requires the *data* to be used in training process. In the example, the *data* is represented by the variable "iris_X" and the "iris_y" corresponds to the feature to be predicted by the model. Once the ML model has been successfully trained, it can be saved as a PMML file. To do so, *sklearn2pmml* package provides a function called *sklearn2pmml*. The parameters required for this function are the pipeline itself, stored in the "pipeline" variable, and the name of the *pmml* file where to store the pipeline, "DecisionTreeIris.pmml".

A simple dictionary[3] type structure named "data" was created, and a value was given to each of the features used in the ML model training phase. This structure was then converted to a dataframe through the use of the *Dataframe* method from *pandas* library and stored in the "df" variable. This method accepts a limited set of data structures and returns a dataframe object containing the same information without compromising its integrity. At last, the *prediction* method was used to make a prediction on the "df" sample. In the Iris dataset used, the "variety" feature has three different classifications: Setosa, Versicolor and Virginica. The Decision Tree model predicted the "df" sample as "Versicolor".

4.1.2 PMML to cPMML

Once the pipeline is saved in a PMML representation, it can be loaded into C++ programming language with cPMML tool. The listing 4.2 shows the code used to load and test the PMML file produced (available in the cPMML github repository).

[3] In Python, *Dictionaries* are structures of $name \rightarrow value$ pairs

```cpp
#include "cPMML.h"
#include <iostream>
#include <unordered_map>

int main(){

  cpmml::Model model("./DecisionTreeIris.pmml");
  std::unordered_map<std::string, std::string> sample = {
    {"petal.length","4.6"},
    {"petal.width","1.3"},
    {"sepal.length","6.6"},
    {"sepal.width","2.9"}
  };

  std::cout << "Prediction: " << model.predict(sample) << std::endl;
  return 0;
}
```

Listing 4.2: Simple algorithm to load and use a *Decision Tree* pmml file (C++)

First, the PMML file is loaded by initializing a *cPMML Model* variable named "model". Then, the *unordered_map* structure from the standard library was used to implement the same sample used in *Scikit-Learn* and to save it in the variable named "sample" . Finally, the *predict* method of the *cPMML Model* structure was utilized to predict the classification of the sample provided.

Python to C++ Results

Even though the pipeline was successfully created in the Python program and *cPMML* successfully loaded the *pmml* description, the *predict* method didn't work. So, it is not possible to extract clear results regarding this solution and the ML tools involved.

4.2 m2cgen

m2cgen is a tool mainly developed in *Python* that provides a representation of the ML trained model in a different language than the one used to train the model in the first place. It supports a wide range of programming languages including *Python*, *C* and *C#*. Amongst the ML models of interest this library only supports the *Scikit-Learn* OCSVM, but it also supports the *Random Forest* and *Decision Tree* models. The figure 4.2 illustrates the process to convert a *Scikit-Learn* model to *C*.

Figure 4.2: Transpilation process from *Python* to *C++*

4.2.1 OCSVM Model

To discover the viability of the *m2cgen* tool, a comparison was required to asses if the transpiled model would match the original's output. The listing 4.3 demonstrates the code used to train and use a simple OCSVM model, as well to transpile the code to *C* and to save an analysis graph.

```
1  from sklearn.svm import OneClassSVM
2  from numpy import where
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from sklearn.datasets import load_wine
6  import m2cgen as m2c
7
8  X1 = load_wine()["data"][:, [1, 2]]
9
10 xx1, yy1 = np.meshgrid(np.linspace(0,6, 178), np.linspace(1.25,3.30,178)
       )
11
12 plt.figure('OneClassSVM Example')
13 plt.subplot(121)
14 plt.title('Normal Data')
15
16 plt.scatter(X1[:,0],X1[:,1], color="black")
17
18 svm = OneClassSVM(kernel='rbf', gamma=0.05, nu=0.1)
19 svm.fit(X1)
20
21 pred = svm.predict(X1)
22
23 anom_index = where(pred==-1)
24 values = X1[anom_index]
25 print("Outlier points")
26 print(values)
27
28 plt.subplot(122)
29 plt.title('Analyzed Data')
30 plt.scatter(X1[:,0], X1[:,1], color="black")
31 plt.scatter(values[:,0], values[:,1], color='red')
32
33 Z1 = svm.decision_function(np.c_[xx1.ravel(), yy1.ravel()])
34
35 Z1 = Z1.reshape(xx1.shape)
36
37 plt.contour(xx1,yy1,Z1,levels=0,colors='blue')
38 plt.savefig("One Class SVM kernel='rbf' gamma=0.05 nu=0.1.png")
39 plt.show()
40
41 code = m2c.export_to_c(svm)
42 print(code)
```

Listing 4.3: Train and output of Scikit-Learn OCSVM model and transpilation to C (Python)

First, the data to train the OCSVM model is defined (line 8). The *X1* variable will save the data of the columns with indexes one and two from the *Scikit-Learn* wine dataset and was only used for testing purposes. To define the OCSVM model parameters *Scikit-Learn* provides the *OneClassSVM* function, but the main objective is to save the outlier points to be further compared with the *C* transpiled code's outliers. The *fit* method is then used to train the model with the *X1* (line 19). *Scikit-Learn* library provides a *predict* method to test ML models and it was used with the same dataset used for training (line 21). As OCSVM is a classification model the output will be -1 for outliers and 0 for the rest.

The figure 4.3 displays the result of this test, generated with Pyplot collection from Matplotlib library, where the red points represent the inputs classified as outliers. For extra

analysis, the hyperplane (in blue) separating the inliers from outliers is calculated using the decision_function and drawn with the *contour* method.

