POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

AI-based anomaly detection on RISC-V

Supervisors

Prof. DI CARLO STEFANO

Prof. SAVINO ALESSANDRO

Dr. PEGORARO CHENET CRISTIANO

Candidate

Zhang Ziteng

March 27, 2024

Contents

1 Introduction 5

1.1 Background and Significance . . . 5

1.1.1 Importance of AI-based Anomaly Detection in RISC-V Architecture . . . 7

1.1.2 Current Gaps in AI-based Anomaly Detection on RISC-V . 8

1.2 Objectives and research questions . . . 8

1.2.1 Develop Tailored AI-based Anomaly Detection Models . . . 9

1.2.2 Overcome Data Scarcity for Model Training . . . 9

1.3 Structure Overview . . . 10

2 Literature Review 12

2.1 Introduction to Anomaly Detection . . . 12

2.2 Evolution of Anomaly Detection Techniques . . . 13

2.3 Limitations of Traditional Security Detection Methods . . . 15

2.4 Advantages of AI-based Approaches for RISC-V Security . . . 16

2.5 Comparison of Anomaly Detection Techniques . . . 17

2.6 AI-based Anomaly Detection Methods . . . 19

2.6.1 Classification algorithm . . . 20

2.6.2 Feature Selection Algorithm . . . 27

2.6.3 Principal component analysis Algorithm . . . 29

2.6.4 Evaluation of Algorithmic Models . . . 29

2.7 Platform . . . 32

2.7.1 RISC-V: PULP . . . 32

2.7.2 SOC environment: GVSOC . . . 34

3 Experimental Design **35**

3.1 Platform choices . . . 36

3.2 Benign and Malicious application design . . . 36

3.3 Scripts design . . . 37

3.3.1 Data Collection . . . 37

3.3.2 Data Preprocessing . . . 37

3.4 AI program design . . . 38

3.5 Experimental procedures . . . 40

4 Experimental Results **44**

4.1 Experiment 1: Classification . . . 45

4.1.1 Classification with Program.1 . . . 45

4.1.2 Controlled experiment: Classification with program.2 . . . 51

4.2 Experiment 2: Feature selection . . . 54

4.2.1 Selection of experimental subjects . . . 55

4.2.2 Experiment procedure . . . 56

4.3 Experiment 3: Feature ranking . . . 59

5 Conclusion and Future Work **62**

5.1 Research Contributions and Limitations . . . 62

5.2 Areas for Improvement and Future Work . . . 62

6 Appendix **68**

List of Abbreviations

AES Advanced Encryption Standard

DDoS distributed denial-of-service

DL Deep Learning

DTs DecisionTree

FPGA Field Programmable Gate Arrays

GaussianNB Gaussian Naive Bayes

HGBT Histogram-Based Gradient Boosting

IoT Internet of Things

LDA Linear Discriminant Analysis

MCUs micro-controller units

ML Machine Learning

PCA principal component analysis

PMCs Performance Monitoring Counters

QDA Quadratic Discriminant Analysis

RISC-V reduced instruction set computer

SDK software development kit

SoCs systems-on-chips

SVC C-Support Vector Classification

SVM support vector machines

1 Introduction

1.1 Background and Significance

This thesis delves into the development of a robust profiling infrastructure, leveraging the cutting-edge specifications of the RISC-V hardware performance counters, to be implemented on a carefully selected RISC-V core. The primary objective is to seamlessly integrate Performance Monitoring Counters (PMCs in following) into the architecture. These PMCs furnish valuable aggregate data, such as the count of memory transactions, which proves indispensable for real-time application profiling, all while incurring minimal overhead.

The proposed profiling infrastructure brings forth a multitude of advantages, enabling in-depth analysis of software behavior and facilitating the identification of potentially malicious alterations. However, as software systems continue to evolve in complexity, conventional approaches to identifying harmful software exhibit inherent limitations that hamper their effectiveness. This disparity between the intricacy of modern software and the constraints of traditional verification and validation tools necessitates a paradigm shift in our methodology.

Enter the realm of Artificial Intelligence (AI). This research pivots towards the integration of AI techniques to empower software developers in their pursuit of anomaly detection. The wealth of data that can be harnessed through the utilization of PMCs offers an expansive arena for the training and refinement of specialized AI-driven anomaly detection models [6]. These models hold the potential to discern not only software glitches but also security vulnerabilities, rendering them a potent ally in the battle against harmful software behaviors.

Shifting our focus towards AI-driven prediction and anomaly detection entails a multitude of benefits over conventional methods:

1. Enhanced Precision and Recall: AI algorithms, particularly machine learning and deep learning models, possess the capability to identify intricate patterns and correlations within vast datasets. This empowers them to achieve higher precision and recall rates in pinpointing potential harmful behaviors, outclassing the relatively simplistic rule-based approaches of traditional tools.

2. Adaptability to Emerging Threats: The landscape of software threats is continually evolving. AI-powered systems can swiftly adapt to new and previously unseen threats by learning from historical data and recognizing novel attack vectors, a feat that conventional methods struggle to accomplish.

3. Reduced False Positives: Traditional methods often generate a substantial number of false positive alerts, leading to time-consuming and resource-intensive investigations. AI-driven models, with their nuanced understanding of software behavior, can significantly reduce the occurrence of false positives, enabling developers to focus their efforts on genuine threats.

4. Real-time Detection: AI-powered systems can operate in near real-time, continuously analyzing software behavior and promptly flagging any aberrations. This agility is crucial in swiftly identifying and mitigating potential harmful actions.

5. Scalability: With the exponential growth in software complexity and the sheer volume of applications, an AI-driven approach provides scalability, as these models can be trained to handle diverse software types and scales efficiently.

6. Reduced Human Effort: By automating the detection process, AI models can significantly alleviate the manual labor involved in sifting through extensive logs and reports, enabling developers to focus on higher-level tasks.

Incorporating AI into the realm of harmful software behavior prediction not only addresses the limitations of conventional methods but also embraces the opportunities presented by the ever-expanding landscape of software intricacy and threats.Therefore, one where AI-driven anomaly detection becomes an integral and indispensable component in the quest for secure and robust software systems.

1.1.1 Importance of AI-based Anomaly Detection in RISC-V Architecture

1. Rising Security Concerns: With the increasing adoption of RISC-V in critical applications, from IoT devices to data centers, security becomes paramount. AI-based anomaly detection offers a proactive approach to identifying and mitigating potential threats before they can cause harm.

2. Flexibility and Customization: RISC-V's open-source and extensible nature allows for tailored implementations. However, this flexibility also introduces unique vulnerabilities. AI-based systems can adapt to these custom configurations, offering bespoke security solutions.

3. Efficiency in Detection: Traditional security measures may not suffice for the lightweight, resource-constrained environments where RISC-V operates. AI techniques, with their ability to learn from data, can efficiently detect anomalies even in such restricted settings.

4. Scalability: As RISC-V-based systems scale, manually monitoring for security breaches becomes impractical. AI-based detection systems can scale alongside these architectures, maintaining security without compromising performance.

1.1.2 Current Gaps in AI-based Anomaly Detection on RISC-V

1. Lack of Tailored Solutions: While AI-based anomaly detection has been explored in various contexts, there's a scarcity of research specifically tailored to the RISC-V architecture. This gap leaves potential security vulnerabilities unaddressed.

2. Data Scarcity: Effective AI models require vast amounts of data. For RISC-V, the specific data on anomalies is scarce, making it challenging to train robust detection models.

3. Complexity of Integration: Integrating AI-based detection systems into RISC-V architectures presents technical challenges, from resource constraints to compatibility with the open-source ecosystem.

4. Evolving Threat Landscape: The rapidly evolving nature of cybersecurity threats means that AI models must continuously learn and adapt. Currently, there's a lack of mechanisms to update these models in real-time as new threats emerge.

5. Underexplored Potential of AI Techniques: While some AI techniques have been applied to anomaly detection, the full spectrum, including deep learning, reinforcement learning, and unsupervised learning, remains underexplored in the context of RISC-V.

1.2 Objectives and research questions

This thesis aims at creating a powerful profiling infrastructure based on the latest specifications of the RISC-V HPM on a selected RISC-V core in order to provide PMCs in the architecture. This thesis focuses on applying AI to support software developers in anomaly detection activities. The huge amount of data that can be

collected resorting to PMCs is an important playground to train specific AI-based anomaly detection models able to identify both software bugs and security threats.

1.2.1 Develop Tailored AI-based Anomaly Detection Models

- Specific: Design and implement AI models specifically tailored for anomaly detection in RISC-V architectures, considering their unique security requirements and operational environments.
- Measurable: Evaluate the models' effectiveness in detecting a comprehensive range of anomalies, using metrics such as detection accuracy, false positive rate, and detection speed.
- Achievable: Utilize existing AI frameworks and RISC-V simulation environments to develop and test the models.
- Relevant: This objective addresses the current gap in tailored anomaly detection solutions for RISC-V, enhancing security in critical applications.

1.2.2 Overcome Data Scarcity for Model Training

- Specific: Create a synthetic dataset that accurately reflects the operational conditions and potential anomalies within RISC-V architectures.
- Measurable: The dataset should be large enough to train AI models effectively, with more than one kind of anomaly scenarios represented.
- Achievable: Collaborate with RISC-V developers and use generative AI techniques to produce the dataset.
- Relevant: Solving the data scarcity problem is crucial for developing robust AI-based detection systems.

1.3 Structure Overview

This thesis is organized into six main chapters, each designed to systematically explore AI-based anomaly detection within RISC-V architecture, from foundational theories to practical applications and future directions. Here's what readers can anticipate in each chapter:

Chapter 1, Introduction: Sets the stage by introducing the research context, defining the problem statement, articulating the research objectives, and highlighting the contributions of this study to the field of AI-based anomaly detection in RISC-V systems.

Chapter 2, Literature Review: Provides a comprehensive review of the existing body of knowledge, tracing the evolution of anomaly detection techniques and the advent of AI in this field. It critically analyzes current research, identifying gaps that this thesis aims to address, particularly within RISC-V architecture.

