An Explainable Machine Learning Methodology For Detecting Novel Network Attacks

Final Project Report

Author: Caroline Smith

Supervisor: Fabio Pierazzi

Student ID: 21068512

April 12, 2024

Abstract

Network attacks are a persistent issue worldwide, and as attackers and their intrusion techniques continue to evolve, security professionals must constantly adapt to new attacks while reviewing a huge amount of network traffic. Machine learning-based detection systems can automate the process of flagging these attacks, allowing security professionals to spend less time detecting network attacks and more time preventing and mitigating them. However, as most high-performance machine learning models for detecting network attacks are black boxes, it is often unknown how a model classifies a particular traffic flow as malicious or benign. Explanations for a model's prediction of a particular traffic flow, i.e. local explanations, can improve the usefulness of an intrusion detection model since they can supply clear reasoning for the prediction and a 'next step' for the security professional. Explainable models could also provide insight on the intrusion vector and techniques of a novel attack approach. This research proposes PABLO, a state-of-the-art methodology which utilises an efficient, game theoretic approach to detect and explain network attacks, and investigates an implementation of PABLO, PyPABLO, which uses a Random Forest classifier, PySpark, and the SHAP software library to detect and explain network attacks. This research will evaluate the ability of PyPABLO to detect and explain novel network attacks such as port scan, brute force, and Patator attacks. It is imperative to explore the explainability of machine learning-based network intrusion detection systems, as these systems will not be useful nor commercially viable until they can provide effective reasoning behind their classification decisions. Locally explainable learning-based detection systems are therefore crucial for defeating real-world attackers.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Caroline Smith

April 12, 2024

Acknowledgements

I would firstly like to thank Dr Fabio Pierazzi for his invaluable guidance as a lecturer and supervisor. He has made a larger impact on my future than he knows. I would also like to thank Victoria Gellner, Liam Castelli, and the SKFC for their support.

Contents

Chapter 1

Introduction

Network security has never been more important. The “Digital 2024: Global Overview Report” dataset suggests that there are approximately 5.35 billion individuals using the Internet, an increase of 1.8% from 2023 [23]. As the World Wide Web and its commercial uses become ever-more commonplace, cyber assaults by both state-backed and independent actors, to gain information from or immobilise governments, corporations and individuals, have become disturbingly common. Businesses in many sectors must contend with huge numbers of assault attempts on their networks, with China-nexus malicious actors targeting “nearly all 39 global industry sectors” in 2023 [5].

There is clear benefit to be gained from utilising machine learning to detect network intrusion attempts, including a potentially faster response time to intrusion incidents and a decrease in workload for system operators [35]. However, as explored by Sommer and Paxson in “Outside The Closed World: On Using Machine Learning for Network Intrusion Detection”, there are many fundamental obstacles to implementing successful and commercially viable machine-learning models for network intrusion detection, including high cost of errors, inability to correctly evaluate models due to a lack of high-quality data, and semantic gap [35]. In particular, research into machine learning-based anomaly detection is biassed towards judging a model on its ability to detect anomalies, and against a step which is arguably just as commercially important for an intrusion detection model– determining what the detection of said anomalies mean for an operator, and what the operator should do next in order to secure their network [35].

This research proposes PABLO (Port-scan And Brute-force Learner Of anomalies), a state-of-the-art methodology which utilises a game theoretic approach [30, 31] to detect and explain network attacks. Furthermore, this research will investigate an implementation of PABLO, PyPABLO, a Random Forest model trained on a modified version of the CICIDS2017 dataset [7] and explained by the SHAP software library [20]. In particular, PyPABLO will be evaluated on its ability to detect and explain particular 'novel' network attacks. SHAP will specifically be utilised to provide operator-friendly local explanations and data visualisations when testing PyPABLO on data containing 'novel' port scan and Patator [17] attacks, simulating the event of a novel attack technique on a network. This paper introduces the novel methodology of utilising SHAP visualisations of local explanations to explain a model's predictions on 'novel' attacks.

A few main ideas will be considered when designing a prototype classification model for anomaly-based intrusion detection. Firstly, a comparison must be drawn between the chosen classifier's 'performance' versus its ability to supply accurate and useful local explanations. Secondly, a suitable evaluation must be completed regarding the model's biases, limitations and pitfalls. In this research, the ability of the anomaly detection model to detect 'novel' attacks is also considered during the design process.

This paper highlights the difficulties of designing machine learning models that are capable of detecting novel attacks, and additionally expands upon the benefits of SHAP values for providing local explanations for network intrusion detection models via SHAP-generated plots. The objectives of this project are defined as follows:

- *Evaluate the quality* of the CICIDS2017 and 'improved' CICIDS2017 datasets by Sharafaldin, et al. and Engelen, et al [7, 32]. This will be done by exploring the limitations and pitfalls of models and their datasets, as set forth by Sommer and Paxson in "Outside The Closed World", Engelen et al. in "Troubleshooting an Intrusion Detection Dataset: the CICIDS2017 Case Study", and Arp et al. in "Dos and Don'ts of Machine Learning in Computer Security" [4, 7, 35].

- *Design an approach* for a prototype machine learning-based intrusion detection model trained on the 'improved' CICIDS2017 dataset by Engelen, et al. to specifically detect novel port scan and brute-force attacks [7].

- *Evaluate the ability* of the prototype model to detect and explain novel port scan and brute forcing attacks by utilising local explanations and other performance metrics.

- *Utilise SHAP* to provide insights on how a model makes classification decisions, and thus improve the explainability and usefulness of the model.

Relevant work regarding the intersection between SHAP and intrusion detection is relatively new [38]. The first paper to consider SHAP in the context of explainable intrusion detection was published in 2020 and analyses its capabilities to provide local and global explanations for both one-vs-all and multiclass classifiers [38]. However, Wang, et al. only evaluate the capabilities of SHAP on one dataset, NSL-KDD, which is outdated and over-used in the learning-based detection community [35, 38]. The authors Keshk, et al. propose the SPIP framework, which is partially dependent on SHAP, but focuses heavily on explainability for end users of IoT devices rather than SHAP's ability to explain novel attacks for security professionals [16]. Authors Wang et al. focus on the use of SHAP to explain the outputs of their tree-based or convolutional neural network models via the original CICIDS2017 dataset, but suffer from pitfalls such as spurious correlations and sampling bias [4, 39]. Existing work often focuses on the use of SHAP to evaluate the performance of different classifiers, explain black box models (those which do not have inherent explainability, such as deep learning-based models), or analyse local and global explanations of non-'novel' attacks [9, 14, 26, 38]. In contrast, this research will focus on the ability of a Random Forest classifier to detect malicious traffic for attacks it has not been trained on, specifically port scanning and Patator brute forcing, and additionally on how SHAP can be leveraged to explain individual classification decisions of a model for these 'novel' attacks.

The remainder of this paper is structured as follows: Chapter 2 (Background) provides relevant background data on machine learning in the intrusion detection domain and software frameworks related to local explainability. Chapter 3 (Design & Specification) outlines the experiments and design for the PyPABLO model prototype, and Chapter 4 (Implementation) delves into the concrete implementation of all experiments. Chapter 5 (Results & Evaluation) presents and discusses the experiment results and SHAP data visualisations. Chapter 6 discusses the legal, social, ethical and professional issues that could be influenced by this research. Finally, Chapter 7 presents key insights from this project and potential threads for future research.

Chapter 2

Background

As stated in Chapter 1, two main objectives of this research are to develop a new methodology for detecting ‘novel’ network attacks and to leverage the SHAP library to create understandable and useful local explanations for a cybersecurity professional. The first question to be explored in this literature review was:

1. What are the recent trends for machine learning-based intrusion detection models, and what are their pitfalls, limitations and commercial challenges?

   Once the issue of explainability and semantic gap was determined as a viable area for novel research, the second question to be explored was:

2. What are the recent trends for introducing explainability into machine learning-based network intrusion detection models, and how can explainable AI be used to mitigate pitfalls in machine learning-based intrusion detection models?

This chapter will first examine relevant factors for the perceived and actual performance of PyPABLO, a newly proposed traffic classification model, by detailing the background surrounding the current trends and pitfalls of machine learning-based intrusion detection systems. It will secondly detail the related explainability frameworks, methodologies and research that have been explored in the novel PABLO approach.

2.1 Trends and Pitfalls of Machine Learning-based Network Intrusion Detection Systems

2.1.1 What is a Network Intrusion Detection System (NIDS)?

Network security involves the protection of a network of devices from internal and external attacks, which can occur on any layer of the OSI model. In this paper, an Intrusion Detection System (IDS) monitors traffic on a network and aims to identify and flag attempted policy violations or anomalous traffic on the basis that they are potential intrusions [24]. One common type of IDS is a Network Intrusion Detection System, or NIDS. NIDSes monitor both inbound and outbound traffic across a network, and utilise two primary detection methods: signature-based and anomaly-based detection [15]. Signature-based detection (i.e. misuse detection) checks network traffic against a database of known malicious traffic patterns known as 'attack signatures' (and thus traditionally cannot detect new kinds of attacks), whereas anomaly-based detection evaluates network traffic against a constantly-changing baseline of 'normal network behaviour', often via machine learning to[15]. Currently, the unique challenges of integrating machine learning into anomaly detection, as compared to other areas, are a key factor for the distinct lack of anomaly-based detection in commercial systems [35]. This research focuses on two of those unique challenges: semantic gap, as detailed below, and the difficulty of adapting models to new attack patterns.

2.1.2 "Outside the Closed World" and Semantic Gap

In their seminal paper, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection", Sommer and Paxson shed some light on the challenges that make machine learning more difficult to integrate into anomaly-based intrusion detection than in other domains [35]. The main difficulties presented are outlier detection, the high cost of errors, semantic gap, diversity of network traffic, and the difficulties of evaluating NIDSes [35]. The particularly crucial issue of 'semantic gap' refers to the gap between the results of a NIDS (i.e., classification of network behaviour as malicious or benign) and 'transferring results into *actionable* reports for the network operator' [35].

Leading cybersecurity technology company CrowdStrike promotes the 1-10-60 rule: that companies should become aware of intrusions within the first minute, understand the attack within 10 minutes, and respond within 60 minutes [5]. However, cybersecurity teams often have a high workload due to the amount of network traffic or behaviour flagged as 'malicious', and it can require a considerable amount of time to understand a detected intrusion, as professionals must determine the location, category, status, and source of a potential attack with high accuracy [35]. This high-pressure incident response time is where our problematic gap could be greatly reduced: if professionals were given more information on the specific behavioural anomalies causing traffic to be labelled as malicious, their workload could be reduced and they could more easily determine methods of responding to this behaviour. This could be a valuable method of improving both operator response time and understanding of the anomalous network behaviour. Thus, the notion of reducing semantic gap ties closely into efforts to improve the explainability of machine learning models, as discussed in section 2.2.

While PABLO contributes novel insights into the issue of semantic gap, challenges such as high cost of errors and diversity and network traffic, as detailed in ‘Outside the Closed World’, are nonetheless prevalent and urgent. As these challenges are outside the scope of this research, the lack of mitigation for them in PABLO means that the performance of PyPABLO would not be the same in a real-world setting. This is transparently discussed in the ‘Evaluation’ section, but could also possibly be an area of value for future work.

2.1.3 Dos and Don’ts of Machine Learning

We now move from the subject of evaluating a model’s performance in relation to its *usefulness*, and address the evaluation of a model’s *statistical* performance. Though this may seem trivial to approach correctly, “Dos and Don’ts of Machine Learning in Systems Security” systematically evaluates 10 common pitfalls which plague the evaluation of research into machine learning-based security systems [4]. The top pitfalls identified by Arp, et. al. are presented as follows:

1. *Sampling Bias* occurs when training data does not represent the true data distribution of a particular problem, causing the results of a model trained on said data to become less trustworthy. To mitigate sampling bias, researchers should shed light on the limitations of their datasets and take measures such as extending their datasets with synthetic data or utilising multiple estimates of the true data distribution [4].

2. *Label Inaccuracy* occurs when the ground truth label for a classification-based security system’s training data is potentially inaccurate, or when the system cannot adapt to changes in adversary behaviour (label shift). This causes the security system to be inaccurate in its classifications. Dataset labels should always be verified, or labelling should be delayed until more data is collected and a more stable ground-truth can be verified [4].

3. *Data Snooping* can be split into three general categories: test snooping, where the model is trained on testing data; temporal snooping, where time dependencies are ignored within data (i.e. ‘snooping from the future’); and selective snooping, where the dataset is modified based on information not available in practice. Test and training data should be separated early to avoid data leakage in the development process of a model. Additionally, temporal dependencies should always be considered, and datasets with a variety of familiarity should be used [4].

4. *Spurious Correlations* occur when a learning model makes false associations that correlate with their classification problem. Explanation techniques should be applied and the objective of a system should be defined well in advance to determine whether associations are actually spurious [4].

5. *Biassed Parameter Selection* is an exceptional case of data snooping which occurs when final parameters of a model are indirectly dependent on its testing dataset. This can be mitigated by using a separate validation set for model selection and parameter tuning [4].

6. *Inappropriate Baselines* occur when a model is insufficiently evaluated in comparison to a variety of other models. Complex models should be compared to both complex and simple models, and simple models can be used as an appropriate baseline for performance. Additionally, the suitability of non-ML approaches should be considered for the problem domain [4].

7. *Inappropriate Performance Measures* occur when a lack of suitable performance measures are considered for a particular application scenario, which can cause the performance of a model to be incorrectly evaluated. Thus, researchers should include measures that are useful for gauging the performance of a model in practice [4].

8. *Base Rate Fallacy* occurs when results are misinterpreted due to class imbalance, e.g. if the negative class is predominant. Performance measures which account for class imbalance should be used, such as precision, recall, or the Matthews Correlation Coefficient [4].

9. *Lab-Only Evaluation* occurs when a machine learning model is not evaluated in a practical setting (i.e., it is evaluated only in a ‘closed-world setting’). Learning-based systems should be tested in conditions that approximate the real world, e.g. accounting for temporal or spatial relations, providing diverse network traffic, or monitoring storage and

runtime constraints [4].