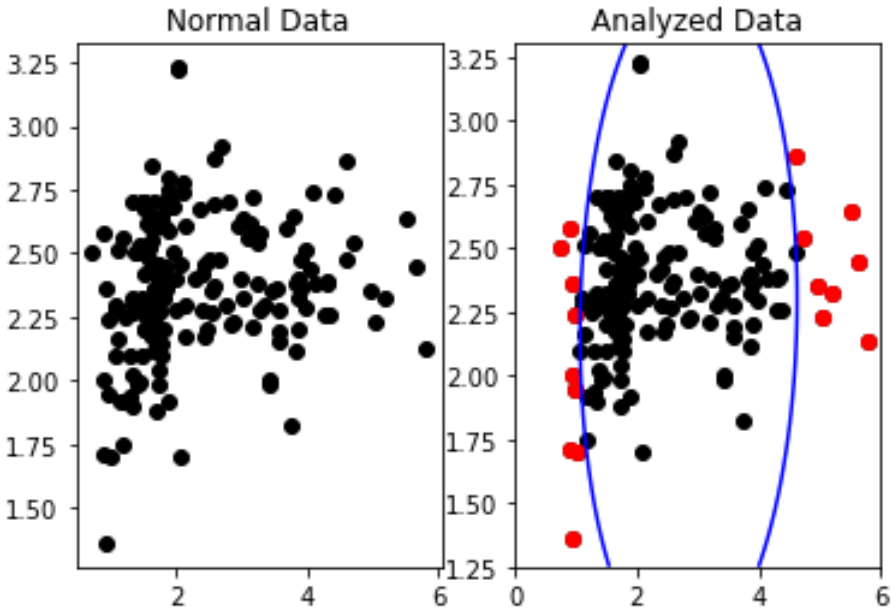


Figure 4.3: Scikit-Learn OCSVM model classification output

4.2.2 OCSVM transpiled model

After printing the generated graph, the OCSVM trained model is transpiled to *C* programming language by the use of the *export_to_c* function from m2cgen library. This function's only argument is the ML trained model which is saved in the *svm* variable and it's output is saved in a variable named *code*. This transpiled code is then printed into the terminal and copied to the *C* program visible in listing 4.4.

```c
#include <math.h>
#include <stdio.h>
#define N_samples 181

double score(double input[]) {
  return -12.920388853549944 + exp(-0.05 * (pow(0.94 - input[0], 2.0) +
    ... + pow(2.45 - input[1], 2.0))) * 1.0;
}

int main()
{

  double samples[N_samples][2]={{1.71,2.43},
              {1.78,2.14},
              .
              .
              .
              .
              {4.1,2.74}};

  double result[N_samples];
  for(int i=0;i<N_samples;i++)
  {

    result[i] = score(samples[i]);
  }
  printf("Outlier points:\n");
  for(int j=0;j<N_samples;j++)
  {
    if(result[j] < 0)
    {
      printf("[%f %f]\n",samples[j][0],samples[j][1]);
    }
  }


  return 0;
}
```

Listing 4.4: Output test of the transpiled model (C)

The *score* function (lines 5-7) contains a snippet of the transpiled code but many of the arguments have been suppressed for visual reasons. In the *main* function the *samples* array is defined and contains the dataset used to train and test the Scikit-Learn model. Then, the *result* array is created to hold the output of the samples and a cycle performed to calculate and save the sample's predictions. Another cycle is created afterwards to print into the terminal the samples classified as outliers to be compared to the ones identified by the original model.

The following table displays the values of the features belonging to the samples classified as outliers by the Scikit-Learn model and the transpiled version of *m2cgen*.

The Table 4.1 shows that between all the 180 samples used, only one was incorrectly classified as an outlier by the *m2cgen* transpiled model (marked as red). It is also notable that all the outliers identified by the *Scikit-Learn* were correctly identified by the transpiled model as well. This could mean that the *m2cgen* model only has an increase in the false positives rate but this conclusion requires further testing.

Table 4.1: Inputs classified as outliers by the m2cgen transpiled code and the original Scikit-Learn model

m2cgen	Scikit-Learn
3.85 3.49	
0.94 1.36	
1.01 1.70	
0.94 2.36	
0.90 1.71	
0.99 1.95	
0.92 2.00	
0.89 2.58	
0.98 2.24	
0.74 2.50	
5.80 2.13	
4.72 2.54	
5.51 2.64	
4.95 2.35	
5.04 2.23	
5.19 2.32	
4.60 2.86	
5.65 2.45	

m2cgen impressions

Despite the interesting results, the *m2cgen* didn't provide a perfect match with the *Scikit-Learn* ML model output and the lack of documentation leaves an uncertainty associated with the transpilation process.

4.3 ONNX

ONNX is a wide adopted ML model format and can be used to convert models between different technologies. As the models were trained with *Scikit-Learn*, first they need to be converted into a ONNX model through skl2onnx library. This library will translate the Scikit-Learn operators into ONNX operators to guarantee compatibility.

```
import pickle
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType

loaded_model = pickle.load(open(Input_Model_Path, 'rb'))

initial_type = [('float_input', FloatTensorType([None, Number_features]))]

ONNX_model = convert_sklearn(loaded_model, initial_types=initial_type, target_opset=required_opset)

with open(Output_Model_Path, "wb") as f:
    f.write(ONNX_model.SerializeToString())
```

Listing 4.5: ML model conversion from Scikit-Learn to ONNX algorithm (Python)

First, the ML models are loaded (line 5). After loading the models, it is required to define the type of inputs the model is expecting. This definition needs to be a python list of tuples, where each tuple is composed of a *variable name* and a *data type* defined in *data_types.py* [Official Documentation]. Since the models were trained with only one type of data, Floating Point (Float)[4] numbers, the python list has only one tuple. *float_input* is the name of the tuple and *FloatTensorType* is the Scikit-Learn data type function to represent Float numbers. This function will return the *numpy*[5] data type equivalent and the data structure. In Figure 4.5, *FloatTensorType* dimensions "[None, Number_features]" represents an array with *Number_features* input values of type *numpy* Float with 32 bit[6] representation. The definition of the model's expected data type and structure is essential for *convert_sklearn* function, which role is to convert Scikit-Learn models to ONNX. This function only requires two parameters: The variable containing the loaded model and the data type and structure. It is also advisable to define the *target_opset* parameter, since it's omission will default to the latest *opset* version potentially creating compatibility issues. Once the Scikit-Learn model has been successfully converted to ONNX, it needs to be serialized into one contiguous memory buffer with *SerializeToString* function, a method available to all *ONNX* objects.