Chapter 3, Experimental Design: Details the research methodology, including the design, data collection, and processing approaches adopted. It elaborates on the development and validation of the AI-based anomaly detection model, explaining the selection of algorithms and evaluation criteria.

Chapter 4, Implementation and Results: Describes the practical implementation of the anomaly detection system, including system design and technical challenges. It presents the experimental setup, followed by a thorough analysis of the results obtained, benchmarking these against the research objectives.

Chapter 5, Conclusion and Future Work: Concludes the thesis by summarizing the key findings and their significance. It provides a reflective commentary on the research journey, offering recommendations for future research and suggesting potential pathways for further exploration in the domain of AI-based anomaly detection in RISC-V architecture.

This structured approach ensures a comprehensive exploration of the subject matter, facilitating a deep understanding of both the challenges and innovations in AI-based anomaly detection for RISC-V systems. Through this journey, the thesis aims to contribute valuable insights and methodologies to the field, encouraging further research and development in enhancing the security and efficiency of RISC-V architectures.

2 Literature Review

2.1 Introduction to Anomaly Detection

In the realm of computer security, anomalies represent deviations from established patterns or behaviors within a system. Malware, encompassing viruses, worms, Trojans, and other malicious software [5] [3] , inherently introduces anomalies by exhibiting behavior that contrasts with the normal operations of a computing environment. These deviations can range from unauthorized access attempts to unusual data access patterns or unexpected system modifications.

- Viruses: Programs that replicate themselves and infect other files or systems by attaching to legitimate programs. Viruses often cause damage by corrupting or deleting files.
- Worms: Self-replicating malware that spreads across networks without user intervention. Worms exploit vulnerabilities to replicate and spread rapidly, consuming network bandwidth and causing system slowdowns.
- Trojans: Malware disguised as legitimate software, tricking users into installing them. Unlike viruses and worms, Trojans do not self-replicate but can create backdoors for hackers, steal data, or perform other malicious actions.
- Ransomware: Encrypts files on a victim's system and demands payment (ransom) to decrypt them. Ransomware can severely disrupt operations for individuals, organizations, or even entire networks.
- Spyware: Designed to spy on users' activities without their knowledge. It collects sensitive information such as keystrokes, browsing habits, or login credentials, compromising user privacy.

- Adware: Displays unwanted advertisements, often in the form of pop-ups or banners, to generate revenue for the malware creators. While not as destructive, adware can be annoying and negatively impact system performance.
- Rootkits: Conceals malicious software within a system, allowing unauthorized access and control while evading detection by antivirus programs. Rootkits can give attackers persistent access to compromised systems.
- Botnets: Networks of compromised computers (bots) controlled by a central server or command center. Botnets can be used for various malicious activities like launching DDoS attacks, sending spam, or stealing data collectively.
- Fileless Malware: Operates in a system's memory without leaving traces on the hard drive. This type of malware is harder to detect by traditional antivirus software as it doesn't rely on files.

2.2 Evolution of Anomaly Detection Techniques

The prevalence and evolving sophistication of malware underscore the critical necessity for effective anomaly detection mechanisms. Traditional security measures, while robust, often struggle to keep pace with the rapidly mutating landscape of malicious software. As malware continuously evolves, adopting camouflage techniques and obfuscation methods, the need for adaptive and intelligent anomaly detection systems becomes paramount.

1. Early Techniques Statistical Methods: Begin with the earliest approaches to anomaly detection, which were predominantly statistical. These methods relied on statistical models to identify outliers based on deviations from expected patterns. Highlight key methods such as Z-score, statistical process

control, and Bayesian networks. Limitations: Discuss the limitations of these early methods, including their reliance on assumptions about data distribution and their difficulty in handling high-dimensional data or dynamically changing environments.

2. Machine Learning Era Supervised Learning: With the advent of machine learning, supervised learning techniques started being applied to anomaly detection. Methods like support vector machines (SVM), decision trees, and neural networks were adapted to identify anomalies by learning from labeled datasets. [6] Unsupervised Learning: Unsupervised learning methods gained popularity due to the scarcity of labeled anomaly data. Algorithms such as k-means clustering, autoencoders, and principal component analysis (PCA) were used to detect anomalies based on data patterns without requiring labeled examples. [15] Semi-Supervised and Hybrid Methods: Discuss the emergence of semi-supervised and hybrid methods that combine supervised and unsupervised learning to improve detection accuracy, especially in situations with limited labeled data.

3. Integration of Big Data and IoT Big Data Analytics: With the explosion of big data, anomaly detection techniques have evolved to process and analyze vast amounts of information in real-time, using big data analytics tools and distributed computing frameworks like Apache Hadoop and Apache Spark. IoT and Edge Computing: Discuss the role of anomaly detection in the Internet of Things (IoT) and edge computing, where detecting anomalies at the source of data generation is crucial for timely responses to potential threats or failures.

4. AI and Anomaly Detection Today AI-driven Approaches: In the current landscape, AI-driven approaches are at the forefront of anomaly detection.

These include the use of machine learning, deep learning, and data science techniques to automatically identify and respond to anomalies in diverse domains.

2.3 Limitations of Traditional Security Detection Methods

Limitations of Traditional Security Methods in the Context of RISC-V Traditional security methods for computing architectures, including RISC-V, often rely on predefined rules, signatures, or patterns to identify threats. These methods, while effective against known threats, exhibit several limitations in a rapidly evolving threat landscape:

1. Static Nature: Traditional methods are inherently static, requiring regular updates to their databases to recognize new threats. This approach is less effective against zero-day exploits or sophisticated attacks that have not been previously encountered.

2. Scalability Issues: As RISC-V architectures are adopted across various domains, from IoT devices to enterprise systems, the scalability of traditional security solutions becomes a challenge. They may not efficiently handle the vast, diverse datasets generated by these systems without significant resource investment.

3. Limited Contextual Analysis: Traditional security mechanisms often lack the ability to perform deep contextual analysis. They might identify anomalies based on deviation from predefined patterns but fail to understand the context behind these deviations, leading to high false positive rates.

4. Customization Challenges: Given the open-source nature of RISC-V, systems built on this architecture can vary widely. Traditional security solu-

tions, which are generally designed for broad application, may not be easily customizable to the specific needs of a particular RISC-V implementation.

2.4 Advantages of AI-based Approaches for RISC-V Security

AI-based anomaly detection methods offer several advantages over traditional security solutions, especially in the context of the adaptable and open-source RISC-V architecture [8]:

1. Dynamic Learning: AI models, particularly those based on machine learning and deep learning, can continuously learn from new data. This capability allows them to adapt to new threats over time without requiring manual updates, making them highly effective against unknown or evolving attacks.

2. Scalability and Efficiency: AI algorithms can analyze vast quantities of data in real-time, making them highly scalable and efficient for use in diverse RISC-V-based applications, from small IoT devices to large-scale computing systems.

3. Contextual Awareness: Through advanced data analysis techniques, AI-based methods can understand the context of data and user behavior. This understanding allows for more accurate detection of genuine anomalies, reducing the rate of false positives and improving the overall security posture.

4. Customization: AI models can be trained on specific datasets relevant to a particular RISC-V implementation. This training enables the models to detect anomalies that are unique to that system's normal operation, offering a tailored security solution that traditional methods cannot provide.

5. Proactive Threat Detection: AI can identify subtle patterns and correlations that may indicate the early stages of a complex attack, enabling proactive threat detection and mitigation before significant damage occurs.

In conclusion, while traditional security methods have played a crucial role in protecting computing systems, the unique characteristics of RISC-V architectures demand more adaptable, efficient, and intelligent security solutions. AI-based anomaly detection represents a promising approach to meeting these demands, offering the potential to significantly enhance the security of RISC-V systems in an ever-evolving threat landscape.

2.5 Comparison of Anomaly Detection Techniques

Traditional Methods:

1. Statistical Methods: Rely on mathematical models to define normal behavior and detect outliers. While effective for static datasets with well-defined patterns, they struggle with the high variability and evolving nature of cybersecurity threats.

2. Threshold-based Techniques: Set fixed values for metrics, above or below which data is considered anomalous. This simplicity can be a drawback in complex systems where anomalies are not easily defined by thresholds. An important work about threshold using in hardware-based malware detection is the work of Malone et. al. [12].

3. Signature-based Detection: Uses known patterns of malicious activity to identify threats. Its major limitation is the inability to detect zero-day attacks or novel anomalies not already in its database.

AI-based Methods:

1. Machine Learning (ML): Utilizes algorithms that learn from data to identify patterns and anomalies. Unlike static methods, ML can adapt to new data, making it highly effective against evolving threats.

   Supervised Learning: Requires labeled datasets to train models, excelling in environments where historical attack data is available.

   Unsupervised Learning: Detects anomalies by learning the normal distribution of data without needing labels, ideal for detecting unknown or zero-day attacks.

2. Deep Learning (DL): A subset of ML that uses neural networks with multiple layers. It's particularly adept at processing large volumes of data and capturing complex patterns that traditional methods might miss. Advantages in Dynamic Landscapes: DL models can automatically feature engineer, extracting relevant features from raw data, which is crucial in cybersecurity where threat indicators can be subtle and highly nuanced.

Hybrid Approaches:

Combining ML and Rule-based Systems: Some systems integrate ML with traditional rule-based methods to leverage the strengths of both. This hybrid approach can balance adaptability with the reliability of established rules,AI-based anomaly detection offers several distinct advantages in handling dynamic, complex threat landscapes.

AI's Advantages:

- Adaptability: AI models continuously learn and adapt to new data, making them more resilient to the evolving nature of cyber threats.
- Scalability: They can efficiently process and analyze vast amounts of data from various sources, essential for monitoring extensive networks like those based on RISC-V.

- Sensitivity to Subtle Anomalies: AI techniques, especially DL, can detect anomalies that might be too subtle for traditional detection methods, offering a deeper layer of security.

- Automation and Efficiency: AI can automate the detection process, reducing the need for manual intervention and enabling quicker responses to potential threats.

In summary, while traditional anomaly detection methods have their place, AI-based techniques stand out for their ability to adapt, scale, and sensitively respond to the complex and dynamic nature of modern cybersecurity threats. This adaptability is particularly critical for securing RISC-V architectures, where the open-source nature and rapid innovation pace require robust, flexible security solutions.