10. *Inappropriate Threat Models* do not properly consider the hostility of a production environment, such as the influence of adversaries on real world learning-based systems through adversarial preprocessing, poisoning, and evasion. Threat models should be specific, should assume that there is an adaptive adversary attempting to exploit the weaknesses of the model, and should be monitored in all stages of development to mitigate potential vulnerabilities [4].

The 30 papers which Arp, et al. evaluated to identify common pitfalls were all published at 'top 4' security conferences, and each contained at least one pitfall [4]. While the results of a well-received paper cannot be entirely discredited by the presence of pitfalls, it is important that this research evaluates works related to learning-based intrusion detection models through a lens of awareness concerning the effects of potential pitfalls on experimental outcomes. In addition, this research will seek to mitigate and transparently acknowledge all ten pitfalls detailed in "Dos and Don'ts".

2.1.4 CICIDS2017 Dataset

Aiming to mitigate many of these pitfalls, Sharafaldin, et al. developed the CICIDS2017 dataset [32]. This dataset was developed in response to one of the most rampant issues in the systems security community– a lack of realistic, cutting-edge, detailed datasets available for evaluating learning-based detection systems– which enables pitfalls such as data snooping and sampling bias [33]. Sharafaldin, et al. created CICIDS2017 with the aim of addressing all eleven key characteristics identified by Gharib, et al. in 2016 for a valid intrusion detection dataset: "attack diversity, anonymity, available protocols, complete capture, complete interaction, complete network configuration, complete traffic, feature set, heterogeneity, labelling, and metadata" [32, 33]. CICIDS2017 and its successor, CSE-CIC–IDS2018, have become increasingly relied-upon in the network intrusion detection research community on the assumption that their stated quality is accurate [2, 8, 10, 39].

However, a few fundamental issues have been recently explored regarding CICIDS2017. Firstly, the dataset contains incomplete records, and has a significant class imbalance [7, 28]. Panigrahi et al. mitigated these issues by eliminating incomplete records and merging similar minority classes (e.g. FTP-Patator and SSH-Patator) in order to form new, more prevalent attack classes [28]. Engelen et al. further evaluated the correctness and validity of the CICIDS2017 dataset, finding that flows had been incorrectly split, that certain attacks were not properly executed, that the original CSV files contain attributes which could encourage shortcut learning in an intrusion detection model, that there was an unacceptable amount of noise in the dataset, and that the performance of the dataset was not properly evaluated [7]. Engelen et al. proposed a 'corrected' version of the CICIDS2017 dataset, which addresses many of these issues while still suffering from class imbalance and potential shortcut learning [7]. Therefore, despite the improvements brought by the datasets proposed by Panigrahi et al. and Engelen et al., pitfalls such as spurious correlations and label inaccuracy are still present in the original CICIDS2017 dataset and its 'improved' counterpart by Engelen et al., and are transparently discussed and mitigated in this research, as can be seen in the 'Evaluation' section [4, 7, 28].

2.1.5 Random Forest

The Random Forest classifier is a popular shallow learning classifier which creates multiple decision trees in randomly selected subspaces of the feature space at training time, and chooses the classification outputted by the most trees (i.e., the 'majority vote rule') [12]. Based on an algorithm designed by Tin Kam Ho, this classifier has a reduced risk of overfitting in comparison to decision-tree classifiers on fixed training datasets [12]. This is because the decision trees in Random Forest are each built with randomly selected variables in order to reduce the likelihood of trees containing similar biasses, i.e., the trees are less correlated [12]. The authors of the original CICIDS2017 dataset and the 'improved' CICIDS2017 dataset each utilised a Random Forest classification model to evaluate the performance of their datasets [7, 32]. Random Forest was also found to be a more promising algorithm for end-user explainability in a study by Herm, et al. [11] Therefore, this research will utilise the Random Forest classifier for PyPABLO due to its superior levels of both explainable and statistical performance, and in order to properly compare the results and explainability of this model to the original CICIDS2017 test model and its 'improved CICIDS2017' counterpart.

Random Forest classifiers have three main hyperparameters which impact their performance: node size, number of trees, and sample size [15]. Node size refers to the minimum number of features in a leaf node of a decision tree; decreasing node size increases the depth of the decision trees but can exponentially increase computation time [27]. In some software implementations of Random Forest, such as the PySpark RandomForestClassifier, node size is tuned by the parameter 'maximum depth', which controls the depth of each decision tree [37]. The number of trees in a Random Forest implementation has a positive correlation with its accuracy, but it has been observed that the greatest performance gain for a classifier occurs within the growth of one hundred decision trees [27]. Probst, et al. define sample size as ‘the number of observations [aka features] that are drawn for each tree’ [27]. Decreasing sample size allows for trees to be more diverse, resulting in less correlation and therefore greater accuracy overall, although the accuracy of individual trees may diminish [27]. The performance of a Random Forest classifier also may be impacted because a significant amount of time and resources are required to compute predictions for each decision tree [15]. In the final prototype for PyPABLO, Random Forest hyperparameters are tuned via a validation dataset in order to gauge the effects of utilising different hyperparameter values for random forest classifiers on the same dataset, and to determine the highest-performing model.

2.1.6 Chosen Attacks for Detection

The final relevant factor for the performance of the experiments conducted in this research (as detailed in Chapter 3) is the choice of attacks on which a model is trained and tested. The two attacks chosen as ’novel’ attacks for testing the PyPABLO model are port scan attacks and Patator brute force attacks.

Port scan attacks can be defined as the malicious use of port scanning tools such as nmap to gain information on potential intrusion vectors in a network [19]. These attacks are a common reconnaissance technique for attackers to discreetly probe target networks for vulnerabilities and can be a precursor to a more compromising attack [3]. For instance, if an attacker determines via port scanning that a device on the network has port 23 (Telnet) open and receiving, they could take advantage of that fact to then remotely connect to that machine. Lee, et al. categorise port scanning attacks into three main types: vertical scans (scanning multiple destination ports on one host), horizontal scans (scanning one particular destination port on many hosts), and block scans (scanning many destination ports on many computers) [19]. Since the CICIDS2017 and ’improved’ CICIDS2017 datasets only simulate port scan attacks between one victim and one attacker at once, only vertical port scans are simulated [7, 32]. We can therefore assume that a correctly-trained model should associate port scanning attacks or anomalous behaviour with features such as ’Destination Port’ and ’Fwd PSH Flags’ [29]. These assumptions will be revisited in the Evaluation Chapter (Chapter 5).

Brute force attacks in the context of network security are often taken to implicitly mean brute force attacks on the SSH protocol; in the CICIDS2017 datasets, the SSH and FTP modules of the brute-forcing tool Patator were chosen to represent brute force attacks [7, 17, 32]. These attacks often come in the form of attackers attempting to guess a user’s password in order to gain privileged access to their machine [25]. Since many users rely on simple or common passwords, or they may reuse passwords, guessing a user's password can be automated via tools like Patator which utilise dictionaries of passwords or brute force algorithms to attempt many possible password inputs [25]. Because of this, brute force attack flows often contain large amounts of data in one direction (from the attacker to the victim), with only small amounts of data being sent in return [29]. Therefore, one could expect a correctly-trained model to associate features such as 'Fwd Packet Length Mean', 'Init Win bytes forward', and 'Bwd Packet Length Std' with SSH-Patator or FTP-Patator traffic flows [29]. As with the previous port scan assumptions, this expectation will be revisited in Chapter 5.

Both port scan and brute force attacks are particularly insidious because they can easily masquerade as benign behaviour and are extremely effective for escalating an attacker's privileges. For instance, port scan attacks can often go un-observed because scanning tools such as nmap are equally as common in the toolbelt of a network administrator as they are in the belt of a malicious actor. Similarly, Brute force attacks can mimic the behaviour of a benign user who may have simply forgotten their password. By *training* a model on port scan or brute force attacks, the quantitative characteristics that identify these attacks can be observed. Additionally, by *testing* a model on 'novel' port scan and brute force attacks, insight can be gained regarding how said model can learn the correct characteristics that are associated with these attacks. We will explore both of these scenarios in this paper.

2.2 Evaluating Models via Shapley Value Explainability

2.2.1 Local Explainability

In machine learning, the 'explainability' of a model refers to the possibility of gaining insight into its behaviour when it classifies predictions, whether into the model's overall behaviour (global explainability) or just its reasoning regarding a particular prediction (local explainability) [38]. Herm, et al. posit that end users prefer explainability that focuses on the ‘reasoning' behind one particular prediction or classification by a machine learning model, i.e. local explainability [11].

Although the sample size of the study by Herm, et al. was relatively small, with only 223 responses, it can still provide some key insights into how the issue of commercial viability can be solved [11]. Firstly, the idea of a 'trade-off' between model explainability and model performance is not as 'clear cut' as is commonly believed [11]. That is, it is possible to create a model that is both high-performance and highly explainable to an end user. Secondly, their subjects preferred explanations that required less cognitive effort, such as those which focused on the 'why' of a particular classification decision [11]. Therefore, end users may prefer high-performance, locally-explainable machine learning models in their products. This aligns with the proposed research objective of creating a machine learning model that could leverage local explainability as a means of becoming more commercially viable.

2.2.2 SHAP – A Unified Approach to Interpreting Model Predictions

SHAP values, as calculated by the SHapley Additive exPlanations (SHAP) Python framework, serve as an effective and unified approach for reaching the objective of creating a highly explainable intrusion detection model [30]. SHapley Additive exPlanations, or SHAP, is a Python framework which utilises the classic game theory approach of Shapley Values to determine a feature's importance relative to the outcome of a model's prediction [20, 30, 39]. In other words, SHAP determines and visualises the 'contribution' of a particular feature to a model's classification decision. While simpler models are more 'inherently' explainable or understandable, such as SVMs, ensemble models such as Random Forest are too complex for humans to easily comprehend, and thus require explanation in order to be more effective for operators [11, 30]. Lundberg and Lee propose a novel class of additive feature importance methods [30], which contains six different explanation methods that all use the same explanation model:

$$g(z') = \phi_0 + \sum_{i=1}^{M} \phi_i z'_i, \tag{2.1}$$

where $z' \in 0, 1^M$, M is the number of simplified input features, and ϕ_i is in the set of rational numbers [30].

SHAP connects two additive feature importance methods, explanation technique of LIME and Shapley Values, in order to produce SHAP values, which measure the importance of a feature in determining a particular concrete prediction [30, 38]. These two feature importance methods are explained in section 2.2.3, below.

2.2.3 LIME and Shapley Values

Local Interpretable Model-Agnostic Explanations (LIME) is a concrete explanation technique which aims to achieve interpretability, local fidelity, model-agnosticism (explainability regardless of the model), and a global perspective when evaluating a model [22, 38]. Ribeiro, et al.

argue that when explaining an output, explainers should use an *interpretable representation*, i.e. one which prioritises interpretability to humans over representing the actual features used by a classifier [22].

In LIME, $x \in R^d$ denotes the 'original representation' of a particular outcome, and $x' \in \{0,1\}^{d'}$ denotes the binary vector which maps the original representation x to its more interpretable counterpart [22]. LIME formally defines an explanation as 'a model $g \in G$, where G is a class of potentially interpretable models' such as decision trees [22]. Given a black box model f, LIME trains g, an interpretable model (such as linear regression or a decision tree) based on a permuted dataset, which then acts as a surrogate or approximation of f. A surrogate, or interpretable, explanation is obtained through the following formula:

$$\xi(x) = \underset{g \in G}{\arg\min} \quad L(f, g, \pi_x) + \Omega(g) \tag{2.2}$$

Where:

- π_x represents the function defining locality around the 'original representation' x,
- L represents the fidelity function which measures how 'unfaithful g is in approximating f in the designated locality defined by π_x', and
- $\Omega(x)$ represents the complexity of g, i.e. the inverse of interpretability [22].

LIME attempts to minimise $L(f, g, \pi_x)$ so that the explainer g will be model-agnostic, and to minimise $\Omega(x)$ so that the explanation g of any particular instance will be as interpretable as possible [22]. Thus, LIME focuses on creating a local explanation model that is interpretable, faithful to the original model, and capable of explaining any shallow or complex model.

Shapley Values were developed in game theory to gauge the responsibility of a player for success in a collaborative game [31]. Shapley Regression Values are an additive feature attribution method which assign importance to a feature based on the effect of that feature being included in a dataset, and are computed via the following formula:

$$\phi_i = \sum_{S \subseteq F \setminus \{i\}} \frac{|S|!(|F| - |S| - 1)!}{|F|!} \left[f_{S \cup \{i\}}(x_{S \cup \{i\}}) - f_S(x_S) \right] \tag{2.3}$$

Where:

- F represents the set of all features,
- S represents a feature subset $\subseteq F$,

- two models are trained, $f_{S \cup \{i\}}$, which contains the feature i which is being analysed, and f_S, which does not contain i,
- x_S represents the values of each feature in S,
- and the differences between models which do and do not contain an arbitrary feature are computed for all possible subsets $S \subseteq F \backslash \{i\}$ [30].

Shapley Regression Values are thus a weighted average of all possible feature differences. The combination of LIME, which favours interpretability and adaptability, and Shapley Values, which provide a comprehensive method of determining feature importance, unite in the SHAP framework to provide a holistic solution for explainability in intrusion detection models.

2.2.4 An Explainable Machine Learning Framework for Intrusion Detection Systems (2020)

SHAP has previously had use cases such as natural language processing and computer vision; however, research on SHAP for intrusion detection explainability is relatively recent. The first use of SHAP for explaining learning-based intrusion detection models was published in 2020 [38]. In that work, Wang, et al. explored the use of SHAP for determining both local and global explanations, and for interpreting the results of both one-vs-all and multi-class classifiers. The paper suffers from its use of the NSL-KDD dataset. The dataset is relatively small, containing only four types of attacks and approximately 150,000 rows of data [13]. In addition, NSL-KDD suffers from class imbalance– for instance, User to Root attacks appear only 252 times, in comparison to the 53,385 times that DoS attacks appear. Finally, the dataset is extremely outdated and inaccurate, being based on the KDDCUP99 dataset, which is twenty-five years old and is made up of simulated attack data. Because of this, it is highly likely that the stated results are inaccurate. Nevertheless, the authors' use of SHAP force plots to create useful visualisations of local explanations to security personnel is particularly intriguing, and the paper has pioneered SHAP in the intrusion detection domain.