4.3.1 Tensorflow Lite

TensorFlow Lite is a ML tool developed for low resource systems. It requires less memory, reducing the technology footprint, and provides a faster inference process by allowing the models to access data without any prior parsing processes. The process to convert the ONNX model into an TensorFlow Lite model is represented in the figure 4.4.

Figure 4.4: ML Model conversion process from Scikit-Learn to TensorFlow

After possessing an ONNX model representation of the original ML model, it can then be further translated to TensorFlow with onnx-tf, the tool developed exclusively for converting ONNX models into a TensorFlow representation. The Figure 4.6 corresponds to the code used to convert the model.

```
1 import onnx
2 from tensorflow.python.tools.import_pb_to_tensorboard import
      import_to_tensorboard
3 from onnx_tf.backend import prepare
4
5 onnx_model = onnx.load("Model.onnx")
6
7 tf_rep = prepare(onnx_model)
8
9 tf_rep.export_graph("tf_model.pb")
```

Listing 4.6: ML model conversion from ONNX to Tensorflow algorithm (Python)

[4] Float numbers are a common way of decimal number representation in the programming community Float history

[5] *Numpy* is a Python package targeting scientific computation known for it's multidimensional array object operations

[6] Bit is the most basic information representation in digital systems

First, the model is prepared to be converted with the *prepare* function and then the model's backend representation can be converted to a *TensorFlow Protobuf* file[7] with *export_graph*. *Protobuf* (Protocol Buffers) is a file format used by TensorFlow to save and load ML models.

Results

The convertion process of the ML models from ONNX to Tensorflow were performed using several Tensorflow versions. Due to hardware limitations, the Tensorflow tool couldn't be installed with Pip Installs Package (pip)[8] and required the tool's code be locally compiled. This process revealed to be very time consuming and *Tensorflow Lite* implementation abandoned.

4.4 ONNXRuntime

To test if *ONNXRuntime* would provide the same output as the original *Scikit-Learn* model, a comparison similar to the one performed for *m2cgen* was executed. The listing 4.7 displays the code used to make this assessment.

[7] ONNX-Tensorflow API describing the *prepare* and *export_graph* functions
[8] *Pip* is a package manager in Python

```python
import numpy as np
from sklearn.datasets import load_wine
from sklearn.svm import OneClassSVM
from skl2onnx import convert_sklearn
from skl2onnx.common.data_types import FloatTensorType
import onnxruntime as rt

X1 = load_wine()["data"][:,[1,2]]

svm = OneClassSVM(kernel='rbf', gamma = 0.05, nu = 0.1)
svm.fit(X1)

pred = svm.predict(X1)

anom_index = np.where(pred==-1)

Number_samples = len(X1)
X1_float = []
for j in range(Number_samples):
    temp = []
    for i in range(2):
        temp.append(float(X1[j][i]))
    X1_float.append(temp)

initial_type = [('float_input', FloatTensorType([None, 2]))]
onx = convert_sklearn(svm, initial_types=initial_type,target_opset=17)
with open("OCSVM.onnx", "wb") as f:
    f.write(onx.SerializeToString())

sess = rt.InferenceSession("OCSVM.onnx")
input_name = sess.get_inputs()[0].name
label_name = sess.get_outputs()[0].name

ONNX_index = []
for i in range(Number_samples):
    pred_onx = sess.run([label_name], {input_name: [X1_float[i]]})[0]
    if pred_onx == -1:
        ONNX_index.append(i)

Number_outliers_onnx = len(ONNX_index)
if len(anom_index[0]) == Number_outliers_onnx:
    for i in range(Number_outliers_onnx):
        if anom_index[0][i] == ONNX_index[i]:
            if i == Number_outliers_onnx-1:
                print("All outliers match")
            else:
                continue
        else:
            print("Value " + str(X1_float[ONNX_index[i]]) + "doesn't
    match")
            break
else:
    print("Number of outliers don't match")
```

Listing 4.7: Comparison of the Scikit-Learn and ONNXRuntime outputs (Python)

First, the data to be used in the model training is defined. It is the same as the one used in the *m2cgen* test (columns with indexes 1 and 2 of the *Scikit-Learn* wine dataset) and

is saved in the *X1* variable (line 8). The ML model parameters are then defined with the *OneClassSVM* function from Scikit-Learn (line 10) and the OCSVM model trained with the *X1* dataset through the use of the *fit* method (line 11). The output classifications are calculated with the *predict* function and all the samples classified as outliers by the *Scikit-Learn* original model arestored in "anom_index".

To be able to use the same *X1* dataset, every value must be converted to float. The "X1_float" array is initiated and a cycle created to store into it the *X1* float values. Then, the process used in listing 4.5 to convert the ML model into the ONNX format is replicated. After having the *ONNX* model a session is created by the use of the *InferenceSession* function. This function main parameter is the model itself. The inputs and outputs metadata are obtained with the *get_inputs()* and *get_outputs()* functions, respectively. This data is crucial for the prediction process as they are essential parameters in the function that provides such feature. A cycle is then created to predict each sample and store the indexes of the ones classified as outliers by the *ONNX* model. The *ONNXRuntime* method to make predictions is the *run* method. This method returns 1 for inliers and -1 for outliers.