2.6 AI-based Anomaly Detection Methods

Anomaly detection techniques encompass a diverse array of methodologies, including statistical analysis, machine learning, and heuristic approaches. Statistical methods involve establishing patterns within data and flagging deviations that fall outside predefined thresholds. Machine learning, on the other hand, leverages algorithms trained on datasets to discern normal behavior from anomalies, adapting to evolving threats. Heuristic methods rely on predefined rules or models to identify anomalies based on specific patterns or signatures associated with known malware.

The fusion of artificial intelligence (AI) with anomaly detection has exhibited promising results, enabling systems to learn and adapt in real-time, thereby enhancing the detection and mitigation of sophisticated, previously unseen malware.

This thesis delves into the exploration and evaluation of AI-based anomaly detection methodologies specifically tailored for identifying and mitigating anomalies,

particularly within the context of RISC-V architecture.

This section sets the stage by introducing the concept of anomalies, emphasizing the significance of detecting anomalies in the context of malware, and providing a brief overview of the methodologies used for anomaly detection, paving the way for the focus on AI-based anomaly detection within RISC-V architecture.

2.6.1 Classification algorithm

In sci-kit there's a comparison of several classifiers. We selected part of them for

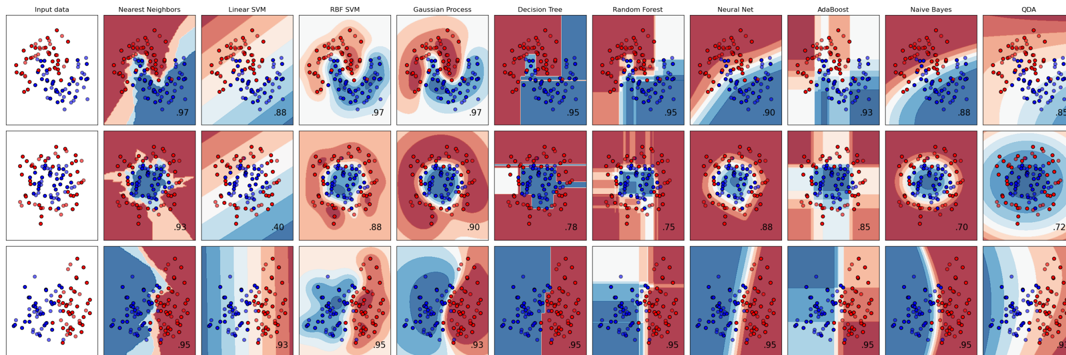


Figure 1: Comparison between various classification algorithms

our experiments.

- Logistic regression

  The logistic regression is implemented in LogisticRegression. Despite its name, it is implemented as a linear model for classification rather than regression in terms of the scikit-learn/ML nomenclature. The logistic regression is also known in the literature as logit regression, maximum-entropy classification (MaxEnt) or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logistic function.

This implementation can fit binary, One-vs-Rest, or multinomial logistic regression with optional , or Elastic-Net regularization. The binary case can be extended to K classes leading to the multinomial logistic regression. Let $y_i \in 1, ..., K$ be the label (ordinal) encoded target variable for observation i . Instead of a single coefficient vector, we now have a matrix of coefficients W where each row vector W_K corresponds to class K . We aim at predicting the class probabilities $P(y_i = K|X_i)$ via $Predict_proba$ as:

$$\hat{p}_k(X_i) = \frac{\exp(X_i W_k + W_{0,k})}{\sum_{l=0}^{K-1} \exp(X_i W_l + W_{0,l})} \tag{1}$$

The objective for the optimization becomes:

$$\min_W -C \sum_{i=1}^{n} \sum_{k=0}^{K-1} [y_i = k] \log(\hat{p}_k(X_i)) + r(W) \tag{2}$$

Where $[P]$ represents the Iverson bracket which evaluates to 0 if P is false, otherwise it evaluates to 1 We currently provide four choices for the regularization term $r(W)$ via the *penalty* argument, where is the number of features:

penalty	$r(W)$
None	0
ℓ_1	$\|W\|_{1,1} = \sum_{i=1}^{m} \sum_{j=1}^{K} \|W_{i,j}\|$
ℓ_2	$\frac{1}{2}\|W\|_F^2 = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{K} W_{i,j}^2$
ElasticNet	$\frac{1-\rho}{2}\|W\|_F^2 + \rho\|W\|_{1,1}$

Table 1: Regularization penalties and their corresponding terms

- Linear SVC

  SVC: C-Support Vector Classification.

  The implementation is based on libsvm [4]. The fit time scales at least quadratically with the number of samples and may be impractical beyond

tens of thousands of samples. Due to the large datasets, we consider using LinearSVC instead.

Linear SVC: Linear Support Vector Classification.

Similar to SVC with parameter kernel='linear', but implemented in terms of liblinear rather than libsvm, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

The main differences between LinearSVC and SVC lie in the loss function used by default, and in the handling of intercept regularization between those two implementations.

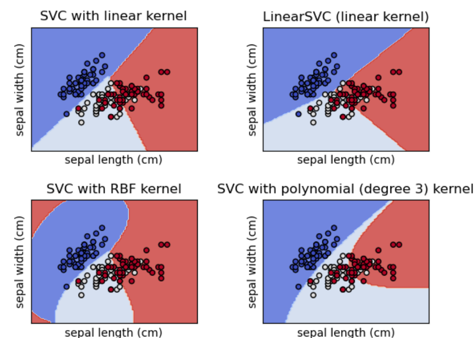


Figure 2: SVC and LinearSVC

- RandomForest

  In random forests, each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set.

  Furthermore, when splitting each node during the construction of a tree, the best split is found through an exhaustive search of the features values of either all input features or a random subset of size max_features.

  The purpose of these two sources of randomness is to decrease the variance of the forest estimator. Indeed, individual decision trees typically exhibit

high variance and tend to overfit. The injected randomness in forests yield decision trees with somewhat decoupled prediction errors. By taking an average of those predictions, some errors can cancel out. Random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias. In practice the variance reduction is often significant hence yielding an overall better model.

A competitive alternative to random forests are Histogram-Based Gradient Boosting (HGBT) models:

Building trees: Random forests typically rely on deep trees (that overfit individually) which uses much computational resources, as they require several splittings and evaluations of candidate splits. Boosting models build shallow trees (that underfit individually) which are faster to fit and predict.

Sequential boosting: In HGBT, the decision trees are built sequentially, where each tree is trained to correct the errors made by the previous ones. This allows them to iteratively improve the model's performance using relatively few trees. In contrast, random forests use a majority vote to predict the outcome, which can require a larger number of trees to achieve the same level of accuracy.

Efficient binning: HGBT uses an efficient binning algorithm that can handle large datasets with a high number of features. The binning algorithm can pre-process the data to speed up the subsequent tree construction (see Why it's faster). In contrast, the scikit-learn implementation of random forests does not use binning and relies on exact splitting, which can be computationally expensive.

Overall, the computational cost of HGBT versus RF depends on the specific characteristics of the dataset and the modeling task [11].

- DecisionTree

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation.

For instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

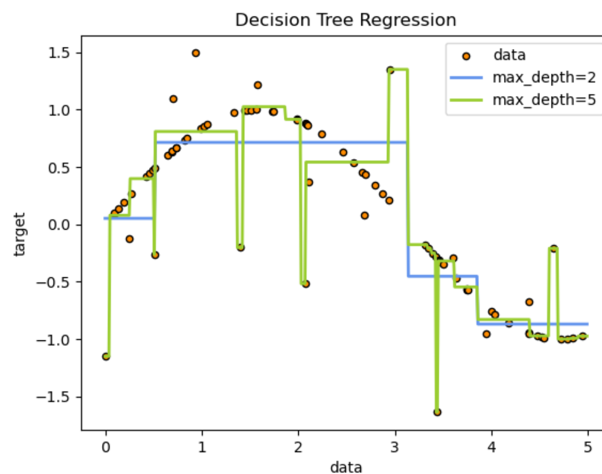


Figure 3: DecisionTree

Some advantages of decision trees are [13]:

Simple to understand and to interpret. Trees can be visualized.

Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed. Some tree and algorithm combinations support missing values.

The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.

Able to handle both numerical and categorical data.

Able to handle multi-output problems.

Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.

Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.

Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

Decision-tree learners can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.

Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.

Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.

The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are

made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.

There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.

Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

- GaussianNB

  GaussianNB implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{3}$$

  This equation represents the density function of a Gaussian distribution, where x_i is a variable, y could be a condition or class, mu_y is the mean of the distribution under condition y , and $sigma_y^2$ is the variance of the distribution under condition y.

  The parameters μ_y and σ_y are estimated using maximum likelihood.

- Quadratic Discriminant Analysis

  A classifier with a quadratic decision boundary, generated by fitting class conditional densities to the data and using Bayes' rule.

  The model fits a Gaussian density to each class.

  According to the model above, the log of the posterior is:

$$\log P(y=k|x) = \log P(x|y=k) + \log P(y=k) + C_{st}$$
$$= -\frac{1}{2}\log|\Sigma_k| - \frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) + \log P(y=k) + C_{st} \quad (4)$$

where the constant term C_{st} corresponds to the denominator $P(x)$, in addition to other constant terms from the Gaussian. The predicted class is the one that maximises this log-posterior.

Linear Discriminant Analysis (LinearDiscriminantAnalysis) and Quadratic Discriminant Analysis (QuadraticDiscriminantAnalysis) are two classic classifiers, with, as their names suggest, a linear and a quadratic decision surface, respectively.

The plot 4 shows decision boundaries for Linear Discriminant Analysis and Quadratic Discriminant Analysis. The bottom row demonstrates that Linear Discriminant Analysis can only learn linear boundaries, while Quadratic Discriminant Analysis can learn quadratic boundaries and is therefore more flexible.