2.2.5 An Explainable Intrusion Detection System (2021)

In 2021, the authors Wang, et al. expanded on the use of SHAP for intrusion detection by creating global and local explanations from tree-type and convolutional neural network (CNN) models [39]. From the CICIDS2017 dataset[32], they created five two-category datasets, each containing a roughly equal distribution of an attack and benign data. The five attacks chosen were DDoS, DoS, FTP-Patator, SSH-Patator, and Port Scan. While the results indicated a good performance, SHAP global visualisations were created in order to indicate the level of unreliability for each model and determine which features were deemed more important for different attacks [39]. As mentioned in Section 2.1.4, CICIDS2017 suffers from multiple flaws including class imbalance, label inaccuracy, inaccurate separation of traffic flows, and potential for shortcut learning [7]. Because of the unusual class distributions of each dataset, the data snooping required to separate the five datasets in this way, and the underlying flaws in CICIDS2017, the results are potentially not indicative of the practical performance of the two models that were investigated. In addition, the tables created in this paper to represent local explanations are unclear and would not be practically useful.

2.2.6 IOT Classification and Explanation (2022, 2023)

In early 2022, Le, et al. explored the use of SHAP for evaluating, optimising and explaining tree-based models trained on three different IoT datasets [18]. The authors explored both binary classification and multiclass classification, and claimed 100% accuracy, F1 score and ROC AUC metrics for multiple datasets [18]. They also used cross-validation to choose the highest performing hyperparameters for their classifiers. Unfortunately, this research suffers from datasets with a distinctly unequal class distribution, the most egregious of which being the NF-BoT-IoT-v2 dataset which contains 98% malicious traffic and only 2% benign traffic [18]. Although the performance of their proposed method is higher than other classifiers trained on the same datasets, the results are likely non-reproducible in real life and the practical viability of their proposed classification method cannot be verified. Due to their use of the inappropriate performance metric ROC AUC, which masks class imbalance and other pitfalls, their results are potentially also inflated within the study [4].

Keshk et al. proposed a novel framework, SPIP (SHAP, Permutation feature importance, Individual conditional expectation, Partial dependence plot), in 2023 [16]. Their objective was to create global and local explanations that were useful to 'end-users, security experts and researchers' [16]. They describe Permutation Feature Importance (PFI) as determining feature importance based on the variability of the model's error when a feature is present or absent. Individual Conditional Explanation (ICE) plots curves representing the relationship between the label of an observation and the value of a particular feature. Partial Dependence Plot (PDP) visualises the global relationship between a model prediction and a particular subset of features. It is based on the unrealistic assumption that there is no correlation between the subset of features chosen and the subset of features not chosen [16]. Utilising all methodologies in SPIP, the researchers aim to provide both IDS debugging information and explanations of attacks to network operators. The researchers acknowledge one particular pitfall, namely, the use of the NSL-KDD dataset. Additionally, while SPIP is introduced as a potential anomaly detector for novel attacks, the researchers do not attempt to test their SPIP-based model on any 'novel' attacks. Despite these drawbacks, the SPIP methodology serves as an interesting area for future research, and should be explored further to determine its practical viability.

Chapter 3

Design & Specification

This section outlines the experiments, design and iterative development for PyPABLO, a learning-based network intrusion detection model prototype. In total, four experiments were designed: two baseline experiments which utilise the classifier and hyperparameters by Engelen et al. [7] and a random split for training and test data, one experiment which utilises the classifier and hyperparameters that were used by Engelen et al. [7], with a custom data split to test the model's ability to detect novel attacks, and one experiment which utilises validated hyperparameters for detecting novel network attacks. The table below details the differences between each experiment:

Experiment	Dataset Used	Attacks Detected	Parameters Used
1.1	CICIDS2017	Known	Engelen
1.2	Improved CICIDS2017	Known	Engelen
2.1	Improved CICIDS2017	Novel	Engelen
2.2	Improved CICIDS2017	Novel	Optimised

The purpose and high-level design of each experiment is detailed below, along with diagrams showing the pipeline of each experiment.

3.1 Baseline Experiments

The purpose of these baseline experiments is twofold: firstly, to attempt to recreate the performance of a Random Forest classifier implemented by Engelen, et al., and secondly, to compare the performance of two models which were identical, excepting that one will be trained and tested on the original CICIDS2017 dataset, and one will be trained and tested on the improved

CICIDS2017 dataset by Engelen, et al. [7, 32]. This will enable analysis and comparison of the quality of each dataset during runtime, and augment dataset quality observations by Engelen et al. and Sommer, et al. [7, 35]. All experiments will be evaluated with regards to their accuracy, F1 scores, precision, recall, and SHAP values.

3.1.1 Baseline Experiment - Original CICIDS2017 Dataset

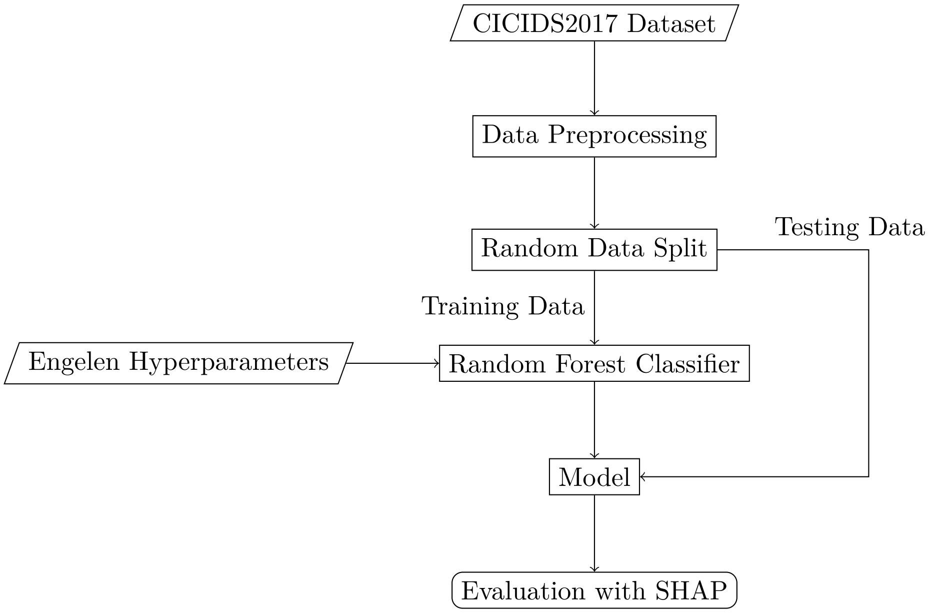

Figure 3.1: Baseline Experiment - Original CICIDS2017 Dataset Pipeline

This experiment will utilise the Pyspark framework in order to preprocess and split data, and to create the model [37]which will be trained and tested on the original CICIDS2017 dataset [32]. The CICIDS2017 dataset will firstly be cleaned and made compatible with Pyspark and SHAP, and then it will be randomly split 75:25 into training and testing datasets. All experiments will utilise a Random Forest classifier; this baseline experiment will maintain the hyperparameters of one hundred trees and a maximum depth of 30 for each decision tree, as utilised by Engelen, et al. in their own experiments [7].

3.1.2 Baseline Experiment - 'Improved' CICIDS2017 Dataset

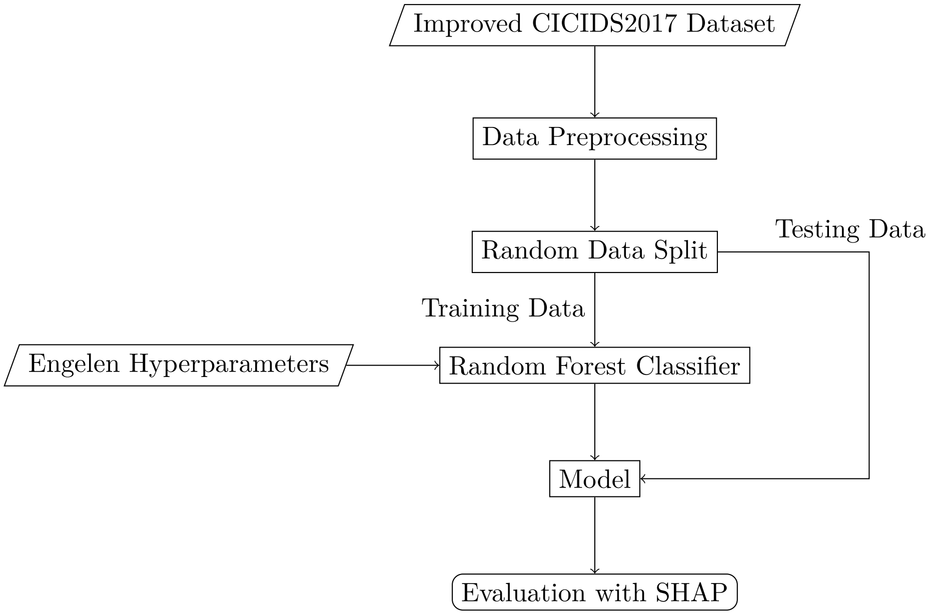

Figure 3.2: Baseline Experiment - Improved CICIDS2017 Dataset Pipeline

This experiment will be implemented identically to the experiment detailed in 1.1; however, this experiment will utilise the 'improved' CICIDS2017 dataset [7]. Therefore, this experiment will be a baseline for performance of the model developed by Engelen, et al. when splitting the 'improved' CICIDS2017 dataset randomly. The experiment will be evaluated with the same metrics and frameworks as all other experiments (see Section 3.1).

3.2 Custom Attack Split Experiments

The purpose of these 'custom attack split' experiments is to build a model which can effectively detect novel network attacks. Rather than executing a random split between training and testing data, testing data will contain all port scanning and brute forcing attacks, and training data will contain the remainder of the traffic flows in the 'improved' CICIDS2017 dataset. Firstly, the Engelen experiment (a high-performance research environment attack detection model, [7]) will be evaluated on its ability to detect novel attacks, and then a novel, optimised model will be designed for more effective novel attack detection.

3.2.1 Novel Attack Detection with Engelen Hyperparameters

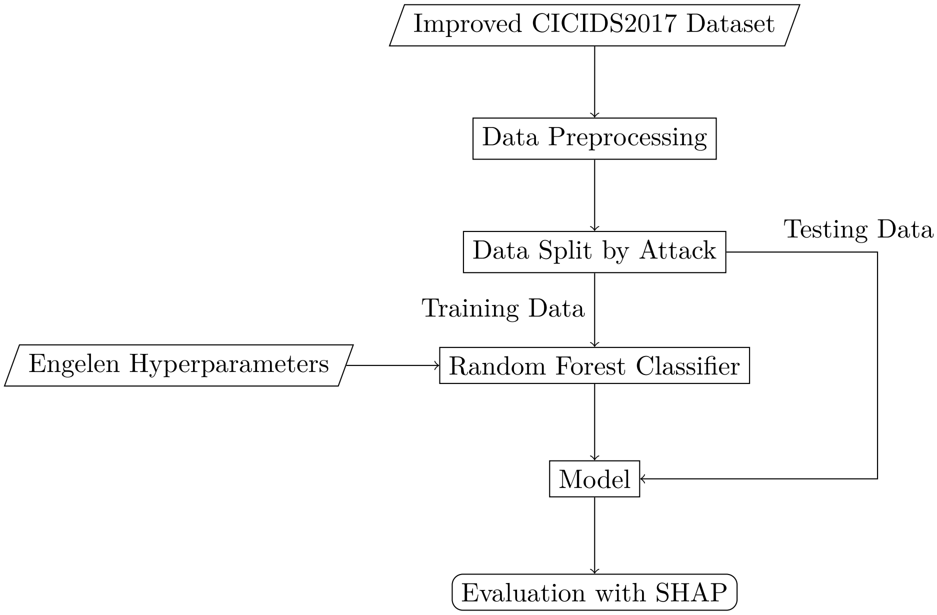

Figure 3.3: Pipeline for Novel Attack Detection with Engelen Hyperparameters

The purpose of this third experiment is to mimic the work of Engelen et al. in order to form a baseline of their model's performance when tested on novel attacks [7]. While Experiment 1.2 attempts to recreate their stated results, Experiment 2.1 aims to evaluate whether their model's performance remains as high when evaluated on novel attacks – a key element for commercially successful anomaly-based detectors. A unique permutation of the improved CICIDS2017 dataset will be used in order to evaluate the Engelen experiment's ability to detect port scan attacks as well as Patator attacks when not trained on those attacks. This experiment can therefore be seen as a baseline for a high-performance model, implemented in a research environment, when confronted with novel attacks.

Similarly to 1.2, this experiment will utilise the same hyperparameters and dataset as Engelen, et al. However, rather than randomly splitting the dataset into training and test data, the test data will be comprised of all port scan and patator (the chosen representation of brute force in the improved CICIDS2017 dataset) attacks, and the training data will be comprised of all other traffic flows in the improved CICIDS2017 dataset.

3.2.2 Novel Attack Detection with Cross-Validated Hyperparameters

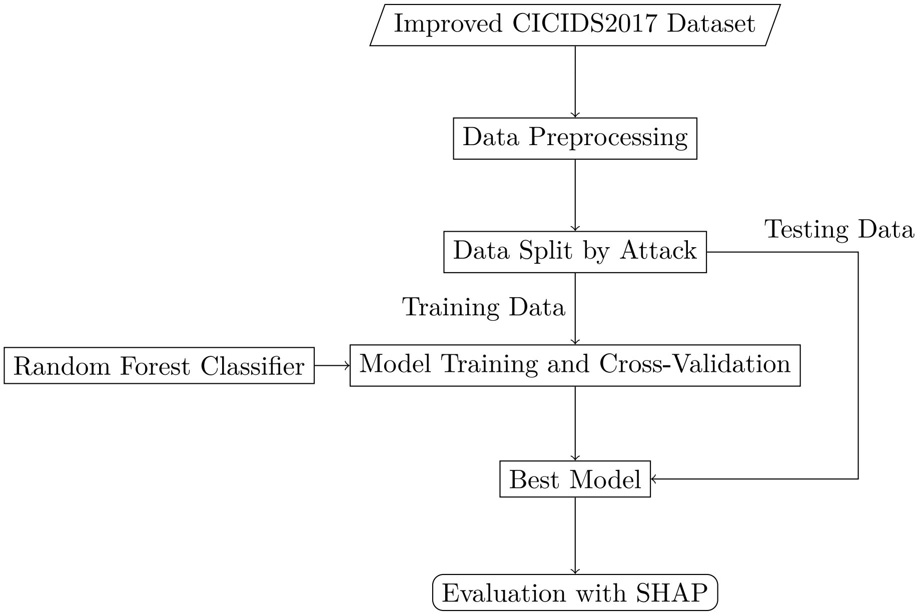

Figure 3.4: Pipeline for Novel Attack Detection with Engelen Hyperparameters

The objective of this experiment is to design an explainable Random Forest model with optimal settings for detecting novel port scan and brute force attacks. Training and testing data will be split identically to the experiment outlined in 2.1. Training validation will be implemented to choose the best model; thus, the hyperparameters of the final prototype model may differ from the Engelen settings [7]. To avoid test snooping [4], the hyperparameters of Experiment 2.2 are determined via a validation dataset composed of part of the original training data. This will result in a model which is optimised for known attacks, and its results will be used to determine the effects of optimising for known attacks on anomaly detection of novel attacks. The experiment will be evaluated with the same metrics and frameworks as experiments 1.1, 1.2 and 2.1, and as with all other experiments, its results will be locally explained via

SHAP to determine the reasoning behind the decisions of this relatively ‘black-box’ ensemble classification model when tasked with detecting novel attacks. This final experiment and its validated model will be used as the final iteration for the PyPABLO prototype.