After obtaining the classification of the *ONNX* model, a comparison is performed against the *Scikit-Learn* results to assess if they match. To make sure the outliers identified by each model are the same, first the number of outliers must be the same and then, each stored index in any *P* position in one of the arrays must be the same as the index stored in the same position *P* of the other array.

Results

The code in listing 4.7 was used with all the models of interest supported by *ONNXRuntime* so, OCSVM, Stochastic Gradient Descent One Class Support Vector Machine (SGDOCSVM), Isolation Forest (ISO) and Local Outlier Factor (LOF). In all the tests, the output was "All outliers match" meaning there was a perfect match on the outliers identified by the original *Scikit-Learn* model and the converted *ONNX* version.

4.5 Choice of tool

The use of the PMML ML model description through *cPMML* library didn't went through. *cPMML* successfully loaded the model's PMML description but was unable to use it to make predictions. *m2cgen* provided interesting results as only one sample out of the whole dataset was misclassified. This tool's transpilation process wasn't clear leaving an uncertainty associated with future results. *TensorflowLite* solution wasn't achieved has the convertion process of the ONNX ML models to Proto Buffers wasn't possible. *ONNXRuntime* supports almost all the models of interest and provided a perfect match on the predictions made. From the solutions explored, *ONNXRuntime* was one of two successfully implemented (alongside *m2cgen*). It has a wider range of supported models, is implemented in a low level programming language and the test performed revealed a perfect match between it's outliers and *Scikit-Learn* outliers for the same dataset and models. As so, *ONNXRuntime* is the tool chosen to be further explored regarding it's prediction times.

Chapter 5

ONNXRuntime Time Analysis

After choosing the *ONNXRuntime* as the best tool to use in the implementation of ML models in constrained devices, a more robust code was developed to assess the time distribution of the *ONNXRuntime* models to execute the predictions. This chapter describes the code flow and shows a time comparison of the results between different devices, libraries (*Scikit-Learn* and *ONNXRuntime*), and models.

5.1 Software Developed for Response Time Analysis

This implementation has the structure displayed in Figure 5.1. The file's structure shown bellow was designed to ease the implementation understanding and, therefore, it's manipulation. This way, it is very straightforward to add new datasets, models and implementations using other libraries.

Figure 5.1: File Hierarchy's

- The "Initiate.py" file is responsible for understanding the user's parameters through the command line. These parameters include the implementations to use, the possibility to perform the time analysis and how many times to predict the same input, and the possibility to create the time graphs;
- The "Dataset" folder holds all the datasets to be chosen from to use as the ML model's input;
- The "Models" folder contains all the available models;
- The "Datasets.csv" defines the datasets to use by the program and is structured as a list of the full paths to the files of interest. Each line has the full path to one of the datasets to be used followed by a comma (",") , and a 0 or a 1 depending on the nature of the dataset. For malicious datasets the comma must be followed by a 1 and for regular datasets by a 0. This classification is crucial to further calculate the ML models performance like the *False Positive* rate;
- The "Models.csv" contains only the full paths to the ML models to use;

- The "Implementations" folder has three different implementations: *Scikit-Learn*, *ONNXRuntime* and *ONNXRuntime Optimized*. The "Implementations" folder also contains two additional folders, *src* and *Auxiliar*, containing support functions to the main code to ease the main code's understanding. The *src* folder contains functions developed by Nuno Schumacer when integrated with the MIRAI project;
- The "Reproduce_Data" folder contains the data and configurations of the last time analysis performed so the graphs can be rebuilt.

5.2 Software Operation

All the implementations have a similar flow of operations and the *ONNXRuntime Optimized* is the most complex so, the following explanation is based on it.

Any of the implementations starts to remove the non interesting features from the datasets. Their values are normalized and converted to *float* as required by *ONNXRuntime*. The normalization process is recommended when the features have values in very different scales. In ML models like OCSVM where the model attempts to calculate a hyperplane to separate the inliers from the outliers, if the normalization doesn't occur, the results can be heavily affected.

After having the dataset ready to be used, a cycle is created to use all the required models. After the first use of the program, new converted models, *ONNX* and *ONNXOptimized* models, are created. So, the cycle starts to verify which model it has and performs the necessary conversions until it possesses the *ONNX optimized* model.

A timer is then initiated, and an inference session opened with the *ONNX optimized* model. The model's output for each flow is calculated and compared with its real classification. The timer is stopped right after this statistics process and the time required to perform this sequence of tasks stored.

5.3 Results

The objective of this test is the assessment of time required to predict the flows classification. Although the output of every ML tool and model used are not directly compared, the statistics metrics calculated based on their classifications are exactly the same meaning the outputs are the same too. This software was tested with the HP Computer, a Mini PC, a Raspberry Pi 3B+ and the ASUS laptop. The following Table displays the major components of the target and used devices.

Table 5.1: Main characteristics of the devices used

Devices \Specifications	Operating System	Architecture	CPU op-model	Number of CPUs
NOS Router		32-bit	Intel(R) Atom(TM) CPU CE2752	2
HP Computer	Ubuntu 20.04.6 LTS	64-bit	Intel(R) Core(TM)2 Quad CPU Q8400	4
Mini PC	Ubuntu 20.04.6 LTS	64-bit	Intel® Celeron® J4125	4
Raspberry Pi 3B+	Debian GNU/Linux 11 (bullseye)	64-bit	Cortex-A53 (ARMv8)	4
ASUS laptop	Ubuntu 20.04.6 LTS	64-bit	Intel(R) Core(TM) i7-6500U	4

The script allows the definition of how many times to predict the same input. The code works with the average time required for each flow to compare the libraries implemented as well as the models.