2.6.2 Feature Selection Algorithm

Feature selection, also known as variable selection or attribute selection, involves automatically selecting those features in our data that contribute most to the prediction variable or output in which we are interested. Not only does this process help in improving the performance of a model, but it also helps in faster training times and better generalization by reducing overfitting. In this thesis we mainly use 2 algorithm:

- Univariate feature selection [13]

  Univariate feature selection works by selecting the best features based on univariate statistical tests. It can be seen as a preprocessing step to an estimator. SelectKBest: removes all but the highest scoring features

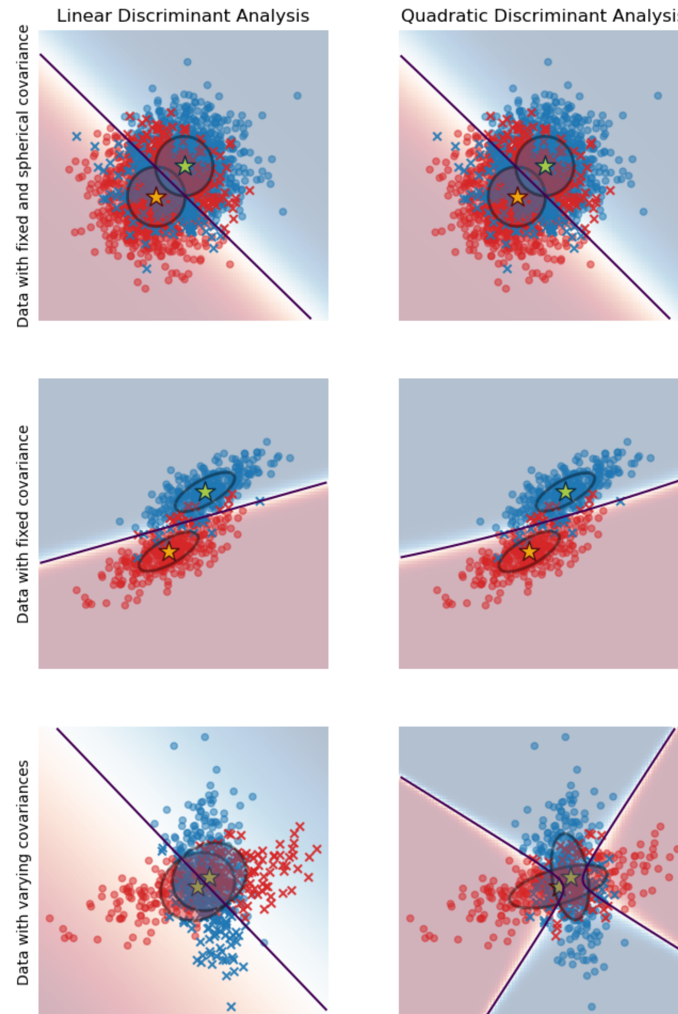


Figure 4: LDA and QDA

- Recursive feature elimination [13]

  Given an external estimator that assigns weights to features (e.g., the coefficients of a linear model), the goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through any specific attribute (such as coef_, feature_importances_) or callable. Then, the least important features are pruned from current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached.

2.6.3 Principal component analysis Algorithm

PCA [10] is used to decompose a multivariate dataset in a set of successive orthogonal components that explain a maximum amount of the variance. In scikit-learn, PCA is implemented as a transformer object that learns n components in its fit method, and can be used on new data to project it on these components.

PCA centers but does not scale the input data for each feature before applying the SVD. The optional parameter whiten=True makes it possible to project the data onto the singular space while scaling each component to unit variance. This is often useful if the models down-stream make strong assumptions on the isotropy of the signal: this is for example the case for Support Vector Machines with the RBF kernel and the K-Means clustering algorithm.

2.6.4 Evaluation of Algorithmic Models

We analyze the results by these coefficient:

1. Confusion matrix

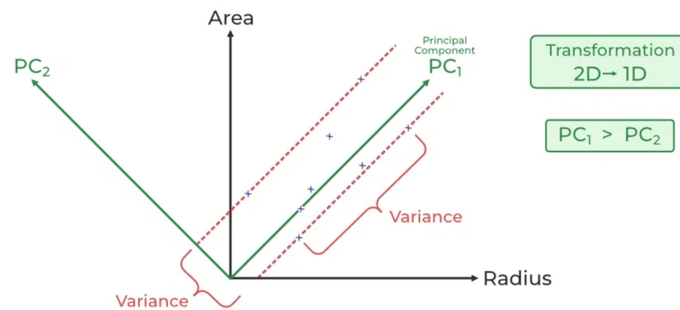


Figure 5: PCA

A confusion matrix is a specific table layout used to visualize the performance of a classification algorithm. It shows the number of correct and incorrect predictions made by the model compared to the actual classifications. The matrix is divided into four parts: true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN), allowing for detailed analysis of the model's performance, including metrics like accuracy, precision, recall, and F1 score.

		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

Figure 6: Confusion Matrix

2. Accuracy

   In machine learning, accuracy is a metric used to measure the proportion of correct predictions made by the model out of all predictions. It is calculated as the ratio of the number of correct predictions (both true positives and true negatives) to the total number of predictions (the sum of true positives, true negatives, false positives, and false negatives). Accuracy is a straightforward way to assess a model's performance, but it may not always provide a complete picture, especially in cases of imbalanced datasets.

$$accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)} \tag{5}$$

3. Precision

   Precision in the context of classification models refers to the ratio of true positive predictions to the total number of positive predictions (both true positives and false positives). It measures the accuracy of the positive predictions made by the model, indicating how many of the instances predicted as positive are actually positive. Precision is particularly important in scenarios where the cost of false positives is high.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{6}$$

4. TPR(recall)

   The True Positive Rate (TPR), also known as sensitivity or recall, measures the proportion of actual positives correctly identified by a classification model. It's a key performance indicator in various domains, especially in medical diagnostics and any field where the cost of missing a positive case is high. TPR is crucial for evaluating the effectiveness of a model in predicting positive outcomes, focusing on the model's ability to capture all relevant

instances.

$$\text{TPR} = \frac{TP}{TP + FN} \tag{7}$$

5. F1 Score

The F1 score is a measure of a test's accuracy. It considers both the precision p and the recall r of the test to compute the score. The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0. It's especially useful for situations where the distribution of class labels is imbalanced.

2.7 Platform

2.7.1 RISC-V: PULP

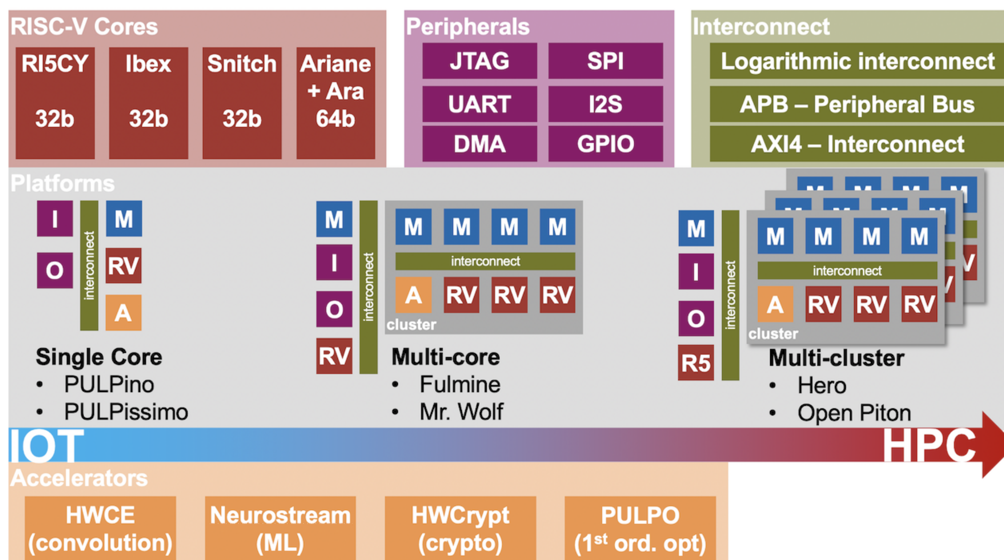


Figure 7: PULP-Family

PULPino is an open-source single-core microcontroller system, based on 32-bit

RISC-V cores developed at ETH Zurich. PULPino is configurable to use either the RISCY or the zero-riscy core. [14]

RISCY is an in-order, single-issue core with 4 pipeline stages and it has an IPC close to 1, full support for the base integer instruction set (RV32I), compressed instructions (RV32C) and multiplication instruction set extension (RV32M). It can be configured to have single-precision floating-point instruction set extension (RV32F). It implements several ISA extensions such as: hardware loops, post-incrementing load and store instructions, bit-manipulation instructions, MAC operations, support fixed-point operations, packed-SIMD instructions and the dot product. It has been designed to increase the energy efficiency of in ultra-low-power signal processing applications.zero-riscy is an in-order, single-issue core with 2 pipeline stages and it has full support for the base integer instruction set (RV32I) and compressed instructions (RV32C). It can be configured to have multiplication instruction set extension (RV32M) and the reduced number of registers extension (RV32E). It has been designed to target ultra-low-power and ultra-low-area constraints. zero-riscy implementes a subset of the 1.9 privileged specification.

When the core is idle, the platform can be put into a low power mode, where only a simple event unit is active and everything else is clock-gated and consumes minimal power (leakage). A specialized event unit wakes up the core in case an event/interrupt arrives.

For communication with the outside world, PULPino contains a broad set of peripherals, including I2S, I2C, SPI and UART. The platform internal devices can be accessed from outside via JTAG and SPI which allows pre-loading RAMs with executable code. In standalone mode, the platform boots from an internal boot ROM and loads its program from an external SPI flash.

The PULPino platform is available for RTL simulation as well FPGA. PULPino has been taped-out as an ASIC in UMC 65nm in January 2016. It has full debug

support on all targets. In addition we support extended profiling with source code annotated execution times through KCacheGrind in RTL simulations.