Chapter 4

Implementation

This chapter details key aspects of the package implementation of the PABLO methodology as PyPABLO. The first section details the general layout for all experiments and the key differences between experiments. The second section details the semantic and syntactic implementation of each module in the PyPABLO package, as well as the full PyPABLO pipeline. Finally, the third section discusses the most notable challenges and aspects of the PyPABLO implementation.

4.1 Experiment Setup

This section first describes the generalised pipeline for model creation and evaluation, then the implementation for customising each experiment.

4.1.1 Generalised Experiment Layout

The general pipeline for all four experiments is detailed as follows:

- **Load, preprocess and save datasets.** Datasets are loaded from directories of CSV files, preprocessed, and then the processed data is saved to minimise unnecessary computations.

- **Split the dataset into train and test datasets.** Depending on the experiment, datasets are either split randomly into train and test sets, or they are split by attack type (specified by the encoded 'Label' column in both datasets).

- **Create, train and save the Random Forest classification model.** Hyperparameters for the model are either hardcoded or generated programmatically.

- **Evaluate and visualise Random Forest model via SHAP and other metrics.** As stated in the Background and Design chapters, these metrics align with those recommended by Arp, et al [4].

While all experiments followed this general pipeline, each experiment had a unique implementation of three control variables: dataset (either CICIDS2017 or ‘improved’ CICIDS2017), data split (either a random 70:30 train:test split or a custom attack split), and Random Forest hyperparameters (either following the tuning by Engelen et al. or utilising training validation to optimise the model). The control variable setup for each experiment is detailed below.

4.1.2 Experiment 1.1 – ‘Baseline Experiment - Original CICIDS2017 Dataset’

As detailed in the Design chapter, the control variables for this experiment were set up as follows:

- The dataset used was the original CICIDS2017 dataset [32]; this is specified as a command line argument.
- The data split used was a random 75:25 training:testing split.
- The hyperparameters for the Random Tree classifier aligned with experiments by Engelen, et al. [7]– maximum tree depth was hardcoded to 30, and the number of trees was set to 100.

The MacOS terminal command for executing this experiment is as follows:

```
python3 create_baseline_model.py MachineLearningCVE
```

4.1.3 Experiment 1.2 – ‘Baseline Experiment - ‘Improved’ CICIDS2017 Dataset’

In this experiment, the control variables were set up as follows:

- The dataset used was the ‘improved’ CICIDS2017 dataset[7]; this was specified as a command line argument
- Data split and hyperparameter tuning were identical to Experiment 1.1.

The MacOS terminal command for executing this experiment is as follows:

```
python3 create_baseline_model.py ImprovedMachineLearningCVE
```

4.1.4 Experiment 2.1 – 'Engelen Experiment with Custom Attack Split'

In this experiment, the control variables were set up as follows:

- The dataset used was the 'improved' CICIDS2017 dataset[7]; this was specified on the command line as in experiments 1.1 and 1.2
- The data split used was set by including 'pbp' as a command line argument when executing the experiment code; this reserved all port scan and Patator attacks for testing while using all data from the dataset as training data.
- The hyperparameters were identical to those in Experiments 1.1 and 1.2, i.e. following the tuning by Engelen, et al. [7]

The terminal command for executing this experiment on MacOS is as follows:

```
python3 create_baseline_model.py ImprovedMachineLearningCVE pbp
```

4.1.5 Experiment 2.2 – 'Optimised Experiment with Custom Attack Split'

In this experiment, the control variables were set up as follows:

- The dataset used was the 'improved' CICIDS2017 dataset, specified as a command line argument.
- The data split used was identical to that in Experiment 2.1, and required the inclusion of the 'pbp' keyword when executing the experiment code
- The hyperparameters were set via a cross-validation function supported by PySpark; the method of cross-validation will further be detailed in the create_pypablo_model.py subsection below.

The MacOS terminal command for executing this experiment is as follows:

```
python3 create_pypablo_model.py ImprovedMachineLearningCVE pbp
```

4.2 Package Implementation

The implementation of the PyPABLO package involved five modules for the four pipeline stages: preprocessing, data splitting, creation of the machine learning model, and evaluation of

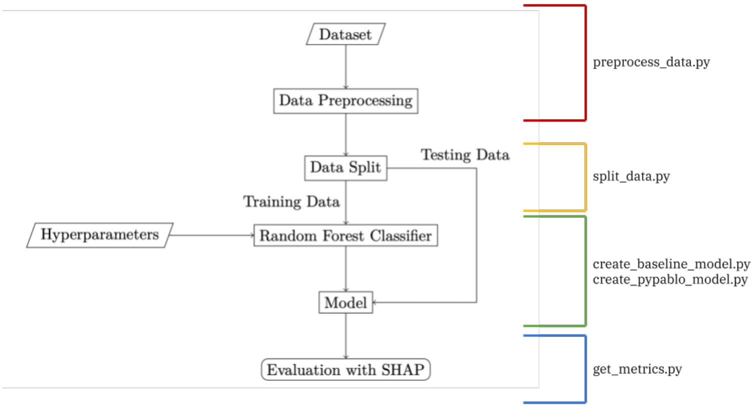

Figure 4.1: A representation of how modules corresponded to the designed general pipeline for all experiments

the model. All experiments shared the same preprocessing, data splitting and model evaluation modules; model creation and pipeline execution functionalities were split between two modules: `create_baseline_module.py` for experiments 1.1-2.1 and `create_pypablo_model.py` for experiments 2.2.

This section will detail the semantic meaning of each module, augmented with notable code snippets from the modules.

4.2.1 preprocess_cic.py

Implementation This module is responsible for preprocessing a directory of CSV files which have the same schema as the original or improved CICIDS2017 datasets [7, 32] (excepting a duplicate 'Fwd Header Length' column in the original CICIDS2017 dataset [7], which had to be manually deleted). Preprocessing these datasets can be abstracted as the first stage of the PABLO machine learning pipeline. The name of the CSV directory to be processed is specified as a command line argument. When `preprocess_cic.py` is executed, it calls the `preprocess_data` function.

Data Preprocessing `preprocess_data` takes one argument: `csv_dir`, the name of a CSV file or directory to be processed. This function calls three functions sequentially:

1. `load_data`
2. `clean_dataframe`

3. `create_ground_truth`

This produces a PySpark DataFrame representation of the processed dataset. This DataFrame is saved as a CSV directory in the Data/Processed subdirectory. `preprocess_data` is called with the argument `sys.argv[1]`. It is displayed in Listing 4.1, below.

Data Loading `load_data` is a helper function which takes one argument: `csv_path`, a string representing an absolute path to a CSV file or directory. It returns a PySpark DataFrame containing the row data and schema of the directory.

DataFrame Cleaning `clean_dataframe` is a helper function which takes one argument: `df`, a PySpark DataFrame. It deletes columns which may cause spurious correlations, such as ‘Timestamp’, or ‘Src IP’, which contains the source IP for a traffic flow and was present in the improved CICIDS2017 dataset [7]. Columns containing ’attempted’ attacks are also deleted from the DataFrame to avoid potential test snooping [4]. This function additionally removes any duplicate rows or rows with problematic values, such as infinite or NaN values. `clean_data` returns a PySpark DataFrame.

Ground Truth Creation `create_ground_truth` is a helper function which takes one argument: `df`, a PySpark DataFrame. This function creates an encoded ground truth column, ‘GT’, which contains a float based on the value in the ‘Label’ column for each row. It returns a PySpark DataFrame identical in schema and rows to `df`, but with the extra ‘GT’ column. Encoding was implemented via PySpark StringIndexer, which can encode ’Label’ values based on their alphabetical ordering. This allowed for consistent label encoding between experiments. `create_ground_truth` is displayed below, in Listing 4.2.

To summarise, this module calls the `preprocess_data` function, which sequentially executes `load_data`, `clean_data`, and `create_ground_truth` to create a processed PySpark DataFrame from the CSV data directory specified at execution time. These three functions create a PySpark DataFrame from the specified file or files, remove problematic columns and rows, and create a binary encoded column “GT” (ground truth), which simplifies all attack labels into values of 1 (benign) or 0 (malicious).

Notable Source Code

Listing 4.1: preprocess_data function

```
def preprocess_data(csv_dir: str):
    rel_csv_path = ”/Data/Raw/” + csv_dir
    df = load_data(rel_csv_path)
    df = clean_dataframe(df)
    df = create_ground_truth(df)
    new_csv_path = os.getcwd() + "/Data/Processed/" + csv_dir
    df.write.option("header", True).mode("overwrite").csv(new_csv_path)
```

Listing 4.2: create_ground_truth function

```
def create_ground_truth(df: DataFrame) -> DataFrame:
    string_indexer = StringIndexer(inputCol="Label", outputCol="GT",
                                   stringOrderType="alphabetAsc")
    df = string_indexer.fit(df).transform(df)
    df = df.withColumn("GT", col("GT").cast(IntegerType()))
    return df
```

4.2.2 split_data.py

Implementation This module is responsible for splitting data into train and test PySpark DataFrames, and for isolating particular Row objects based on column values. Its `split_dataset` function is called by both `create_baseline_model.py` and `create_pypablo_model.py`. It contains other helper functions for isolating and splitting data within a PySpark DataFrame.

`split_data.py` can be abstracted as the second stage in the PyPABLO pipeline. `split_data` is not meant to be executed in isolation, rather, it contains the following data-splitting helper functions which are used in the model creation modules: `split_dataset`, `match_keyword`, `isolate_attacks`, `get_row_with_matching_cols_index`, `get_row_with_matching_cols`, and `get_gt_row`. These functions are explained below.

Dataset Splitting `split_dataset` takes two arguments: `df`, a PySpark DataFrame, and `args`, a list of strings, which are specified at execution time for the model creation modules. When `split_dataset` is called, it evaluates the size of `args` to determine whether an optional keyword has been specified for a custom dataset split. If no keyword is specified, `split_dataset` proceeds with a standard 75:25 random split between train and test data, and returns the two train and test DataFrames. If any keyword has been specified, the `match_keyword` function is called, which creates and returns its own train and test datasets.

'Split' Keyword Matching `match_keyword` takes two arguments: `df`, a PySpark DataFrame, and `keyword`, a specified keyword. If the `keyword` matches any cases within the function, the function returns a custom split of training and test data. For instance, specifying the `pbp` (port scan, brute force, patator) keyword calls the `isolate_attacks` function with arguments for returning a training and test DataFrame where port scan, SSH-Patator, and FTP-Patator attacks have been isolated in their own test DataFrame separately from the rest of the data. If the keyword has no matches, a standard 75:25 random split for training and test data is returned. In the event that PyPABLO is used for future research on detecting other novel attacks, extensibility has been ensured: new keywords with custom train-test splits can be easily specified in the `match_keyword` function. `match_keyword` returns the train and test DataFrames that it has created. It is shown in Listing 4.3, below.

Attack Class Isolation `isolate_attacks` is a function which takes two arguments: `df`, a PySpark DataFrame, and `search_values`, a list of float values which match potential values in the 'GT' (ground truth) column of the DataFrame. Any row with a GT value matching a value in `search_values` is separated into an isolated DataFrame. This function returns two DataFrames for training and testing: one which contains all data excepting the rows matching those 'GT' values, and one containing the rows with those specified values. The pattern-matching snippet of `isolate_attacks` is shown in Listing 4.4.

Constraint-Matching via Index `get_row_with_matching_cols_index` is a helper function that returns the index of a row of a PySpark DataFrame, where two specified columns are equivalent to a passed value. It takes four arguments: `df`, the DataFrame, `val`, the passed value, and `col_1` and `col_2`, which are the two names of columns to compare. It returns the index of the first Row object found that adheres to these constraints, or `None` if no index is found. This helper function was initially used in `create_baseline_model.py` to find the indexes of observations that should be visualised, but instead, `get_row_with_matching_cols` was used for greater efficiency.

Constraint-Matching via Row `get_row_with_matching_cols` is a helper function which takes the same four arguments as `get_row_with_matching_cols_index`. The only difference between this function and the above function is that `get_row_with_matching_cols` returns a Row object that adheres to the passed constraints (`val, col_1, col_2`). This function is called in `create_baseline_model.py` to find correctly-predicted observations for SHAP visualisation, revealing the most impactful features for a particular prediction of a model.

`get_gt_row` is similar to `get_row_with_matching_cols`, but it takes only three arguments: `df`, `val` and `col`. This is because this helper function returns a Row object based on the value of only one column `col`. `get_gt_row` is called in both `create_baseline_model.py` and `create_pypablo_model.py` to provide observations to plot when a model cannot correctly predict certain attack classes. the source code for `get_gt_row` is provided below as Listing 4.5.