The following tables display the average time, in milliseconds, required by each of the devices to execute the prediction process for every single ML model and tool. Table 5.2 presents the time results of the *Scikit-Learn*, Table 5.3 of the *ONNXRuntime* and Table 5.4 of the *ONNXRuntime Optimized* models.

Table 5.2: Time performance of Scikit-Learn

Model \Device	HP Computer	Mini PC	ASUS laptop
ISO	64.93	58.59	35.01
LOF	11.39	6.25	3.68
OCSVM	3.47	2.89	1.73
SGD OCSVM	2.65	2.38	1.45

Table 5.3: Time performance of ONNXRuntime

Model \Device	HP Computer	Mini PC	Raspberry Pi 3B+	ASUS laptop
ISO	3.93	3.53	7.13	1.41
LOF	32.51	24.02	44.75	8.91
OCSVM	0.23	0.20	0.34	0.12
SGD OCSVM	0.05	0.06	0.07	0.03

Table 5.4: Time performance of ONNXRuntime Optimizer

Model \Device	HP Computer	Mini PC	Raspberry Pi 3B+	ASUS laptop
ISO	3.90	2.89	7.09	1.45
LOF	30.48	22.76	43.04	8.57
OCSVM	0.23	0.21	0.34	0.14
SGD OCSVM	0.005	0.006	0.07	0.02

Tools time performance

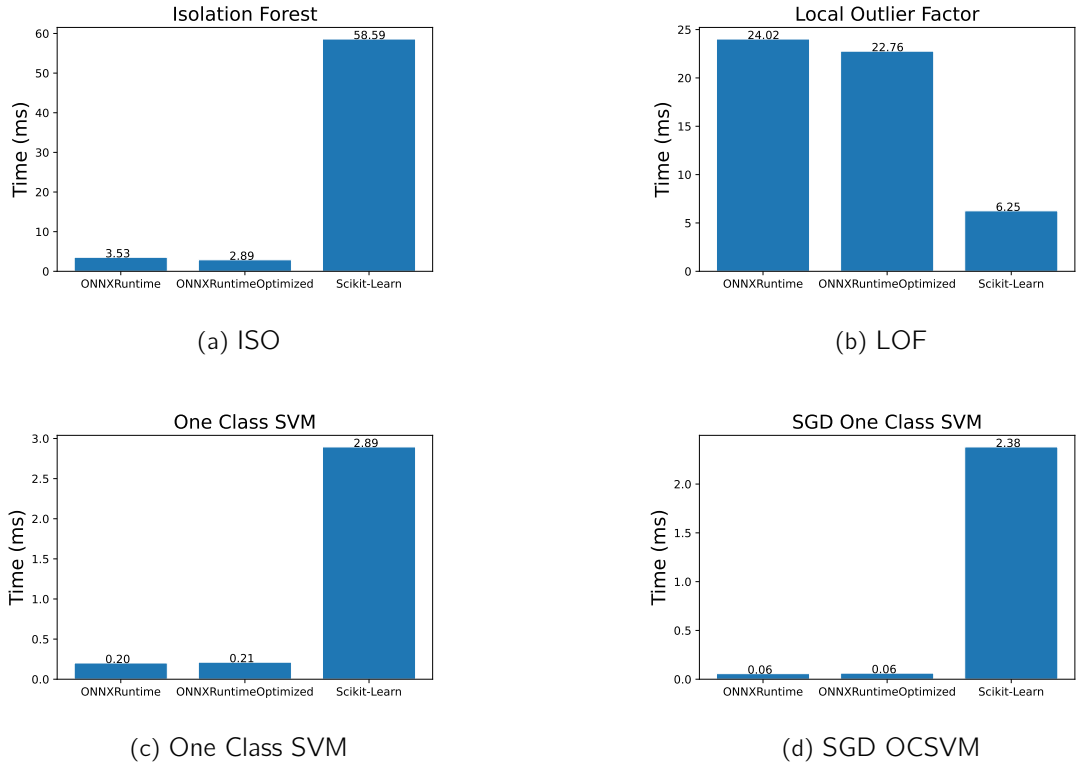
The figure 5.2 represents the the time required by each of the libraries to run the four selected models in the Mini PC. Each column corresponds to the average time required per input. It is clear that besides the LOF model, *ONNXRuntime* takes less time than *Scikit-Learn* to perform the same operation. *Scikit-Learn* takes 17 times longer compared to *ONNXRuntime* when running the ISO ML model, 14 for the OCSVM and 40 for the SGDOCSVM. The results on the other platforms are similar so they are not presented.

Performance on the Various Platforms

Figure 5.3 shows the time required by each device to run each of the ML models in the *ONNXRuntime*. It is noticeable that in the four different devices used, Raspberry Pi was the slowest. The graphs also show that there's a relationship between the device's resources and the time required to perform the prediction process. For the ISO, OCSVM and LOF models, the Raspberry Pi took 1.81, 1.48 and 1.38 times longer than the HP computer. Except for the SGDOCSVM model, the relationship between the different devices and the average time required to execute the predictions is similar.

Models Time Performance

The Figure 5.4 represents the time distribution of the different models in the three different implementations with the Mini PC. Regardless of the library used, SGDOCSVM and OCSVM are the fastest models. The ISO is the model that takes the most time per prediction in *Scikit-Learn* and LOF in *ONNXRuntime*.