2.7.2 SOC environment: GVSOC

The last few years have seen the emergence of IoT processors: ultra-low power systems-on-chips (SoCs) combining lightweight and flexible micro-controller units (MCUs), often based on open-ISA RISC-V cores, with application-specific accelerators to maximize performance and energy efficiency. Overall, this heterogeneity level requires complex hardware and a full-fledged software stack to orchestrate the execution and exploit platform features. For this reason, enabling agile design space exploration becomes a crucial asset for this new class of low-power SoCs. In this scenario, high-level simulators play an essential role in breaking the speed and design effort bottlenecks of cycle-accurate simulators and FPGA prototypes, respectively, while preserving functional and timing accuracy. GVSoC is a highly configurable and timing-accurate event-driven simulator that combines the efficiency of C++ models with the flexibility of Python configuration scripts. GVSoC is fully open-sourced, with the intent to drive future research in the area of highly parallel and heterogeneous RISC-V based IoT processors, leveraging three foundational features: Python-based modular configuration of the hardware description, easy calibration of platform parameters for accurate performance estimation, and high-speed simulation. Experimental results show that GVSoC enables practical functional and performance analysis and design exploration at the full-platform level (processors, memory, peripherals and IOs) with a speed-up of 2500x with respect to cycle-accurate simulation with errors typically below 10% for performance analysis. [1]

3 Experimental Design

The system architecture for the AI-based anomaly detection in RISC-V environments is designed with flexibility, scalability, and performance in mind. At its core, the architecture comprises 4 main components:

1. the Data generation Module: Including 2 sets of C application(malware and benign), 1 random input generator.

2. the Data Collection Module: read output file and select target data.

3. the Anomaly Detection Module: Classification algorithm.

4. the Reporting Module: Matlab processing.

Based on the Pulp platform tool "Pulp-SDK [2]," my overall approach involved the following steps: first, I designed a normal program and a malicious program with harmful behavior. Then, I used a series of registers called "PMCs" provided by the Pulp-SDK to monitor the behavior of the programs. Next, I collected 17 sets of data from the monitoring process and used machine learning techniques to analyze the data. Finally, I tested the approach using a test dataset and used the results to verify the effectiveness of this method. Each module is designed with specific challenges in mind. The open-source RISC-V SDK is still in the development stage and lacks a comprehensive library of C files. As a result, during the application development process, we had to undertake the writing of some header files ourselves, such as those for AES encryption. When generating random numbers, we opted to use an external program to modify the data of the target program, thereby gradually overcoming all difficulties encountered. To facilitate a more intuitive evaluation of the final data, I employed MATLAB for real-time analysis, aiming to derive the most direct insights from the data.

3.1 Platform choices

Choice of RISC-V Platform: The selection of the RISC-V platform for anomaly detection is driven by its growing adoption in embedded systems and the unique security challenges it presents due to its open-source nature and architectural flexibility. This choice aligns with our research objective to contribute to the safety and reliability of RISC-V systems in the face of sophisticated cyber threats.

3.2 Benign and Malicious application design

Given the scarcity of publicly available datasets for anomaly detection in RISC-V, we designed 2 applications: Benign application and malicious application design to collect data.

For the benign program, we started with an existing C program that sorts 10 data elements. We modified the program such that instead of a fixed input of 10 data elements, the program now accepts a random sequence of data with a length between 100 and 200. The data content is also randomized. However, since the Pulp-SDK's C library does not have a standard library function for generating random numbers, we used an external program called "random" that was designed using the standard C library to provide the random length and content for the program. Finally, we monitored the program's PMCs.

For the malicious program, we added an encryption function called "Caesar cipher" to the main program(based on benign program),this function change encode the content of array(numbers) of benign application, which we considered to be a harmful behavior. Then, we monitored the program's PMCs to detect any unusual behavior.

In addition, to ensure the reliability of the experiment, a control group was added to this experiment: AES encryption was used as a benign program.

	Benign Program	Malware Program
Program 1	Quiksort	Quiksort + Ceaser
Program 2	AES Encription	AES + Ceaser

Table 2: Benign and Malware Application

3.3 Scripts design

Given the scarcity of publicly available datasets for anomaly detection in RISC-V, to collect sufficient data, I designed the following script(shell command): first, the script runs the random program to generate a random input, then it modifies the input of the benign program accordingly, and runs the benign program. The output of the program is then saved to a file. This process is repeated in a loop until 10,000 sets of data are collected. The same process is applied to the malicious program.

3.3.1 Data Collection

After analyzing the results of the two programs (17 sets of register data), we found that some of the data remained constant at 0, while some data remained unchanged throughout the program's execution. To improve the efficiency of the experiment, we removed these data and collected only the effective 9 sets of register data.

3.3.2 Data Preprocessing

For the testing dataset, we created a one-dimensional dataset using the results from a single register. We then randomly selected two sets of register data to create a two-dimensional dataset. This process was repeated for three, four, and five-dimensional datasets until we had a nine-dimensional dataset that included all of the registers. Note that for each training and testing dataset, the data used

came from the same set of registers.

(a) 1-D samples (b) 2-D samples

(c) 3-D samples (d) 4-D samples

(e) 5-D samples (f) 9-D samples

Figure 8: Sample data format

3.4 AI program design

To conduct machine learning, I designed a Python program based on the Sci-Kit [13] platform. The program performs machine learning on two sets of data, each consisting of 10,000 samples. The results from the malicious program are labeled

as "Malware." The program then tests a separate set of data that is generated from two programs, determine the percentage of this set of data from malicious programs. The testing dataset is not used in the machine learning process, but instead is used to evaluate the accuracy of the model's predictions.

By analyzing the training results of different models on the same dataset, we can assess the feasibility of this experiment. Then, by examining the training outcomes of each model on various datasets, we ensure the reliability of the results.

My AI program design includes:

1. Data integration and preprocessing: creating combinations with 2, 3, 4, 5 registers, each providing one feature.
2. Data merging to form arrays required by the algorithm model.
3. Determining the model's algorithm.
4. Using existing X, Y datasets to derive the confusion matrix, Accuracy, TPR (True Positive Rate), and F1 Score.

This section details the training and evaluation of several machine learning models to identify anomalies in RISC-V based systems, utilizing two distinct sets of data generated from programs based on sorting algorithms and AES encryption. The models selected for this study include Logistic Regression, SVC-linear, Decision Tree Classifier, RandomForestClassifier, GaussianNB, and Quadratic Discriminant Analysis, as implemented in scikit-learn. Following model training and initial evaluation, Principal Component Analysis (PCA) was employed to enhance data processing and potentially improve model performance.

1. Data Preparation: The datasets were prepared from synthetic logs generated by simulating the execution of sorting algorithms and AES encryption on a

RISC-V architecture. This approach ensured the inclusion of diverse and representative patterns of normal and anomalous behavior.

2. Model Selection and Training: Each model was chosen for its unique strengths in classification tasks, providing a broad perspective on anomaly detection performance across different algorithms. Two rounds of training and evaluation were conducted for each model to assess consistency and reliability across varying data characteristics.

3. Feature Reduction with Feature Selection: To address potential issues of high dimensionality and improve model efficiency, Feature selection algorithm was applied to the datasets post-initial training. This step aimed to reduce the feature space while retaining the most informative aspects of the data.

4. Impact of PCA: The application of PCA resulted in noticeable improvements in model performance, particularly for Logistic Regression and Quadratic Discriminant Analysis, which benefited from the reduced feature space and enhanced generalization capabilities.

3.5 Experimental procedures

Then we did feature selection:

1. Malware Design: Using examples provided by the Pulp-SDK, I modified designs to create two sets of "benign" C programs focused on sorting and AES encryption (6). Then, I introduced modifications to segments of these datasets to alter data processing in a manner consistent with the definition of a "virus", referring to these as "malicious programs". Subsequently, data recorded by PMCs registers was printed out.