Notable Source Code

Listing 4.3: match_keyword function

```
def match_keyword(df: DataFrame, keyword: str):
    match keyword:
        case ”pbp”:
            training_data, test_data = isolate_attacks(df,
                                                        [”10.0”, ”11.0”, ”7.0”])
        case '' | _:
            training_data, test_data = df.randomSplit([0.75, 0.25])
    return training_data, test_data
```

Listing 4.4: isolate_attacks snippet

```
    for search_value in search_values:
        attack_rows = df.filter(df[”GT”] == search_value).collect()
        attack_df = spark.createDataFrame(data=attack_rows,
                                          schema=isolated_df.schema)
        isolated_df = isolated_df.union(attack_df)
```

Listing 4.5: get_gt_row snippet

```
return_row = None
    df_collect = df.collect()
    for row in df_collect:
        if row.__getitem__(col) == val:
            return_row = row
            break
    if return_row is None:
        print(”row␣not␣found␣for␣GT␣of␣value:␣”, val)
    return return_row
```

4.2.3 get_metrics.py

Implementation This module is responsible for getting metrics of the performance of all experiments. It contains the following helper functions, many of which are implemented in `create_baseline_model.py` and `create_pypablo_model.py`: `get_unique_model_info`, `get_test_data_directory`, `get_prediction_metrics`, `get_shap_values`, and `get_shap_values_multicore`.

Unique Model Information `get_unique_model_info` is a helper function that extracts a unique sequence of numbers from the name of a saved model. This function was initially used to re-load saved models and evaluate them with a saved test dataset in a separate program; however, when an error with re-evaluating stored models was encountered, the decision was made to create and evaluate models in the same program. For this reason, `get_unique_model_info` is not present in the final iterations of `create_baseline_model.py` or `create_pypablo_model.py`. The 'stored model' error is explored further in the 'Notable Challenges and Aspects' section, below.

Absolute Path to Test Data `get_test_data_directory` is a helper function with a similar purpose: taking one argument, `model_id`, this function returns an absolute path to saved test data that corresponds to the specified model. It is also not present in the final iterations of either model creation module.

Prediction Metrics `get_prediction_metrics` is a function which takes four arguments: `predictions_df` (a PySpark DataFrame), `label_col` (a string, the name of the label column), `prediction_col` (a string, the name of the column containing predictions, to be compared to `label_col`), and `split` (a list of passed command line arguments). This function returns the accuracy, precision, recall, F1 score, and precision-recall AUC for a given DataFrame of model predictions. These values are utilised for model evaluation purposes, as can be seen in the Evaluation chapter (Chapter 5). It serves as the main source of performance metrics for models created by `create_baseline_model.py` or `create_pypablo_model.py`.

Single-Node SHAP Value Calculation `get_shap_values` is a function that takes two arguments: `model`, a tree-based model, and `test_df`, a pandas DataFrame. This function is the single-node implementation for calculating SHAP values, in order to determine local explanations of particular prediction observations and gain more knowledge about a model's reasoning. It is included in the pipelines of both `create_baseline_model.py` and `create_pypablo_model.py`. This function creates and then extracts SHAP values from a SHAP TreeExplainer (which provides explanations for tree-based models). This function then returns the SHAP values and the Explainer.

The pandas DataFrame (test_df) argument is a result of the SHAP library's incompatibility with PySpark DataFrames. Despite this particular incompatibility, the SHAP TreeExplainer is actually compatible with tree-based PySpark classifiers, such as the RandomForestClassifier [37].

Multi-Node SHAP Value Calculation get_shap_values_multicore is a function which utilises multiple nodes to extract SHAP values from a PySpark model (model) and a pandas DataFrame (test_df). It utilises code snippets from a blog post by Ebrahimi and Patel [6], which have been cited in the module as necessary. This function takes advantage of the parallelism inherent in PySpark to efficiently calculate SHAP values for large datasets. Firstly, a SHAP TreeExplainer is created (as in the single-node SHAP value function, above). Secondly, a pandas UDF calculate_shap is created, which takes an iterator of pandas DataFrame objects as input and yields SHAP values for each DataFrame in the iterator [6]. After that, SHAP values for the entire DataFrame are extracted via the PySpark mapInPandas function, which partitions the DataFrame and applies calculate_shap to each partition concurrently. This function shows promise for calculating SHAP values in a highly efficient manner [6], but suffers from its extraction of SHAP values as a PySpark DataFrame. Due to difficulties converting this DataFrame into the correct format for SHAP visualisation, this function was ultimately not used in the final pipelines of any model creation module. Altering this function for compatibility with SHAP visualisation functions, e.g. shap.force_plot is a viable area of future improvement for both create_baseline_model.py and create_pypablo_model.py.

Notable Source Code

Listing 4.6: get_shap_values function

```
def get_shap_values(model, test_df):
    explainer = shap.TreeExplainer(model)
    shap_values = explainer.shap_values(test_df,
                                        check_additivity=False)
    return shap_values, explainer
```

Listing 4.7: get_shap_values_multicore snippet

```
def calculate_shap(iterator: Iterator[pd.DataFrame])
                -> Iterator[pd.DataFrame]:
```

```
    for X in iterator:
        yield pd.DataFrame(
            explainer.shap_values(np.array(X), check_additivity=False)[0],
            columns=cols,
        )

return_schema = StructType()
for feature in cols:
    return_schema = return_schema.add(StructField(feature, FloatType()))
print("Built␣return␣schema")

# must be spark df so that data is compatible with calculate_shap
spark_test_df = spark.createDataFrame(test_df)
shap_values = spark_test_df.mapInPandas(calculate_shap,
schema=return_schema)
print("made␣spark␣shap_values␣df")
```

4.2.4 create_baseline_model.py

Implementation The role of this module is to create trained models for experiments 1.1, 1.2 and 2.1. It is therefore the implementation for creating baseline models rather than an optimised model prototype (Experiment 2.2).

Execution and Output `create_baseline_model.py` is executed with one mandatory command line argument (the name of a CSV directory), and one optional command line argument, (a keyword for specifying a custom attack split). This module creates, trains, evaluates, visualises, and saves a Random Forest model trained on this CSV data in the Scripts/Trained_Model subdirectory. `create_baseline_model.py` is different from previously discussed modules in that it is not a collection of helper functions, but simply a series of executed statements which make use of helper functions. Note that the CSV directory or file must be located in the Data/Processed subdirectory. This following subsection describes the pipelines steps that are executed for the `create_baseline_model.py` module.

Dataset A DataFrame `df` is created utilising the `load_data` function from `preprocess_cic.py`.

Data Preprocessing and Splitting Train and test PySpark DataFrames are created from df using the split_dataset function from split_data.py. Both the training DataFrame and the testing DataFrame are further transformed to remove non-numerical features and add a ‘features’ column of type SparseVector, which contains the values for all features in a particular row and is required for model fitting and evaluation in PySpark. The source code for this pipeline stage is displayed in Listing 4.8, below.

Hyperparameters, Random Forest Classifier Secondly, a RandomForestClassifier is tuned to match the Engelen experiment hyperparameters (maximum tree depth set to 30 and number of individual decision trees set to 100), and is fitted to the training data. Because this module is for baseline experiments, model optimisation via cross-validation has not been implemented. This is displayed in Listing 4.9, below.

Model Testing and Predictions The Random Forest model then is tested and a DataFrame is created, containing the model's predictions for the test dataset. The get_prediction_metrics function from get_metrics.py is utilised to print the model's accuracy, precision, recall, F1 score, and PR AUC to the terminal.

SHAP Calculation and Visualisation Three observations are selected from the predictions DataFrame: one from the Port Scan attack class, one from the SSH-Patator class, and one from the FTP-Patator class. If the ’pbp’ keyword has been specified as a command line argument, the first three observations for these attack classes are chosen. This is due to the low accuracy of Experiment 2.1, which utilises the pbp keyword and does not classify predictions correctly. Otherwise, a correctly-predicted observation is selected for each attack class.

SHAP values are calculated for each observation utilising the get_shap_values function. For multi-class classification with one instance per class, SHAP values are stored as a list of arrays. The length of the list is equal to the number of attack classes in the specified dataset [20, 21]. Each row represents the SHAP values for a particular attack class, and each column represents the SHAP value for a specific feature [21]. For each attack class, the SHAP value also represents whether the classifier labelled the observation positively (meaning ’this observation belongs to this class’) or negatively (’this observation does not belong to this class’) [20, 30].

A locally-explainable SHAP force plot is then programmatically generated for each observation. SHAP force plots were chosen to visually represent each observation because they create an intuitive representation of the most impactful features that positively or negatively influenced a classifier's prediction [21]. This allows for an evaluation of feature importance for each attack class, an explanation of each prediction, and an investigation into potential spurious correlations [4] that a model has made. Relevant code for this pipeline stage is displayed in

Listing 4.10, below.

Saving Model and Test Data Finally, a unique model ID is created, the trained Random Forest model is saved in the Scripts/Trained_Models subdirectory, and its corresponding data for testing the model is saved in a CSV directory in the Data/Processed-Test subdirectory for future model evaluation.

Notable Source Code

Listing 4.8: DataFrame transformation snippet from create_baseline_model.py

```
training_data, test_data = split_dataset(df, args)
training_data = training_data.drop(”Label”)  # drop label column
# since not numeric
test_data = test_data.drop(”Label”)  # drop label columnsince not numeric

# exclude features from training that
# would cause spurious correlations
col_list = df.columns
columns_to_exclude = [”Label”, ”GT”]
features = list(set(col_list) - set(columns_to_exclude))

# use VectorAssembler to add 'features' column to df,
#containing values of all features used for training
vector_assembler = VectorAssembler(inputCols=features,
                                   outputCol=”features”)
training_data = vector_assembler.transform(training_data)
test_data = vector_assembler.transform(test_data)
```

Listing 4.9: Model training snippet from create_baseline_model.py

```
rf = RandomForestClassifier(maxDepth=30, numTrees=100, labelCol=”GT”,
                            featuresCol=”features”)
rf_model = rf.fit(training_data)
```

Listing 4.10: SHAP calculation and visualisation from create_baseline_model.py

```
rows = [portscan_row, ssh_row, ftp_row]
selected_rows_df = spark.createDataFrame(rows)
pd_df = selected_rows_df.drop('features', 'GT', 'rawPrediction',
                               'probability').toPandas()
shap_values, explainer = get_shap_values(rf_model, pd_df)
print("portscan plot: ")
index = 0
shap.force_plot(explainer.expected_value[0],
                shap_values[0][index],
                pd_df.iloc[index, :], matplotlib=True)
```

4.2.5 create_pypablo_model.py

Implementation The role of this module is to create trained models for experiment 2.2, i.e. 'Novel Attack Detection with Cross-Validated Hyperparameters'. `create_pypablo_model.py` is quite similar to `create_baseline_model.py`. It is executed with the same mandatory and optional command line arguments as `create_baseline_model.py`, it also saves a Random Forest model in the Scripts/Trained_Models subdirectory, and it also implements functionality for model training, testing, evaluation, SHAP calculation, and SHAP visualisation.

Where `create_pypablo_module.py` differs from its baseline model counterpart is that it utilises a train-validation data split to further tune and optimise the hyperparameters for its Random Forest classifier. As the creation module for the final PyPABLO prototype model, the model produced from this module will be evaluated on its ability to detect novel attacks in comparison to its Engelen Experiment counterpart (Experiment 2.1), which is created by the `create_baseline_model.py module`.

Dataset, Data Preprocessing and Splitting The dataset loading and train-test split pipeline stages are identical to those in
`create_baseline_model.py`; the test dataset follows the custom attack split in Experiment 2.1.

Hyperparameters, Random Forest Classifier Then, a RandomForestClassifier is created without any numerical hyperparameters specified. These hyperparameters are decided via cross-validation. Nine Random Forest classification models with different hyperparameters are tested on a 'validation set', i.e. a subset of the initial training dataset. In PySpark, the method

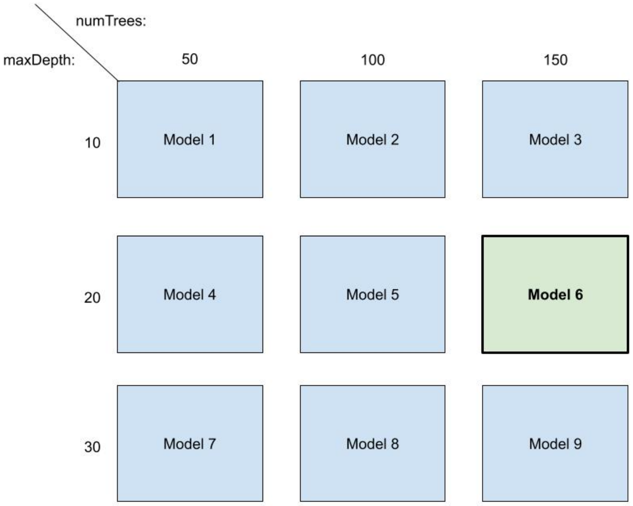

Figure 4.2: A visualisation of the param map implemented in Experiment 2.2; the green model was chosen as the best model for the PyPABLO prototype

of splitting a DataFrame into train and validation datasets is to firstly build a ‘param grid’, which sets specified hyperparameters (in this case, maxDepth and numTrees) to specified values ([10, 20, 30] and [50, 100, 150]) respectively [36]. This param grid will result in nine different models being trained, as can be seen in the figure below:

Then, a TrainValidationSplit model is instantiated, taking five arguments:

- `rf`, the RandomForestClassifier
- `param_grid`, the param grid
- `evaluator`, a BinaryClassificationEvaluator, which is used to evaluate each potential model
- the value of the attribute `trainRatio`, which is set between 0 and 1. This attribute represents the fraction of training data to be used for training (the remaining data is used as the validation set); in this case, it was set to 0.75.
- the value of the attribute `parallelism`, which defines the number of concurrent threads while running parallel algorithms. This was set to 5 to increase performance up to the computational limits of the research machine.

This `TrainValidationSplit` model is then fitted to the training data, resulting in nine different models being trained. The Train Validation process is illustrated in Listing 4.11, below. The best model out of the nine is separated and saved; the rest are discarded. In this implementation, the ‘best’ model is decided by `evaluator`, using two metrics: area under the ROC curve (receiver operating characteristic curve) and area under the precision-recall curve. The hyperparameter were chosen based on an analysis of Random Forest hyperparameters, as shown in the Background Chapter. Increasing the maximum number of trees and maximum depth of trees can increase a model’s accuracy for known data; the chosen ’best model’ provides insight on if this holds true for this particular classifier.