(a) ISO

(b) LOF

(c) One Class SVM

(d) SGD OCSVM

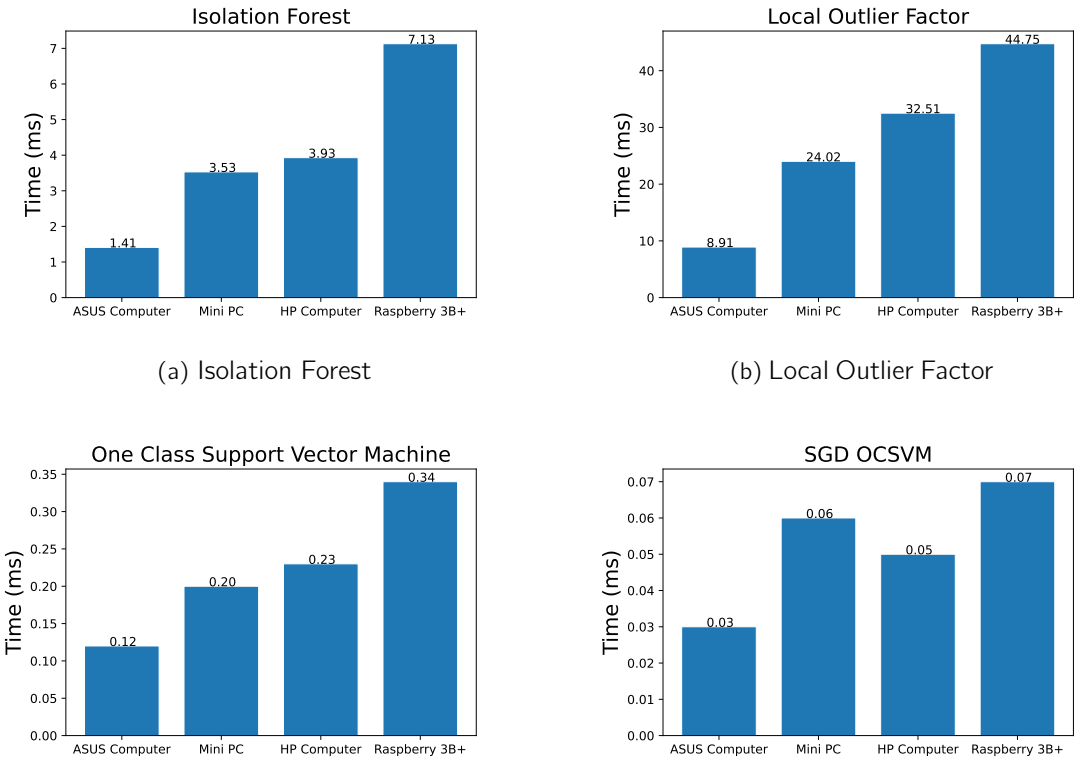
Figure 5.2: Libraries time efficiency for each model

5.3.1 Discussion

The time required for the prediction process of ML technology depends on the ML model, the ML prediction tool and the device's characteristics. The comparison of the *ONNXRuntime* and *Scikit-Learn* times (Figure 5.2) show that for all the used models except the LOF, *ONNXRuntime* takes substantially less time to execute the same tasks. This fact can be specifically related to the *ONNXRuntime* implementation of the LOF model, the tasks parallelization process in the thread pool, but an investigation on this event reasons revealed inconclusive. The *ONNXRuntime* optimizer proved to be a little faster than the non optimized ones in all the ML models except for the SGDOCSVM where the time required was the same. This last model behaviour can be justified if there are no optimizations available or the time difference is just not that significant.

The components (processor, available memory and more) of the four devices show their impact in the execution of the prediction task. All the graphs in Figure 5.3 show a similar order in the device's required time to run the inference process in all the used models but OCSVM. While in the rest of the ML models the HP computer took more time than the Mini PC, in the OCSVM it was faster. Although the difference is approximately 0.01 milliseconds (maybe irrelevant), it might be due to hardware specific details or system load related.

Through the analysis of the time distribution of the different tools and ML models in the Mini PC (Figure 5.4), it is possible to assess if the time required to predict each sample is constant or if it fluctuates a lot around the media. From the three tools implemented, *Scikit-Learn* is the most time consistent as only one sample used in the LOF model took considerably more time than the others. As *ONNXRuntime* is a real time environment and the resources are being allocated as they are need, a spontaneous delay can be the reason



(a) Isolation Forest

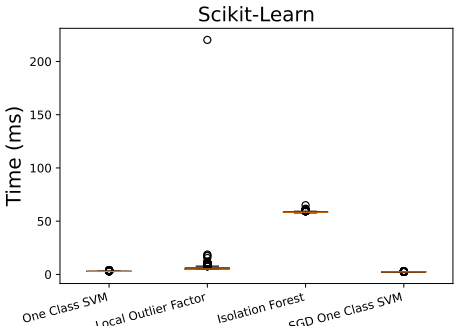
(b) Local Outlier Factor

(c) One Class SVM

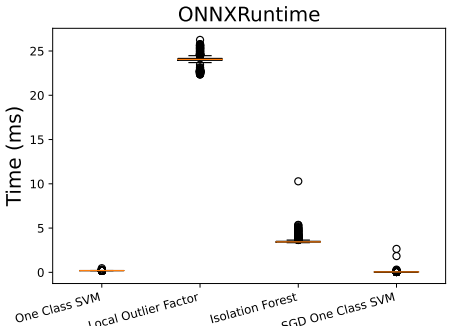
(d) SGD OCSVM

Figure 5.3: Devices time efficiency for each ML model

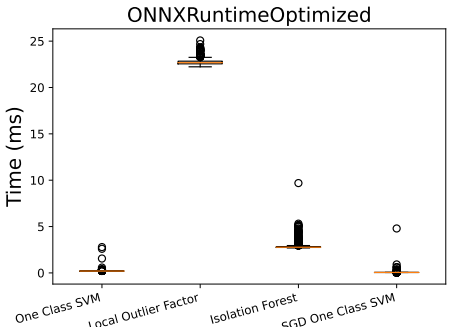
behind this fact. Nevertheless, for the amount of samples used (+500), *ONNXRuntime* also demonstrates a low variance in the samples prediction's time, meaning that any prediction will most likely take the average time to execute it.