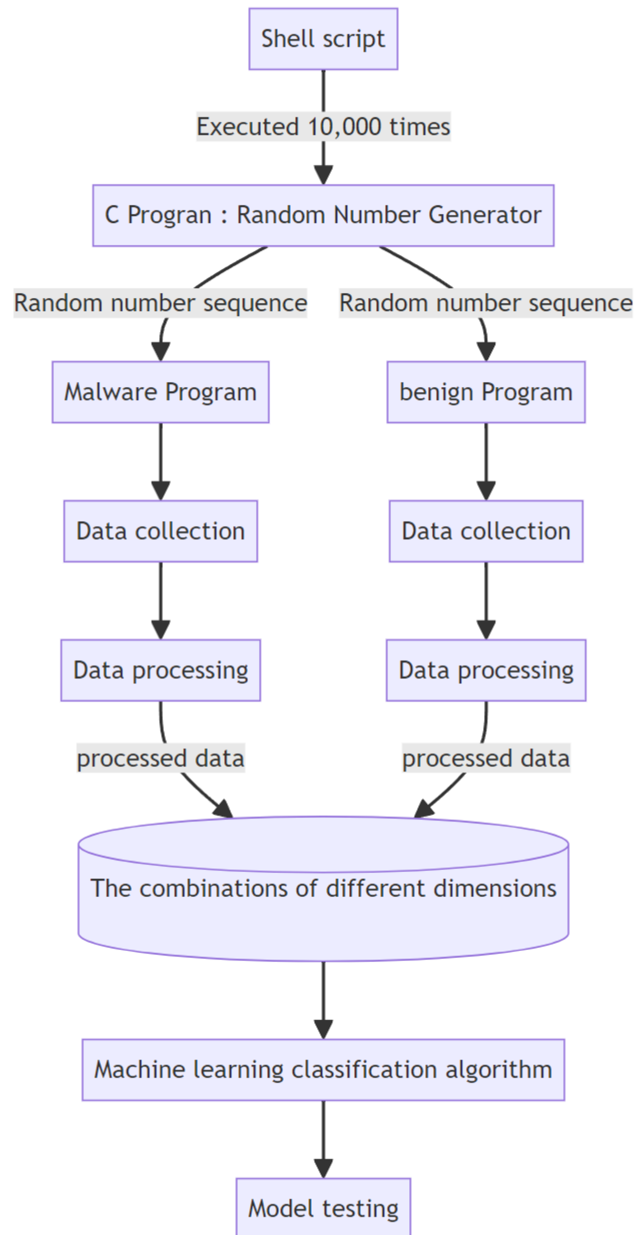


Figure 9: Flowchart of Classification

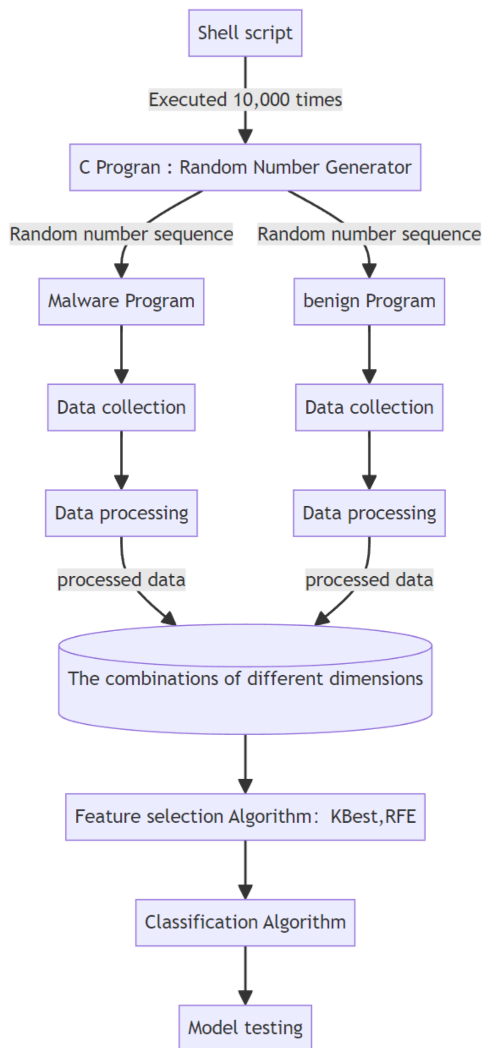


Figure 10: Flowchart of Feature Selection

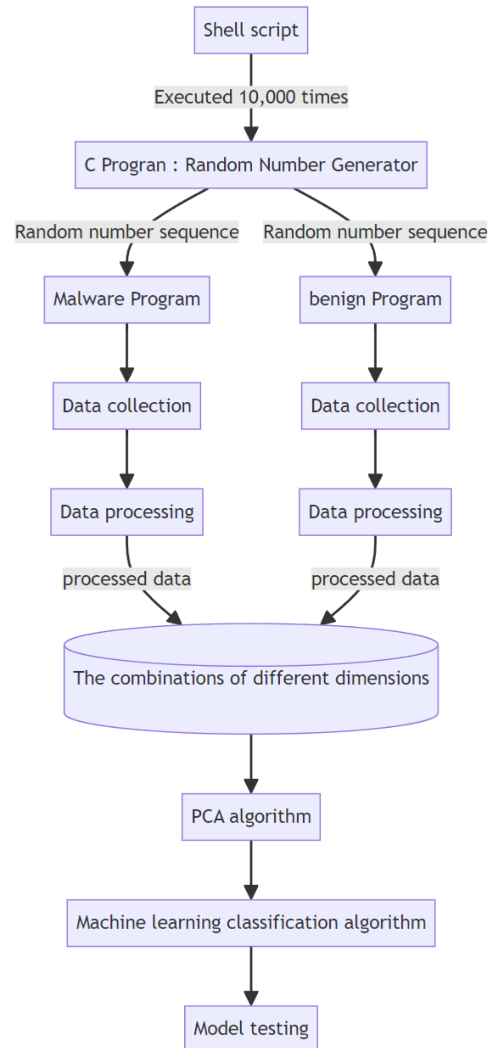


Figure 11: Flowchart of PCA

2. Input Variation through External C Program: The inputs to the programs were varied using an external C program random number generator, ensuring that each run of the programs had unique inputs and outputs.

3. Automated Execution via Shell Script: Programs were executed in a blocking loop through a shell script, with output files being saved for further analysis.

4. Register Data Extraction: I extracted the corresponding register data from the output files for analysis.

5. Model Training with scikit-learn: Utilizing various Python algorithms provided by scikit-learn, I trained models on the extracted data to determine their accuracy. These models were then used to identify newly generated data, validating the models' performance.

6. Dimensionality Reduction with PCA: I applied Principal Component Analysis (PCA) to the data for dimensionality reduction, followed by retraining the models.

7. Feature Ranking of Registers: I identified and ranked the features of the registers to determine their significance.

This summary encapsulates the methodology and steps taken to design, implement, and evaluate an AI-based approach for malware detection within the context of PMCs data, leveraging machine learning techniques and PMCs for enhanced analysis and model optimization.

4 Experimental Results

There are 17 sets of registers in total, 8 of which have no valid data, so we only analyze the remaining 9 sets.

Register	Availability
PERF_CYCLES	✓
PERF_INSTR	✓
PERF_ACTIVE_CYCLES	×
PERF_LD_STALL	✓
PERF_JR_STALL	×
PERF_JR_IMISS	×
PERF_LD	✓
PERF_ST	✓
PERF_JUMP	✓
PERF_BRANCH	✓
PERF_BTAKEN	✓
PERF_RCV	✓
PERF_LD_EXT	×
PERF_ST_EXT	×
PERF_LD_EXT_CYC	×
PERF_ST_EXT_CYC	×
PERF_TCDM_CONT	×

Table 3: PMCs list

4.1 Experiment 1: Classification

4.1.1 Classification with Program.1

First of all, I would like to report that I started with one-dimensional data as the learning set(use one of all 9 registers as a learning set)，and one ML classification method "LogisticRegression". and then test the random combination of registers,from groups of 2 up to groups of 9. After using every single one of these nine registers, I found that all Machine Learning models had an accuracy close to 50%. When testing on the test set, the results were all incorrect.
Here is one of the results，In this test, we input the sample 100% from the benign application，but the testing and predicting are the exact opposite, so we can say that we can't do malware detection by a single set of PMCs:

```
Accuracy: 0.49241666666666667
Percentage of data from malware: 1.0
```

Next, I proceeded to use 2-D, 3-D, 4-D, 5-D, and finally 9-D data sets to complete all the tests, for testing use the registers same as training. The results obtained from the tests are as follows: In these charts, x-axis corresponding to each combination of registers. For example, for "Accuracy Data for 2-D", x-axis corresponding to 36 different set of registers. The accuracy charts presented above are obtained from 2-D, 3-D, 4-D, and 5-D data, respectively. The four bar charts present the increasing of accuracy, as the sample dimension increases, the accuracy shows a significant improvement, and the accuracy reaches 100% in 5-D. One thing to note is that，in 3-D accuracy, there are there combination of registers reach to 100%, I report these 3 accuracy with a test of 100% samples from benign application:

```
branch_cnt.txt jump_cnt.txt ls_cnt.txt
Accuracy: 1.0
```

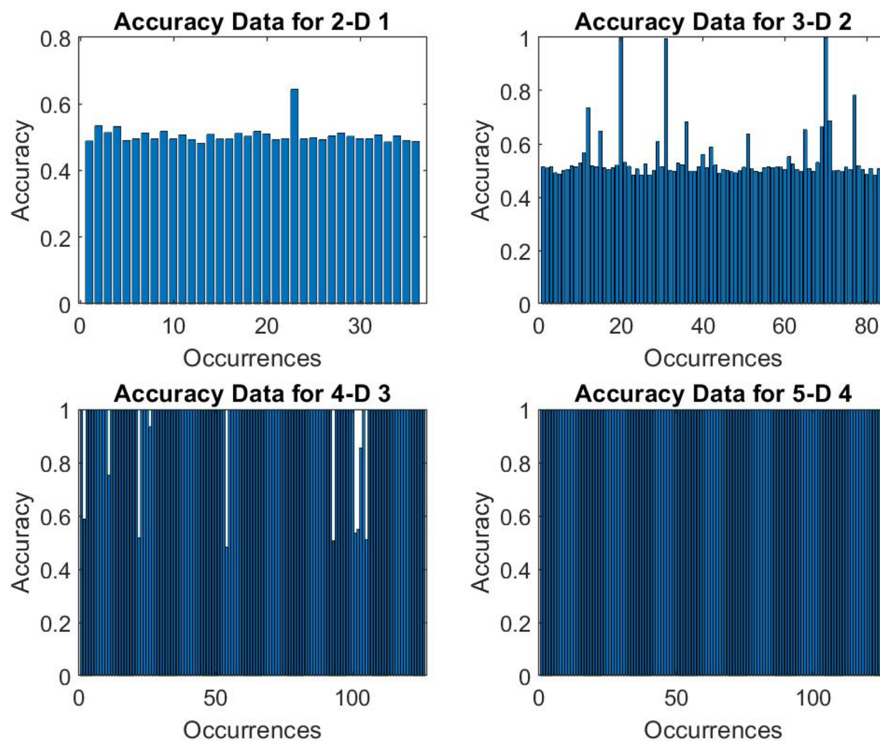


Figure 12: Accuracy of Machine Learning

```
Percentage of malware from data of benign programs: 0.0

btaken_cnt.txt clk_cnt.txt ld_cnt.txt
Accuracy: 0.992
Percentage of malware from data of benign programs: 0.0

instr_cnt.txt ld_cnt.txt rcv_cnt.txt
Accuracy: 1.0
Percentage of malware from data of benign programs: 0.0
```

After that, I verified the accuracy of the model:

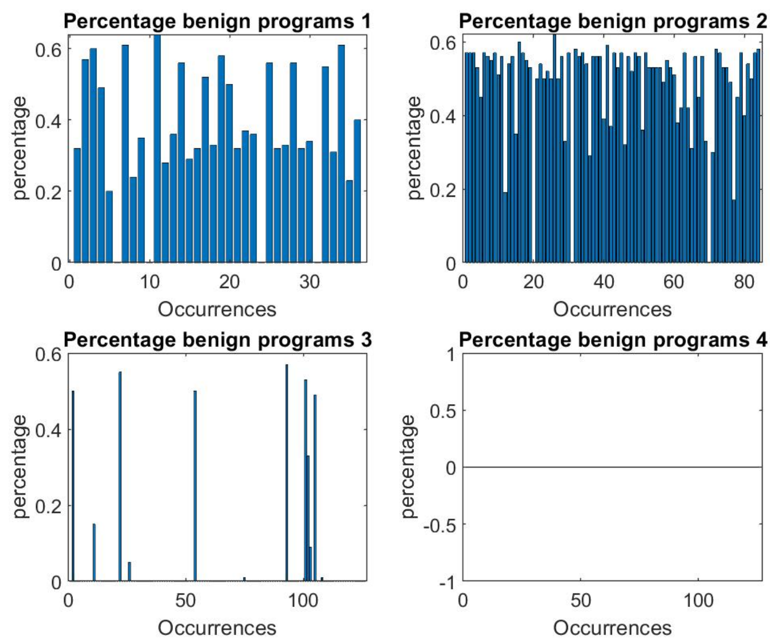


Figure 13: Percentage of Malware, data from benign application

Finally, we found that when the sample reaches 5 dimensions, the accuracy reaches 100%.