Given the custom attack split, the model selected as the ‘best model’ was Model 6, which interestingly has a maximum tree depth of 20 and 150 individual decision trees. The hypothesis of PyPABLO is that because this hyperparameter selection was evaluated as performing higher than a model with experiment 2.1’s hyperparameters on the validation set, then this classifier will perform higher than experiment 2.1 on detecting novel attacks. This is explored and evaluated further in Chapter Five.

Model Testing and Predictions The model is tested identically to Experiment 2.1 (i.e. `create_baseline_model.py` with the ’pbp’ keyword specified as a command line argument).

SHAP Calculation and Visualisation SHAP Value calculation and SHAP force plot visualisation are also identical to Experiment 2.1; that is, SHAP values and force plots are generated for three observations with ground truth values matching the Port Scan, SSH-Patator and FTP-Patator attack classes.

Saving Model and Test Data Identically to `create_baseline_model.py`, a unique model ID is created using the current date and time, the trained Random Forest model is saved in the Scripts/Trained_Models subdirectory, and its corresponding `testing_data` DataFrame is saved in a CSV format in the Data/Processed-Test subdirectory for future evaluation.

Notable Source Code

Listing 4.11: Train validation snippet

```
rf = RandomForestClassifier(labelCol=”GT”, featuresCol=”features”)


param_grid = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [10, 20, 30]) \
    .addGrid(rf.numTrees, [50, 100, 150]) \
    .build()


evaluator = MulticlassClassificationEvaluator().setLabelCol(”GT”)
tvs = TrainValidationSplit(estimator=rf,
                           estimatorParamMaps=param_grid,
                           evaluator=evaluator,
                           trainRatio=0.75,
                           parallelism=5)

rf_model = tvs.fit(training_data)
best_model = rf_model.bestModel
```

4.2.6 prove_shap_pyspark_compatibility.py

This small program is a proof of concept for the compatibility of a PySpark Random Forest Classification model and a SHAP Explainer object, when attempting to generate SHAP values. It was a 'prototype for the prototype' created in late December to ensure that the PySpark framework could be used with the SHAP library, as had been planned. This program contains similar pipeline stages to `create_baseline_model.py`, but does not align any particular experiment category. It is included in the source code and this report to provide additional insight into the development process for this research.

4.3 Notable Challenges and Aspects

4.3.1 Stored Model Error

While developing the PyPABLO implementation, one initial plan was to separate different pipeline stages into different executable programs. First, data would be preprocessed by `preprocess_cic.py`. Models would be generated and saved in `create_baseline_model.py` and `create_pypablo_model.py`. They could then be evaluated in a module named `evaluate_model.py`, and SHAP values and visualisations could be generated in a module named `generate_shap.py`. This isolated the execution of individual pipeline stages, allowing for useful features such as evaluating a model multiple times with different `evaluate_model.py` implementations. Initially, this was streamlined and drastically reduced time spent debugging. However, late in the development process, a serious issue was discovered. If one were to train and evaluate a PySpark model in the same program, its performance metrics were very different than if a model and its corresponding test dataset were saved, and then re-loaded for evaluation. In fact, saving and re-loading a model and its test data resulted in the model classifying every traffic flow as benign, regardless of its actual class! Obviously, this could not remain in the pipeline. Although debugging for this particular issue occurred for a significant amount of time, the issue could not be resolved whilst still retaining the model saving and re-loading features. Therefore, multiple pipeline stages (model creation, evaluation, and SHAP visualisation) were condensed into the create_model modules, which resulted in models that performed far more highly (i.e., correctly predicted attacks, and did not claim that every traffic flow was benign). The source of this issue remains unknown, but its effects on PyPABLO were resolved.

4.3.2 Multi-Core SHAP Value Generation

Another particularly interesting challenge regarding SHAP is the inefficiency of generating SHAP values. This is because Shapley Value generation is NP-hard [30, 31]. One main reason behind the use of the PySpark framework in this research is its efficiency with regards to processing large datasets [36], [16]. It also showed great promise for efficiently and concurrently generating SHAP values [6], calculating SHAP values for an entire 700,000-row test dataset in only four minutes in this research. Despite this incredible processing time, the implementation of PySpark and multi-node SHAP calculation resulted in a PySpark DataFrame containing all SHAP results, rather than the usual list of arrays that would be generated to contain multi-class SHAP values. Great difficulties were encountered whilst attempting to shape this DataFrame into the correct format for SHAP force plots [21], and efforts were not successful. The `get_shap_values_multicore` function was left unresolved.

However, the solution for this challenge was very simple. In fact, SHAP values did not have to be generated for an entire dataset containing hundreds of thousands of observations. The main objective of utilising SHAP–generating locally-explainable plots that could visualise the reasoning behind a particular observation– could be simply obtained by only generating SHAP values for a few observations. This results in a SHAP value calculation time that is still quite low, without wasting time and computational resources on needlessly calculating large amounts of SHAP values. This is also perhaps a more realistic implementation for how SHAP could be utilised for a commercial intrusion detector, since operators would generally only require SHAP plots for a small percentage of the network traffic that they would observe and filter.

Chapter 5

Results & Evaluation

This chapter of the report details the key results and insights from all experiments. The Experiment Settings section details the conditions under which each experiment took place. The two following 'Experiments' sections utilise accuracy, precision, recall, and F1 scores to provide a holistic evaluation of each experiment. They additionally use SHAP to visualise examples of local explanations, as designed and implemented in the Design & Specification (link) chapter and the Implementation chapter (link). The 'Overview of Results' section details key insights gleaned from all experiment results.

5.1 Experiment Settings

All experiments were conducted and evaluated on a 2022 Apple M2 laptop with 8 GB of RAM, an 8-core CPU, and a 16-core NPU.

5.2 Baseline Experiments

Table 5.1 displays the chosen metrics for experiments 1.1 and 1.2. Each experiment was run five times, and its metrics (accuracy, precision, recall and F1 score) were averaged from these five models.

The command used to run each experiment 5 times was:

```
code % for i in {1..5}; do python3 Scripts/create_baseline_model.py
[name of dataset]; done
```

Tables 5.1-5.3 display the results from experiments 1.1 and 1.2, as designed in subsections 3.1.1 and 3.1.2 in the Design & Specification chapter (Chapter 3), and as implemented by the

Experiment	Accuracy	Precision	Recall	F1 Score
1.1	0.998	0.999	0.999	0.998
1.2	0.994	0.999	0.993	0.993

Table 5.1: Results for Experiments 1.1 and 1.2

	Trials					
1.1	1	2	3	4	5	Average
Accuracy	0.998	0.998	0.998	0.998	0.998	0.998
Pr	0.999	0.999	0.999	0.999	0.999	0.999
Re	0.999	0.999	0.999	0.999	0.999	0.999
F1	0.998	0.998	0.998	0.998	0.998	0.998

Table 5.2: Full Results for Exp 1.1

	Trials					
1.2	1	2	3	4	5	Average
Accuracy	0.993	0.994	0.994	0.993	0.994	0.994
Pr	0.999	0.999	0.999	0.999	0.999	0.999
Re	0.993	0.993	0.994	0.993	0.993	0.993
F1	0.993	0.993	0.994	0.993	0.993	0.993

Table 5.3: Full Results for Exp 1.2

`create_baseline_model.py` module in the Implementation Chapter (Chapter 4).

These two baseline experiments both have nearly indistinguishable results upon a first glance; they both appear to perform quite well, including when class imbalance in the set is considered (as evaluated by the F1 score metric). SHAP force plots are used to provide operator-friendly examples of local explanations for each experiment, as seen below.

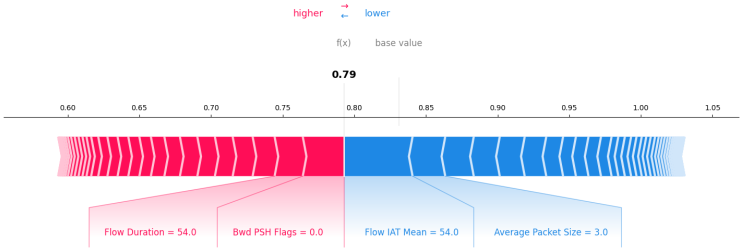

Figure 5.1: (Port Scan, 1.1)A SHAP force plot visualisation of feature importance for a Port Scan observation, Experiment 1.1

Figures 5.1 and 5.2 illustrate feature importance for individual Port Scan observations from Experiments 1.1 and 1.2.

Figure 5.1 (Port Scan, 1.1) suggests that Backward PSH Flags and Flow Duration were the most impactful features for correctly classifying the observation as being a Port Scan attack,

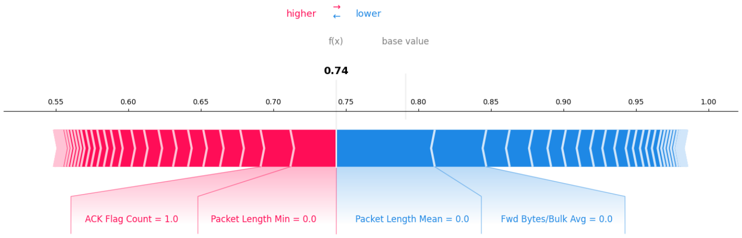

Figure 5.2: (Port Scan, 1.2) A SHAP force plot visualisation of feature importance for a Port Scan observation, Experiment 1.2

whereas Flow IAT Mean (‘the mean time between two packets sent in the flow’)[1] and Average Packet Size influenced the predicted score negatively.

Figure 5.2 (Port Scan, 1.2) depicts Packet Length Minimum and ACK flag count as being the most impactful positive features for the Port Scan attack, and Fwd Bytes/Bulk Avg (‘average number of bytes bulk rate in the forward direction’)[1] and Packet Length Mean being the most impactful features for influencing the classifier’s decision negatively.

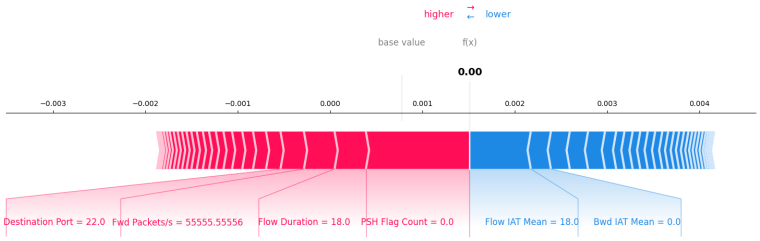

Figure 5.3: (SSH-Patator, 1.1) A SHAP force plot visualisation of feature importance for a SSH-Patator observation, Experiment 1.1

Figures 5.3 and 5.4 represent feature importance for individual SSH-Patator observations from Experiments 1.1 and 1.2.

Figure 5.3 (SSH-Patator, 1.1) depicts how PSH Flag Count, Flow Duration, Fwd Packets per second, and Destination Port most positively influenced the classifier’s predicted score, whereas Backward IAT Mean (‘mean time between two packets sent in the backward direction’ [1]) and Flow IAT Mean (‘Mean time between two packets sent in the flow’[1]) were most impactful for negatively influencing the classifier’s predicted score.

Figure 5.4 (SSH-Patator, 1.2) suggests many positively impactful features for the SSH-

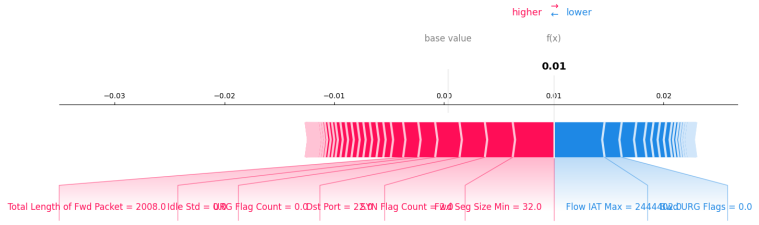

Figure 5.4: (SSH-Patator, 1.2) A SHAP force plot visualisation of feature importance for a SSH-Patator observation, Experiment 1.2

Patator attack, namely, Fwd Seg Size Min (‘Minimum Segment Size observed in the forward direction’ [1]), SYN Flag Count, Destination Port, URG Flag Count, Idle Std (‘standard deviation time a flow was idle before becoming active’ [1]), and Total Fwd Packet. The most negatively impactful features were Fwd Urg Flags and Flow IAT Max (‘Maximum time between two packets sent in the flow’ [1]).

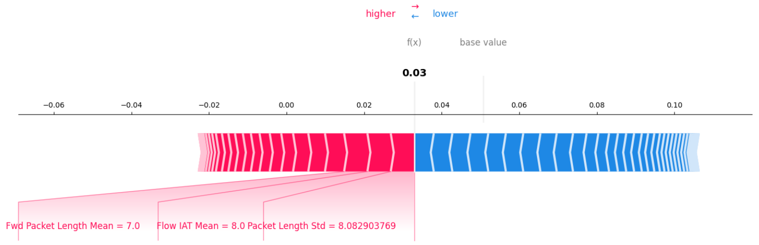

Figure 5.5: (FTP-Patator, 1.1) A SHAP force plot visualisation of feature importance for a FTP-Patator observation, Experiment 1.1

Figures 5.5 and 5.6 refer to the most impactful features for two FTP-Patator observations from Experiments 1.1 and 1.2.

Strangely, Figure 5.5 (FTP-Patator, 1.1) suggests only positive feature contributions: Packet Length Std (standard deviation length of a packet), Flow IAT Mean, and Fwd Packet Length Mean.

Figure 5.6 (FTP-Patator, 1.2) displays Flow IAT Max and Bwd Packet Length Mean (’Mean size of packet in backward direction’, [1]) as the most positively impactful features on the classifier’s prediction, and Bwd IAT Total (‘total time between two packets sent in the backward direction’ [1]), Total Length of Bwd Packet, and Active Min (‘minimum time a flow was active

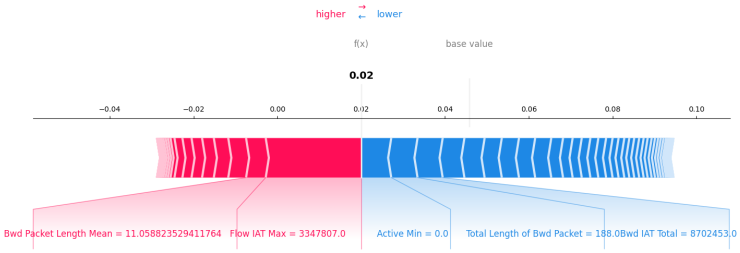

Figure 5.6: (FTP-Patator, 1.2) A SHAP force plot visualisation of feature importance for a FTP-Patator observation, Experiment 1.2

before becoming negative' [1]).