(a) Scikit-Learn



(b) ONNXRuntime



(c) ONNXRuntimeOptimized

Figure 5.4: Time performance of the inference process

Chapter 6

Discussion and Conclusions

This section resumes the thesis work. It starts by mentioning this dissertation purpose and its contribution within the MIRAI project. It continues with the main results and conclusions of the thesis and the announcement of a paper developed around it. It ends with a description of some future works related to this thesis connected topics.

6.1 Conclusions

This thesis aimed at deploying ML technology in resource-scarce systems for cybersecurity purposes. To achieve it, an investigation on ML solutions was made to assess their potential regarding results and time performance. Then, a plan based on the tool's potential was designed to guide the tools exploration phase. After achieving very good results with one of the tools, a software was developed to test the chosen tool's time required to perform the prediction process. The software was then introduced in all the available devices except one, (NOS router), to have an idea about the time required to perform this tasks on more capable devices. The integration of the software with the NOS router didn't went through and is still in investigation.

The results show that the proposed tool, *ONNXRuntime*, is able to use the *Scikit-Learn* ML trained models with a perfect match in samples classification and executes all the predictions much faster in all the models except for LOF. The prediction for the LOF model proved to be 3.84 times slower, but 14 times faster for the OCSVM model, 17 for the ISO and 40 for the SGDOCSVM. The results also show that *ONNXRuntime Optimizer* function provides a slight improvement in the time performance of the prediction process for almost all the models and devices with a few exceptions.

6.2 Future Work

There is still a lot of improvements and investigations required to achieve the best possible result in the topic of implementing ML technology in devices with limited resources. This thesis only looked at open-source and free ML tools but, there are still a lot of paying tools that can achieve better results. The following list describes some of improvements or related investigations that can be done to understand even better the ML technology and their use in both restricted and non restricted devices:

- Explore native low level solutions - The tools mentioned in Section 3.1 developed in low level programming languages weren't explored in the tools exploration phase. Despite the fact that none of them supports a wide range of ML models, the ones they support

could be tested and their time results even compared with *ONNXRuntime*'s. These libraries can prove to be even faster and/or lighter than *ONNXRuntime*;

- Implement a Real Time system using live network activity rather than pre-processed datasets (Currently under development) - The software described in Chapter 5 works in static conditions as the datasets captured from the network are already stored. A live implementation of the system requires TSTAT to update the log files in real-time (only available in the version under development) and an adaptation of the software created to comply with all the systems new requirements;
- Investigate the features that achieve the best ML model accuracy - There is still uncertainty associated with the best features to use in the model's training and TSTAT live logs reported features differ from the ones used so far in the project. So, a deep research to evaluate the impact of the available features on the models accuracy is vital.

Bibliography

Abdul-Ghani, Hezam Akram, Dimitri Konstantas, and Mohammed Mahyoub (2018). *A Comprehensive IoT Attacks Survey based on a Building-blocked Reference Model*. url: www.ijacsa.thesai.org.

Bahri, Ahed (Sept. 2020). *Difference between AI ML DL*. English. Web page. Consulted at 20th December 2022. Medium. url: https://ahedbahri.medium.com/difference-between-ai-ml-dl-4e50ab243f04.

Baker, Kurt (Sept. 2021). *THE 14 MOST COMMON CYBER ATTACKS*. English. Web page. Consulted at 11th January 2023. 150 Mathilda Place, Ste. 300 Sunnyvale, CA 94086: crowdstrike. url: https://www.crowdstrike.com/cybersecurity-101/cyberattacks/most-common-cyberattacks/.

Cifredo-Chacón, María Ángeles et al. (2019). "Implementation of processing functions for autonomous power quality measurement equipment: A performance evaluation of CPU and FPGA-based embedded system". In: *Energies* 12 (5). issn: 19961073. doi: 10.3390/en12050914.

Department, Statista Research (June 2022). *Annual number of malware attacks worldwide from 2015 to first half 2022*. English. Web page. Consulted at 12th January 2023. 209-215 Blackfriars Road, 5th Floor - SE1 8NL London - United Kingdom: statista. url: https://www.statista.com/statistics/873097/malware-attacks-per-year-worldwide/.

Diskin, Dani (2020). *Machine Learning Basics*. English. Web page. Consulted at 15th December 2022. Rua Prof. Atílio Innocenti 165 (Itaim Bibi), São Paulo, SP: Pinterest. url: https://br.pinterest.com/pin/596656650606774777/.

Eskandari, Mojtaba et al. (Aug. 2020). "Passban IDS: An Intelligent Anomaly-Based Intrusion Detection System for IoT Edge Devices". In: *IEEE Internet of Things Journal* 7 (8), pp. 6882–6897. issn: 23274662. doi: 10.1109/JIOT.2020.2970501.

Greenberg, Sarah (July 2021). *Birth And Evolution Of Internet of Things (IoT)*. English. Web page. Consulted at 9th January 2023. 49 Walnut Park, Building 2 Wellesley, MA 02481, USA: BCC Research. url: https://blog.bccresearch.com/birth-and-evolution-of-internet-of-things-iot.

Guazzelli, Alex et al. (2009). "PMML: An open standard for sharing models." In: *R J.* 1.1, p. 60.