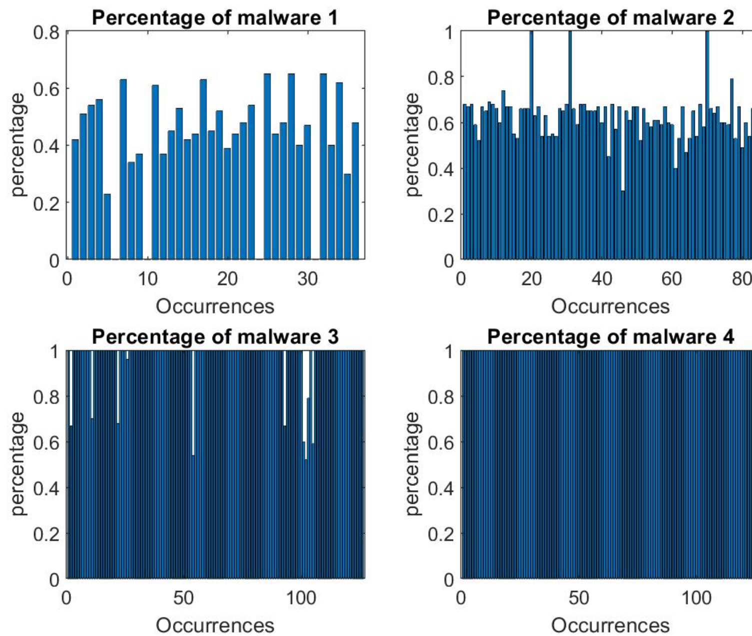


Figure 14: Percentage of malware, data from malicious application

In the previous section, we conducted a feasibility test from which we selected two models that showed the most promising results: "Logistic Regression" and "Quadratic Discriminant Analysis (QDA)." We aimed to explore the relationship among four key performance metrics: Accuracy, Precision, Recall (also known as True Positive Rate, TPR), and F1 Score.

Firstly, it's important to note that these metrics align with the mathematical relationships presented in the confusion matrix, a table used to describe the performance of a classification model. This alignment underscores the coherence and reliability of our evaluation approach.

Upon a horizontal comparison between the two models, we observed that QDA slightly outperformed Logistic Regression in terms of Precision. This suggests that the QDA model exhibits a higher reliability in classifying instances correctly within the specific context of the 'qsort' program. Precision, in this sense, mea-

```
branch_cnt.txt jump_cnt.txt ld_cnt.txt
Accuracy: 0.5195
Percentage of malware from data of benign programs: 0.53
Confusion Matrix:
[[ 800 1181]
 [ 741 1278]]
Accuracy: 0.52
Precision: 0.52
Recall (TPR): 0.63
F1 Score: 0.57

branch_cnt.txt jump_cnt.txt ls_cnt.txt
Accuracy: 1.0
Percentage of malware from data of benign programs: 0.0
Confusion Matrix:
[[1981    0]
 [   0 2019]]
Accuracy: 1.00
Precision: 1.00
Recall (TPR): 1.00
F1 Score: 1.00
```

Figure 15: Logistic matrix

```
branch_cnt.txt jump_cnt.txt ld_cnt.txt
Accuracy: 0.528
Percentage of malware from data of benign programs: 0.44
Confusion Matrix:
[[ 971 1010]
 [ 878 1141]]
Accuracy: 0.53
Precision: 0.53
Recall (TPR): 0.57
F1 Score: 0.55

branch_cnt.txt jump_cnt.txt ls_cnt.txt
Accuracy: 1.0
Percentage of malware from data of benign programs: 0.0
Confusion Matrix:
[[1981    0]
 [   0 2019]]
Accuracy: 1.00
Precision: 1.00
Recall (TPR): 1.00
F1 Score: 1.00
```

Figure 16: QDA matrix

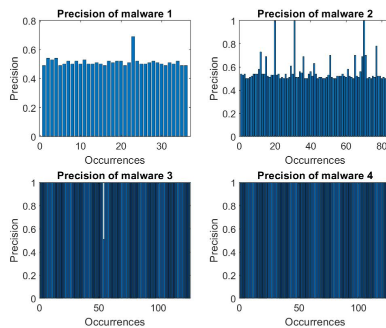


Figure 17: Logistic Precision

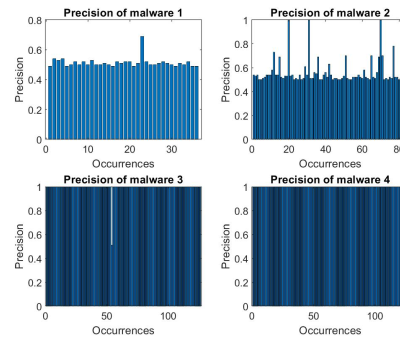


Figure 18: QDA Precision

sures the model's ability to return only relevant instances, making this finding particularly significant. However, the margin of difference between the two models is not substantial, indicating that both models perform comparably well under the conditions tested.

This analysis provides valuable insights into the performance characteristics of Logistic Regression and QDA models, specifically in their application to the 'qsort' program. The nuanced difference in Precision highlights the importance of selecting the appropriate model based on the specific criteria and context of the task at hand. To make the data more visually intuitive, we can observe by

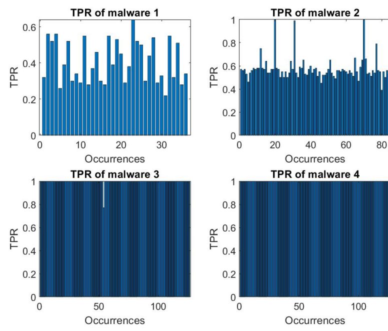


Figure 19: Logistic TPR

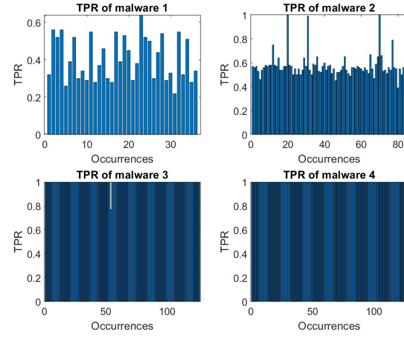


Figure 20: QDA TPR

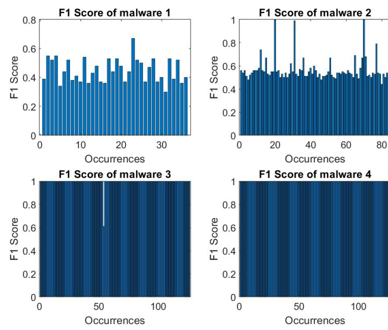


Figure 21: Logistic F1 Score

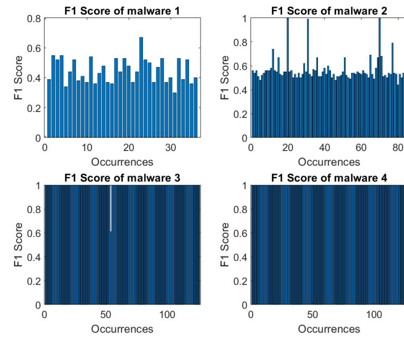


Figure 22: QDA F1 Score

comparing the mean values of two groups of algorithms.

	Logistic				Quadratic Discriminant Analysis			
	Accurancy	Precision	TPR	F1	Accurancy	Precision	TPR	F1
2-D	0.51	0.46	0.42	0.43	0.51	0.51	0.41	0.45
3-D	0.55	0.55	0.61	0.58	0.58	0.55	0.56	0.56
4-D	0.97	0.97	0.97	0.97	1.00	1.00	1.00	1.00
5-D	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 4: Program.1, The mean values of two groups of algorithms.

From this data, we can also verify the correspondence between matrices and

data, as well as evaluate the strengths and weaknesses of algorithm models, as mentioned above.

Afterwards, we tried several other classification methods provided by sci-kit : By observing the form, there are three were viable. However, it should be noted

	2	3	4	5	9
LogisticRegression	50	55	97	100	100
(*)SVC-linear	50	50	100	100	100
DecisionTreeClassifie	50	50	50	50	50
RandomForestClassifier	50	50	50	50	50
GaussianNB	50	50	50	50	50
QuadraticDiscriminantAnalysis	50	60	99	100	100

Table 5: Accuracy of Program.1

that the SVC-linear method showed a very high accuracy rate, but its speed was extremely slow, approaching 1% of other methods. We had difficulty testing all the data with this method. Interestingly, when using the QuadraticDiscriminantAnalysis method for machine learning on the training set, there was often a warning message saying "Variables are collinear." We can assume that some of the register data in this experiment are linearly related, and we may only need to use one of them when creating the training set.

4.1.2 Controlled experiment: Classification with program.2

We changed the benign program, as a control experiment.

In this experiment, I employed two sets of programs as controls: one group consisted of a benign program called "AES Encryption/Decryption," while the other group consisted of a malicious program named "AES Encryption/Decryption + Caesar Encryption."

Using PMCs, I observed that the size of the AES Encryption/Decryption program was approximately 100 times larger than the size of the Caesar Encryption program. Additionally, both programs were tested with randomly generated inputs of varying lengths.