5.3 Novel Port Scan and Brute Force Attack Detection Experiments

This section pertains to the results from Experiments 2.1 and 2.2, as designed in Section 3.2. The following table represents the performance metrics for each 'novel attack split' experiment:

Experiment	Accuracy	Precision	Recall	F1 Score
2.1	0.0	Not Available	Not Available	0.0
2.2	0.0	Not Available	Not Available	0.0

Table 5.4: Results for Experiments 2.1 and 2.2

Obviously, both classifiers were completely unable to correctly classify 'novel' port scan and Patator attacks. For both datasets of model predictions, all flows were labelled incorrectly, and generally were labelled as benign traffic. The SHAP force plots below provide some insight into the reasoning behind both models' predictions.

Figure 5.7 (Port Scan, 2.1) displays the many features that contributed to the classifier's incorrect prediction of the attack as benign. The positively impactful features were Bwd Init Win Bytes ('The total number of bytes sent in initial window in the backward direction', [1]), Bwd IAT Mean ('Mean time between two packets sent in the backward direction', [1]), ECE Flag Count ('Number of packets with ECE', [1]), Total Bwd Packets, Subflow Bwd Bytes ('the average number of bytes in a sub flow in the backward direction', [1]), and Bwd Packet Length Min. The most negatively impactful feature was Source Port.

Figure 5.8 (Port Scan, 2.2) displays the most positively impactful features as being Idle

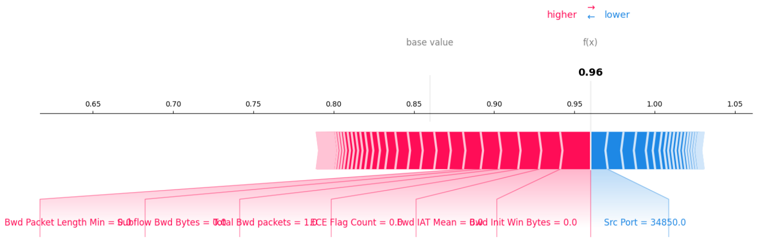

Figure 5.7: (Port Scan, 2.1) A SHAP force plot visualisation of feature importance for a Port Scan observation, Experiment 2.1

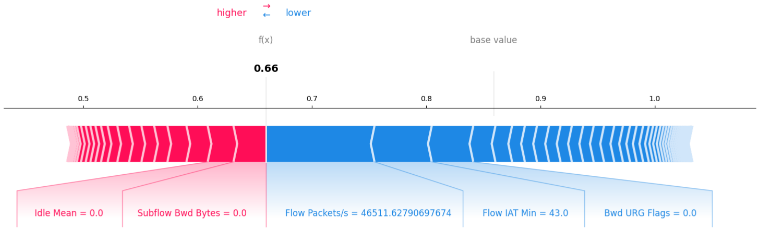

Figure 5.8: (Port Scan, 2.2) A SHAP force plot visualisation of feature importance for a Port Scan observation, Experiment 2.2

Mean (‘mean time a flow was idle before becoming active’, [1]) and Subflow Bwd Bytes for a Port Scan observation from Experiment 2.2. The most negatively impactful features were Bwd URG Flags, Flow IAT Min (’Minimum time between two packets sent in the flow’, [1]), and Flow Packets per second.

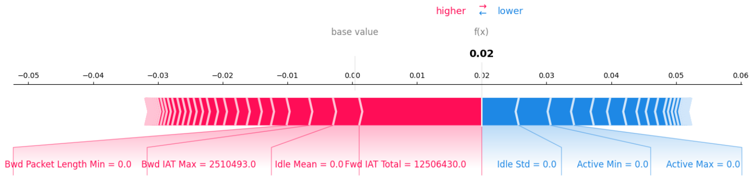

Figure 5.9: (SSH, 2.1) A SHAP force plot visualisation of feature importance for a SSH-Patator observation, Experiment 2.1

Figure 5.9 (SSH-Patator, 2.1) displays the most impactful features for the classifier’s incorrect prediction. The most positively impactful features were Fwd IAT Total (’total time between two packets sent in the forward direction’, [1]), Idle Mean, Bwd IAT Max (’Maximum

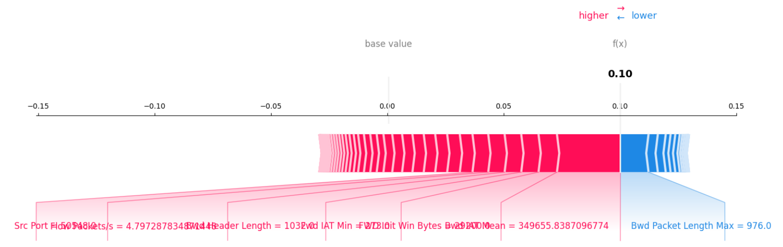

Figure 5.10: (SSH, 2.2) A SHAP force plot visualisation of feature importance for a SSH-Patator observation, Experiment 2.2

time between two packets sent in the backward direction', [1]), and Bwd Packet Length Min. The most negatively impactful features were Active Max ('maximum time a flow was active before becoming idle'), Active Min, and Idle Std.

Figure 5.10 (SSH-Patator, 2.2) displays the most impactful features for the classifier's incorrect prediction. The most positively impactful features were Source Port, Flow Packets per second, Bwd Header Length, Fwd IAT Min ('minimum time between two packets sent in the forward direction', [1]), Fwd Init Win Bytes ('total number of bytes sent in initial window in the forward direction', [1]), and Bwd IAT Mean. The most negatively impactful feature was Bwd Packet Length Max.

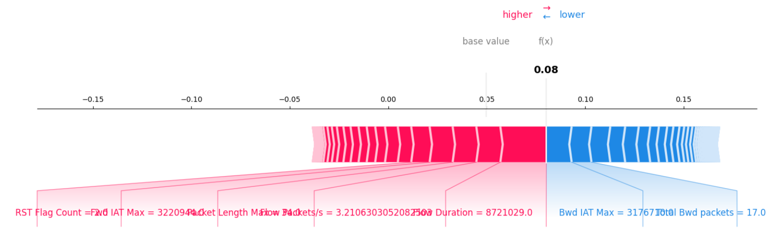

Figure 5.11: (FTP, 2.1) A SHAP force plot visualisation of feature importance for a FTP-Patator observation, Experiment 2.1

Finally, Figures 5.11 and 5.12 display the feature importance behind the incorrect predictions of FTP-Patator observations from both the 2.1 and 2.2 classifiers.

Figure 5.11 (FTP-Patator, 2.1) suggests five main positively impactful features on the classifier's prediction: Flow Duration, Flow Packets per second, Packet Length Max, Fwd IAT Max, and RST flag count. The two most negatively impactful features for the prediction were Total Bwd Packets and Bwd IAT Max.

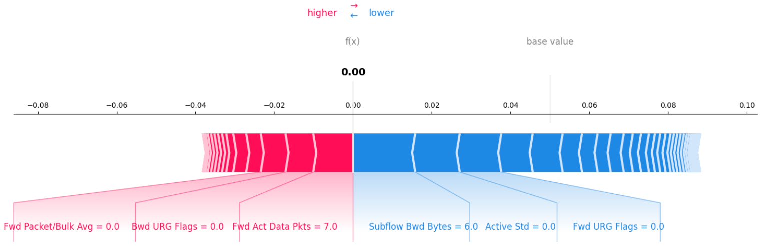

Figure 5.12: (FTP, 2.2) A SHAP force plot visualisation of feature importance for a FTP-Patator observation, Experiment 2.2

Figure 5.12 (FTP-Patator, 2.2) suggests that Fwd Packet/Bulk Avg, Bwd URG Flags, and Fwd Act Data Pkts were the most positively impactful features, whereas Subflow Bwd Bytes, Active Std, and Fwd URG Flags were the most negatively impactful.

5.4 Overview of Results

5.4.1 Exp. 1.1 - Results and Prediction Behaviour

In Experiment 1.1, the Random Forest classifier was trained and tested on the original CICIDS2017 dataset [32]. The classifier performs quite well statistically. However, the dataset on which it was trained was shown to have considerable issues regarding delineation between traffic flows [7], and this shows in some of the SHAP force plots generated. For instance, for a correctly-predicted Port Scan attack, relevant features such as 'Destination Port' and 'Fwd PSH Flags'[29] were not considered impactful. Instead, generally unrelated features such as Flow IAT mean, average packet size and flow duration were considered more impactful. Similarly, for the SSH-Patator observation, features that may not be related to the actual characteristics of the attack class were given great importance. These 'impactful' features include 'Fwd Packets per second', 'Flow Duration' and 'PSH Flags count'. One feature given greater importance for the FTP-Patator observation generally aligned with expectations – 'Fwd Packet Length Mean'.

These features, which do not have an obvious association with their observation labels, could wrongly indicate a false positive result to a network operator if they were to see this plot. Therefore, despite the high performance of the 1.1 classifier, high performance or high-trustworthiness of this classifier in a real network cannot be ensured at this time.

5.4.2 Exp. 1.2 - Engelen, et al. Results Reproducibility

In Experiment 1.2, the performance of the classifier in this research aligns with the performance of the classifier in the paper by Engelen, et al [7]. Both classifiers report high accuracy, precision, and recall. The 1.2 classifier does exhibit some potentially useful feature contributions. For instance, ‘ACK flag count’ and ‘Packet Length Min’ both are relevant features for correctly predicting port scan attacks [29]. This could be potentially explained by if Sharafaldin, et al. [32] executed ACK scanning attacks. Additionally, ‘Destination Port’ and ‘Total Fwd Packet’ are good potential indicators of SSH brute forcing attacks that were identified by the classifier [29].

Nevertheless, both classifiers also present potential shortcut learning. Both the Engelen classifier and the 1.2 classifier utilise the (generally) irrelevant ‘Total Fwd Packet’ and ‘Fwd Seg Size Min’ features to differentiate between different attack classes [7]. There were also some features deemed important by the 1.2 classifier, but the validity or spuriousness of their correlation to an attack class was not known. These features include the negatively-impactful ’Packet Length Mean’ and ’Fwd Bytes/Bulk Avg’ features for port scan attacks, the positively-impactful ’Idle Std’ and ’URG Flag Count’ features for SSH-Patator attacks, and the negatively-impactful ’Bwd IAT Total’ feature for FTP-Patator attacks.

Overall, there is an indication of some shortcut learning behaviours and some valid correlations. A network operator may find the feature importance of these observations to align more with their expectations, and thus might trust this classifier more than the Experiment 1.1 classifier. The performance and behaviours in Experiment 1.2 were also found to generally align with those of the classifier created and evaluated in the Engelen, et al. 2021 paper [7].

5.4.3 Exp 2.1 - Engelen Practical Viability

In Experiment 2.1, the withholding of ’novel’ port scan, SSH-Patator, and FTP-Patator attacks serves as an attempt to simulate how a commercial intrusion detector regularly encounters novel attack techniques. The classifier chosen for Experiment 2.1 is generated with the same hyperparameters as Experiment 1.2 and those used in the Engelen, et al. 2021 paper [7].

As can be immediately seen in Table 5.4, the 2.1 classifier was incapable of detecting any malicious attacks. Regarding the port scan observation (Figure 5.7), no features with obvious relevance to a Port Scan attack could be identified. Some immediately irrelevant features (e.g. Source Port) were wrongly identified by the classifier as impactful for this prediction. The features identified as impactful for both Patator observations appear similarly not relevant (Figures 5.9, 5.11). Based on the model performance and the non-relevant 'important' features, a network operator would be able to more immediately tell that this classifier was not trustworthy for a real network.

One hypothesis for this unsuccessful classifier is that these attacks are too similar to the benign traffic that was generated for the dataset. Sharafaldin et al. detail how activity from 25 network users was used by a Java-based agent to produce benign traffic 'based on HTTP, HTTPS, FTP, and SSH protocols' [32]. In particular, the bot-generated benign FTP and SSH traffic may have caused the classifier to identify all Patator traffic as benign.

5.4.4 Exp 2.2 - PyPABLO Results and Correlations

The PyPABLO classifier was similarly completely unsuccessful. The port scan and FTP-Patator observations did not contain any obviously relevant features. One important feature for the SSH-Patator observation, Fwd Init Win Bytes, was present in the list of relevant brute force features mentioned in the Background Chapter [29]. Unfortunately, this feature alone was not important enough for the classifier to predict the observation as being malicious.

Based on the lack of malicious traffic detected and the failure of the classifier to incorporate actually relevant features into its reasoning for these observations, a network operator would easily identify this classifier as being untrustworthy. It is also clear that cross-validation and optimising for known attacks did not improve the performance of the classifier on novel attacks. This is clear because there was no performance improvement between the 2.1 and 2.2 classifiers, despite one utilising cross-validation for model selection and one utilising pre-decided hyperparameters. Therefore, that aspect of the PABLO methodology should be re-evaluated and improved upon. Despite this, the use of SHAP is still clearly useful for understanding what locally 'went wrong' or was not 'trustworthy' for particular novel attack predictions. The soundness of the methodology for PABLO is therefore somewhat successful.

5.4.5 Contribution of SHAP Plots to Model Explainability

Although other aspects of the PABLO methodology were not successful for identifying novel attacks, SHAP force plots were a valuable method of determining the reasoning and potential information behind particular predictions. More insight was able to be gained on the reasoning behind a particular traffic flow and the potential vector of intrusion for that flow. For instance, if a network operator were alerted to the observation in Figure 5.2 as being a port scan attack, the feature importance on that plot could allow them to hypothesise that an ACK scan was the particular port scan attack being performed.

5.4.6 Potential Research Pitfalls

In this subsection, a list is displayed of potential research pitfalls, and the potential of those pitfalls in this research is discussed.