Howarth, Josh (Dec. 2022). *The Ultimate List of Cyber Attack Stats (2023)*. English. Web page. Consulted at 9th January 2023. 548 Market St. Suite 95149 San Francisco, California: EXPLODING TOPICS. url: https://explodingtopics.com/blog/cybersecurity-stats.

Igarashi, Mitsuhiko et al. (2014). "Assessment of reliability impact on GHz processors with moderate overdrive". In: *Fifteenth International Symposium on Quality Electronic Design*, pp. 456–460. doi: 10.1109/ISQED.2014.6783359.

Jay, Allan (Nov. 2022). *Number of Internet of Things (IoT) Connected Devices Worldwide 2022/2023: Breakdowns, Growth & Predictions*. English. Web page. Consulted at 9th January 2023. EU Office: Grojecka 70/13 Warsaw, 02-359 Poland: FinancesOnline. url:

https://financesonline.com/number-of-internet-of-things-connected-devices/.

Kaspersky (Nov. 2022). *DDoS attacks in Q3 2022*. English. Web page. Consulted at 9th January 2023. url: https://securelist.com/ddos-report-q3-2022/107860/.

Lukehart, Mark (Jan. 2022). *2022 Cyber Attack Statistics, Data, and Trends*. English. Web page. Consulted at 12th January 2023. One Sansome Street - Suite 3500 - San Francisco, CA 94104: Parachute. url: https://parachute.cloud/2022-cyber-attack-statistics-data-and-trends/.

Magnani, Gabriele et al. (Mar. 2021). "The impact of precision tuning on embedded systems performance: A case study on field-oriented control". In: vol. 88. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing. isbn: 9783959771818. doi: 10.4230/OASIcs.PARMA-DITAM.2021.3.

Marwedel, Peter (2021). *Embedded System Design*. English. Ed. by Nikil D. Dutt, Grant Martin, and Peter Marwedel. Fourth. Embedded Systems. Springer Nature. url: https://library.oapen.org/handle/20.500.12657/46817.

News, Cyware Alerts - Hacker (Jan. 2023). *Global Cyberattack Statistics 2022: Check Point Report*. English. Web page. Consulted at 12th January 2023. 111 Town Square Place - Suite 1203, #4 - Jersey City, NJ 07310: cyware. url: https://cyware.com/news/global-cyberattack-statistics-2022-check-point-report-40e1742e/.

Palandrani, Pedro and Alec Lucas (Jan. 2022). *Rising Cybersecurity Threats Expected to Continue in 2022*. English. Web page. Consulted at 12th January 2023. 605 Third Avenue, 43rd Floor - New York, NY 10158: GLOBALX. url: https://www.globalxetfs.com/rising-cybersecurity-threats-expected-to-continue-in-2022/.

Rouse, Margaret (June 2020). *Runtime Environment*. English. Web page. Consulted at 20th September 2023. Tower Financial Centre, 12th Floor, 50th Street Corner of Elvira, Panama City, Panama: Techopedia. url: https://www.techopedia.com/definition/5466/runtime-environment-rte.

Silva, Natassya B.F. et al. (Jan. 2016). "Case studies of performance evaluation of cryptographic algorithms for an embedded system and a general purpose computer". In: *Journal of Network and Computer Applications* 60, pp. 130–143. issn: 10958592. doi: 10.1016/j.jnca.2015.10.007.

Society, IEEE Signal Processing et al. (2017). *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS) : Aug. 29 2017-Sept. 1 2017.* isbn: 9781538629390.

Sovrin (Aug. 2022). *Internet Of Things (IoT)*. English. Web page. Consulted at 19th December 2022. 86 N University Ave, Suite 110, Provo, UT 84601: Sovrin Foundation. url: https://sovrin.org/library-iot/.

Szakál, Anikó et al. (2017). *SISY 2017 : IEEE 15th International Symposium on Intelligent Systems and Informatics : proceedings : September 14-16, 2017, Subotica, Serbia*. isbn: 9781538638552.

Tietsort, J.R. (Dec. 2022). *17 Types of Cyber Attacks Commonly Used By Hackers*. English. Web page. Consulted at 12th January 2023. Aura. url: https://www.aura.com/learn/types-of-cyber-attacks.

Viegas, Eduardo, Altair Santin, et al. (Sept. 2018). "A reliable and energy-efficient classifier combination scheme for intrusion detection in embedded systems". In: *Computers and Security* 78, pp. 16–32. issn: 01674048. doi: 10.1016/j.cose.2018.05.014.

Viegas, Eduardo, Altair Olivo Santin, et al. (Jan. 2017a). "Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems". In: *IEEE Transactions on Computers* 66 (1), pp. 163–177. issn: 00189340. doi: 10.1109/TC.2016.2560839.

– (Jan. 2017b). "Towards an energy-efficient anomaly-based intrusion detection engine for embedded systems". In: *IEEE Transactions on Computers* 66 (1), pp. 163–177. issn: 00189340. doi: 10.1109/TC.2016.2560839.

Vinayakumar, R. et al. (2019). "Deep Learning Approach for Intelligent Intrusion Detection System". In: *IEEE Access* 7, pp. 41525–41550. issn: 21693536. doi: 10.1109/ACCESS.2019.2895334.

Wikipedia (Jan. 2023a). *Pipeline (software)*. English. Web page. Consulted at 29th June 2023. 1 Montgomery Street, Suite 1600, San Francisco, California: Wikipedia. url: https://en.wikipedia.org/wiki/Pipeline_%5C%28software%5C%29.

– (June 2023b). *Traffic flow (computer networking)*. English. Web page. Consulted at 15th July 2023. 1 Montgomery Street, Suite 1600, San Francisco, California: Wikipedia. url: https://en.wikipedia.org/wiki/Traffic_flow_(computer_networking).