1. *Sampling Bias* occurs when training data does not represent the true data distribution of a particular problem, causing the results of a model trained on said data to become less trustworthy [4]. This pitfall is **present** in Experiments 1.1 and 1.2, due to the unrealistic data distribution in both the CICIDS2017 [32] and Engelen CICIDS2017 [7] datasets. This is not applicable in Experiments 2.1 and 2.2, because those experiments are intended to solely test the classifier on novel attacks.

2. *Label Inaccuracy* occurs when the ground truth label for a classification-based security system's training data is potentially inaccurate, or when the system cannot adapt to changes in adversary behaviour (label shift)[4]. Label inaccuracy **may** be present in these experiments, since there are multiple instances of inaccuracy being spotted in the CICIDS2017 dataset [7, 28]. This issue has been **mitigated** by the 'improved' dataset[7] in experiments 1.2-2.2.

3. *Data Snooping*[4]. Test snooping has been avoided, particularly in experiments 2.1 and 2.2, where certain attack classes are reserved solely for testing purposes. Training and testing data for other experiments was randomly selected, reducing the risk of selected snooping. These snooping variants are **not present**.

4. *Spurious Correlations* occur when a learning model makes false associations that correlate with their classification problem [4]. Explanation techniques were applied to all experiments to evaluate the reasoning behind each classifier and to ensure that performance was not inflated. This pitfall is **not present**.

5. *Biassed Parameter Selection* is an exceptional case of data snooping which occurs when final parameters of a model are indirectly dependent on its testing dataset [4]. This issue was **not present**; cross-validation for hyperparameter selection was utilised in experiment 2.2, and all prior hyperparameters for experiments were chosen in line with other research.

6. *Inappropriate Baselines* occur when a model is insufficiently evaluated in comparison to a variety of other models. This issue was **somewhat mitigated** by the presence

of four different experiments, with different levels of complexity. Non-machine learning approaches were not considered, which is a **present** aspect of this pitfall in this research [4].

7. *Inappropriate Performance Measures* occur when a lack of suitable performance measures are considered for a particular application scenario [4]. Appropriate performance measures were utilised, such as F1 score for considering class imbalance. Explainability was also utilised for classifier evaluation. This issue is **not present**.

8. *Base Rate Fallacy* occurs when results are misinterpreted due to class imbalance, e.g. if the negative class is predominant [4]. Despite class imbalance, this issue was **mitigated** by the use of precision, recall and F1 metrics when applicable.

9. *Lab-Only Evaluation* occurs when a machine learning model is not evaluated in a practical setting (i.e., it is evaluated only in a 'closed-world setting') [4]. This pitfall is **present but somewhat mitigated**. All classifiers were not evaluated in a practical setting, but the use of 'novel attack splits' were somewhat effective at simulating novel attacks on an anomaly detector.

10. *Inappropriate Threat Models* do not properly consider the hostility of a production environment, such as the influence of adversaries on real world learning-based systems through adversarial preprocessing, poisoning, and evasion[4]. Poisoning and other adversarial methods were not considered in this research; this pitfall is **present**.

Chapter 6

Legal, Social, Ethical & Professional Issues

This chapter details the impact and implications of this project on legal, ethical, social and professional issues, as well as the adherence of this project and researcher to the British Computing Society's Code of Conduct.

6.1 Legal Issues

The machine learning models in this research were trained on the CICIDS2017 and 'improved' CICIDS2017 datasets by Sharafaldin, et al. [32] and Engelen, et al. [7]. These are publicly available datasets created using the traffic of a research-setting network, ensuring that no individual's private information has been revealed.

In addition, any third-party ideas or libraries utilised in the implementation of PyPABLO have been explicitly attributed in both this report and the source code of PyPABLO. Any usage of third-party code for this project has strictly adhered to their licensing and attribution requirements for academic projects.

6.2 Ethical and Social Issues

This project details multiple network penetration techniques which may be used for malicious purposes, such as targeting specific groups or organisations, causing legal, ethical, social or financial issues in society. In this research, the motive for understanding these attacks is to craft more effective learning-based network intrusion detection systems.

One potential cause for ethical or legal concern is that an attacker could attempt to utilise learning-based detection research for their own malicious purposes. For instance, a creative attacker could simulate potential attacks and utilise learning-based models like PyPABLO to evaluate if their attacks will be detected by the model. This is an understandable concern which must be taken into account when undertaking intrusion detection research; nevertheless, this research must be done so that networks can be defended more successfully against existing attackers.

Although all machine learning-based models may contain biases or incorrect associations due to its creators or its training data, great care has been taken by Sharafaldin, et al. [32, 33], Engelen, et al. [7], and this researcher to eliminate any spurious correlations or identifying data in the datasets used for PyPABLO, thus mitigating the risk of specific groups or organisations being unfairly labelled or profiled as malicious by PyPABLO.

6.3 Professional Issues

This dissertation makes no claims to being production-ready as an intrusion detection system, and it is not recommended that any companies or organisations utilise PyPABLO in its current state. It is unlikely that the publication of this code in an academic setting will cause security issues for any company currently using a learning-based anomaly detection or intrusion detection system.

6.4 British Computing Society Code of Conduct

This project has adhered to the British Computing Society Code of Conduct [34] at all times. Particular attention was paid to standards 2a-e, which relate to professional integrity and a truthful representation of this researcher's competence, by explicitly citing any third-party code or inspiration that had any impact on this project. As per standards 2b and 3e, the progress and competence of this researcher was truthfully related to all relevant parties at all times.

In addition, the motivation for the project–to improve the usefulness and commercial viability of learning-based intrusion detection systems–is complementary to standards 1a and 1d of the Code of Conduct, which state "You shall have due regard for public health, privacy, security and wellbeing of others and the environment... [and] promote equal access to the benefits of IT and seek to promote the inclusion of all sectors in society wherever opportunities arise."

Chapter 7

Conclusion & Future Work

Ultimately, the four experiments in this research are a small introduction into the complex and constantly-evolving domain of intrusion detection. The PABLO methodology was not successful at detecting novel attacks, nor was it more successful than other approaches[7] that did not attempt to optimise for novel attacks. Nevertheless, the use of locally-explainable SHAP force plots was effective for better understanding of particular observations. Therefore, three main conclusions can be made:

1. SHAP plots act as an elegant and useful method for producing explainable results from a particular model observation.

2. SHAP visualisations can contribute to narrowing the ’semantic gap’ when models have the ’correct’ reasoning behind an observation.

3. Optimising for known attacks is not a viable methodology for attempting to detect novel attacks.

As long as there are adversaries, the domain of intrusion detection will never be definitively finished. The following list describes some potential areas for future research:

- *Efficient PySpark and SHAP compatibility.* The multi-core SHAP-PySpark approach discussed in this research could be modified for compatibility with SHAP visualisations. This could result in visualisations that are more efficient, and thus more viable for real-time intrusion detectors. This area is particularly intriguing for creating commercial learning-based anomaly detectors.

- *Novel attack detection.* This research was unsuccessful at detecting novel attacks. More research should specifically be aimed at determining quantifiable differences between malicious and benign port scanning and brute forcing attacks, due to the subtlety and unobtrusiveness of these attacks.

- *Optimisation of the 'improved' CICIDS2017 dataset*[7]. As Engelen, et al. note themselves, the 'improved' CICIDS2017 dataset still suffers from some pitfalls, notably label inaccuracy [7]. Nevertheless, the dataset is still quite useful, not least due to its size and scope. Further improvements and eradication of dataset pitfalls should be explored to make this dataset even more useful.
- *Exploration of the PABLO methodology on other classifiers.* The scope of this research was limited to evaluation of Random Forest Classifiers. The validity of the PABLO methodology should be tested on higher-performing and less-explainable models. This could determine whether optimisation for known attacks could improve the performance of more high-performance classifiers, or if real-time explainability via SHAP visualisations is not possible for less-explainable classifiers.

References

[1] ahlashkari. CICFlowMeter ReadMe.txt. GitHub repository, 2021. Accessed: 2024-04-07.

[2] Ahmed Ahmim, Leandros Maglaras, Mohamed Amine Ferrag, Makhlouf Derdour, and Helge Janicke. A novel hierarchical intrusion detection system based on decision tree and rules-based models. In *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 228 – 233. IEEE, 2019.

[3] Qasem Abu Al-Haija, Eyad Saleh, and Mohammad Alnabhan. Detecting port scan attacks using logistic regression. In *2021 4th International Symposium on Advanced Electrical and Communication Technologies (ISAECT)*, pages 1–5. IEEE, 2021.

[4] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallarok, and Konrad Rieck. Dos and don'ts of machine learning in computer security. In *31st Usenix Security Symposium*. Usenix, 2022.

[5] CrowdStrike. 2023 global threat report, 2023.

[6] Sepideh Ebrahimi and P. Patel. Scaling shap calculations with pyspark and pandas udf, 2022.

[7] Gints Engelen, Vera Rimmer, and Wouter Joosen. Troubleshooting an intrusion detection dataset: the cicids2017 case study. In *2021 IEEE Symposium on Security and Privacy Workshops (SPW)*, pages 7–12. IEEE, 2021.

[8] Mohamed Amine Ferrag, Leandros Maglaras, Sotiris Moschoyiannis, and Helge Janicke. Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. *Journal of Information Security and Applications*, 50, 2020.

[9] Gil Fidel, Ron Bitton, and Asaf Shabtai. When explainability meets adversarial learning: Detecting adversarial examples using shap signatures. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2020.

[10] Bimal Ghimire and Danda B. Rawat. Recent advances on federated learning for cybersecurity and cybersecurity for federated learning for internet of things. *IEEE Internet of Things Journal*, 9(11):8229 – 8249, 2022.

[11] Lukas-Valentin Herm, Kai Heinrich, Jonas Wanner, and Christian Janiesch. Stop ordering machine learning algorithms by their explainability! a user-centered investigation of performance and explainability. *International Journal of Information Management*, 69:102538, 2023.

[12] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 2. IEEE, 1995.

[13] Rui-Fong Hong, Shih-Cheng Horng, and Shieh-Shing Lin. Machine learning in cyber security analytics using nsl-kdd dataset. In *2021 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 260 – 265. IEEE, 2021.

[14] Zakaria Abou El Houda, Bouziane Brik, and Sidi-Mohammed Senouci. A novel iot-based explainable deep learning framework for intrusion detection systems. *IEEE Internet of Things Magazine*, 5(2):20–23, June 2022.

[15] IBM. What is an intrusion detection system (ids)? — ibm, 2023.

[16] Marwa Keshk, Nickolaos Koroniotis, N. Pham, N. Moustafa, B. Turnbull, and A. Y. Zomaya. An explainable deep learning-enabled intrusion detection framework in iot networks. *Information Sciences*, 639:119000, 2023.

[17] lanjelot. patator. GitHub repository, 2012. Accessed: 2023-12-10.

[18] Thi-Thu-Huong Le, Haeyoung Kim, Hyoeun Kang, and Howon Kim. Classification and explanation for intrusion detection system based on ensemble trees and shap method. *Sensors*, 22(3), 2022,.

[19] Cynthia Bailey Lee, Chris Roedel, and Elena Silenok. Detection and characterization of port scan attacks. *University of California, Department of Computer Science and Engineering*, 2003.

[20] Scott M. Lundberg. An introduction to explainable ai with shapley values — shap latest documentation, 2023.

[21] Scott M. Lundberg. shap.plots.force, 2023.

[22] Carlos Guestrin Marco Tulio Ribeiro, Sameer Singh. 'why should i trust you?': Explaining the predictions of any classifier. In *KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. KDD, 2016.

[23] We Are Social & Meltwater. Digital 2024 global overview report, 2024.

[24] Muhammad Nadeem, Ali Arshad, Saman Riaz, Shahab S. Band, and Amir Mosavi. Intercept the cloud network from brute force and ddos attacks via intrusion detection and prevention system. *IEEE Access*, 9:152300–152309, 2021.

[25] Maryam M. Najafabadi, Taghi M. Khoshgoftaar, Clifford Kemp, Naeem Seliya, and Richard Zuech. Machine learning for detecting brute force attacks at the network level. In *2014 IEEE International Conference on Bioinformatics and Bioengineering*, pages 379–385. IEEE, 2014.

[26] Ayodeji Oseni, Nour Moustafa, Gideon Creech, Nasrin Sohrabi, Andrew Strelzoff, Zahir Tari, and Igor Linkov. An explainable deep learning framework for resilient intrusion detection in iot-enabled transportation networks. *IEEE Transactions on Intelligent Transportation Systems*, 24(1):1000 – 1014, 2023.

[27] Philipp Probst, Marvin N. Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: data mining and knowledge discovery*, 9(3):e1301, 2019.

[28] Samarjeet Borah Ranjit Panigrahi. A detailed analysis of cicids2017 dataset for designing intrusion detection systems. *International Journal of Engineering & Technology*, 7:479 – 482, 2018.

[29] María Rodríguez, Álvaro Alesanco, Lorena Mehavilla, and José García. Evaluation of machine learning techniques for traffic flow-based intrusion detection. *Sensors*, 22(23), 2022.

[30] Su-In Lee Scott M. Lundberg. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS, 2017.

[31] Lloyd Stowell Shapley. *A value for n-person games*, volume 2 of *Annals of Mathematics Studies*, chapter 17, pages 307–317. Princeton University Press, 1953.

[32] Iman Sharafaldin, Amirhossein Gharib, Arash Habibi Lashkari, and Ali A. Ghorbani. Towards a reliable intrusion detection benchmark dataset. *Journal of Software Networking*, pages 177–200, 2017.

[33] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proc. of the 4th International Conference on Information Systems Security and Privacy*, pages 108 – 116. ICISSP, 2018.

[34] British Computing Society. British computing society code of conduct, 2021.

[35] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316. IEEE, 2010.

[36] Apache Spark. Paramgridbuilder – pyspark, 2024. version 3.1.3.

[37] Apache Spark. Randomforestclassifier – pyspark, 2024. version 3.1.3.

[38] Maonan Wang, Kangfeng Zheng, Yanqing Yang, and Xiujuan Wang. An explainable machine learning framework for intrusion detection systems. *IEEE Access*, 8:73127 – 73141, 2020.

[39] Yun Wang, Pan Wang, ZiXuan Wang, and Mengting Cao. An explainable intrusion detection system. In *2021 IEEE 23rd Int Conf on High Performance Computing & Communications*, pages 1657 – 1662. IEEE, 2021.