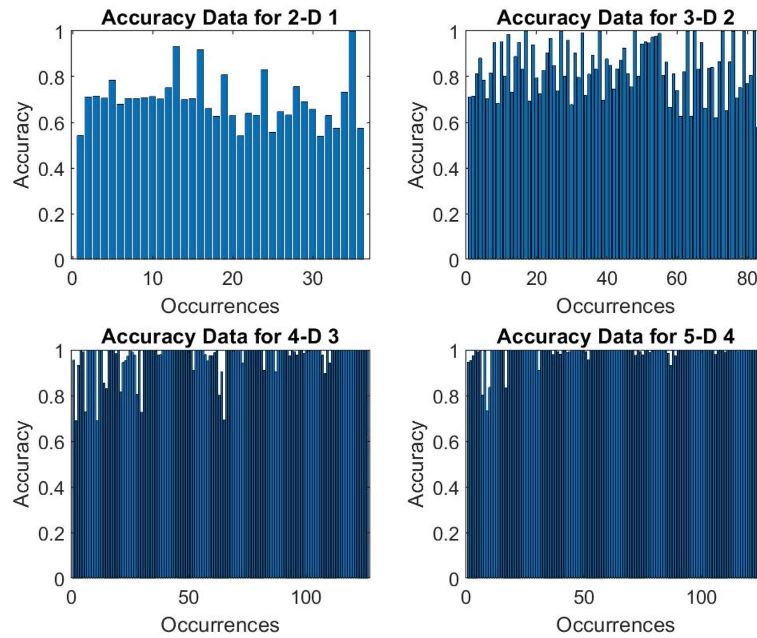


Figure 23: AES:LogisticRegression

	2	3	4	5
LogisticRegression	70	84	97	99
QuadraticDiscriminantAnalysis	87	99	100	100
RandomForestClassifier	100	100	100	100
DecisionTreeClassifier	100	100	100	100
GaussianNB	57	59	60	61

Table 6: results average of 4 machine learning methods

In our study, we made several key observations through the control experi-

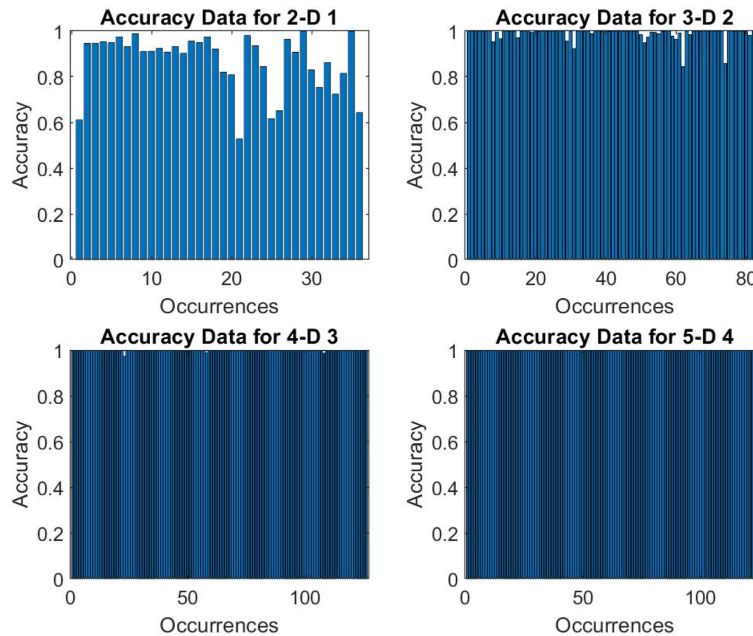


Figure 24: AES:QuadraticDiscriminantAnalysi

ments:

1.Different data sources yield different results for various machine learning analysis methods. For instance, the LogisticRegression and QuadraticDiscriminant-Analysis, which previously did not demonstrate the ability to analyze malicious programs effectively in the quicksort program, exhibited excellent performance in this experiment. Even with just two sets of registers, they were able to accurately distinguish harmful programs.

2.The "disguise" capability of malicious software is not only reflected in the program's size but also in its similarity to benign programs. Further experiments are needed to validate this hypothesis.

To summarize, our findings highlight the importance of considering different data sources and analysis methods for effective detection of malicious programs. Furthermore, we emphasize the need for further research to explore the relationship

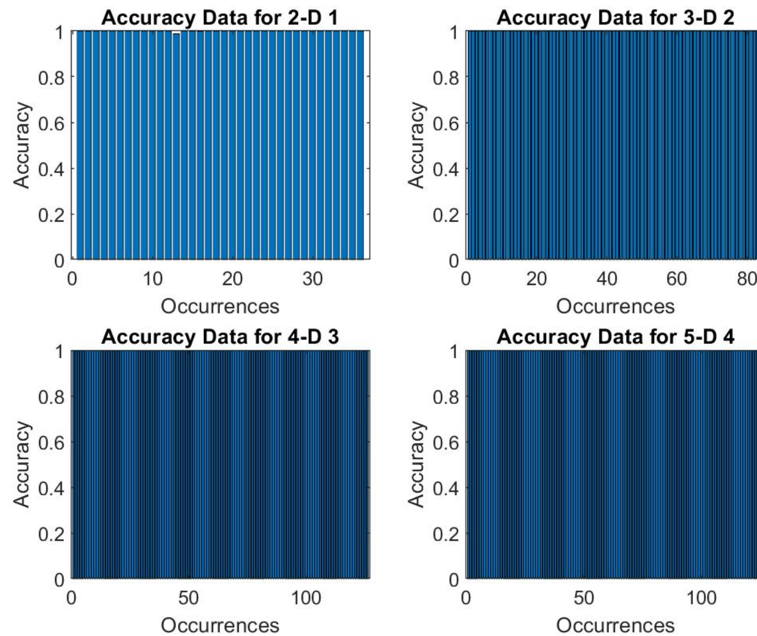


Figure 25: AES:RandomForestClassifier

between program size, similarity to benign programs, and the ability to evade detection.

4.2 Experiment 2: Feature selection

In this section, we have decided to employ feature detection to rank the features we are utilizing.

In the previous two sets of experiments, the control group utilized very low dimensions (e.g., randomly generated 2-dimensional data, figure:25, 27), and the dataset exhibited high discriminability. Consequently, we have chosen to leverage the results from the first set of experiments for feature detection.

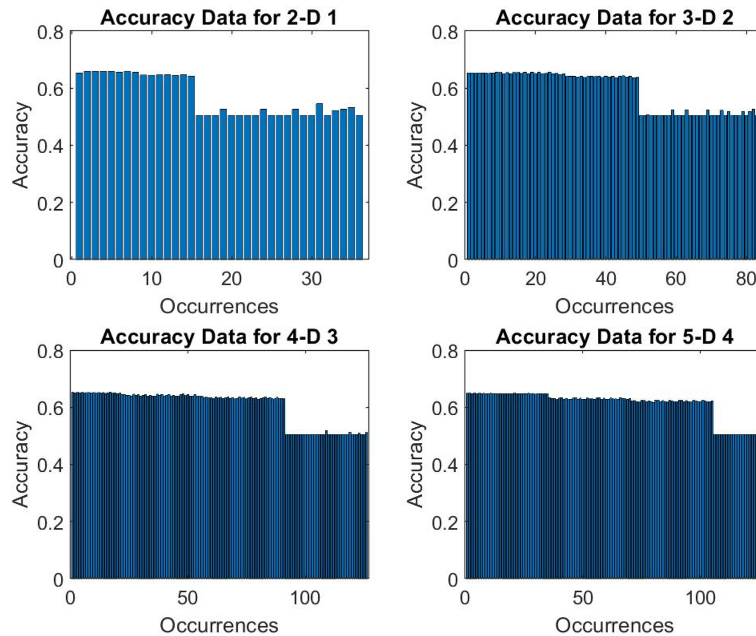


Figure 26: AES:GaussianNB

4.2.1 Selection of experimental subjects

We have two experiments. The benign software in the first experiment is the qsort program, and the benign program in the second experiment is the AES encryption program. Why was the first program chosen as the experimental subject for this feature selection? Because of the results of the first experiment, when classifying the data, the second experiment has shown a high recognition rate in the combination of the two sets of data, and the result I hope to get is that only a few target registers are our The desired result, if the results all have a success rate of 100, we cannot determine whether the feature selection result is right or wrong. We need a "unique answer"if the target register can be found using feature selection, then we can say that the experiment was successful.

In the first experiment (29), we saw that among the three sets of data, three combinations showed a success rate of 100. We assumed that these three combi-

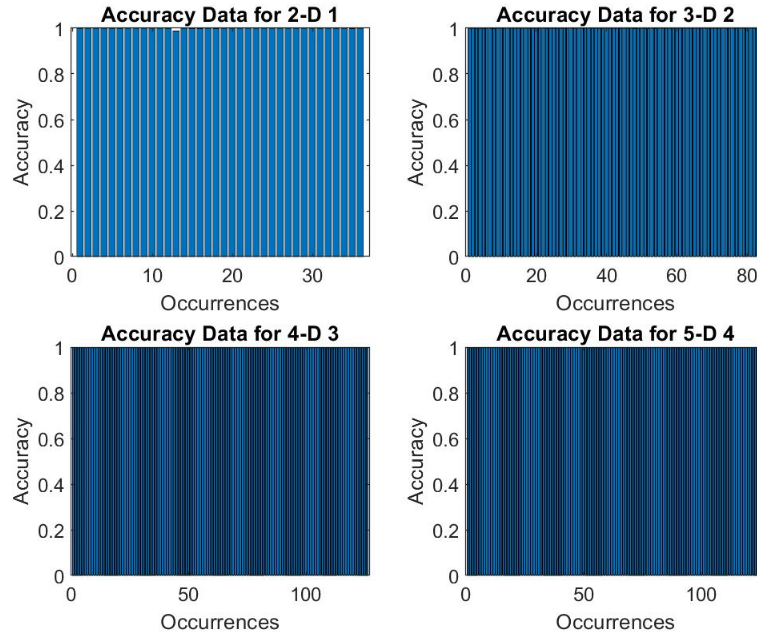


Figure 27: AES:DecisionTreeClassifier

nations were our "unique answer".

So, our "unique answer" is these three sets of registers : According to the comparison table below, we can express it as: (2, 4, 6), (1, 3, 5), (0, 5, 7)

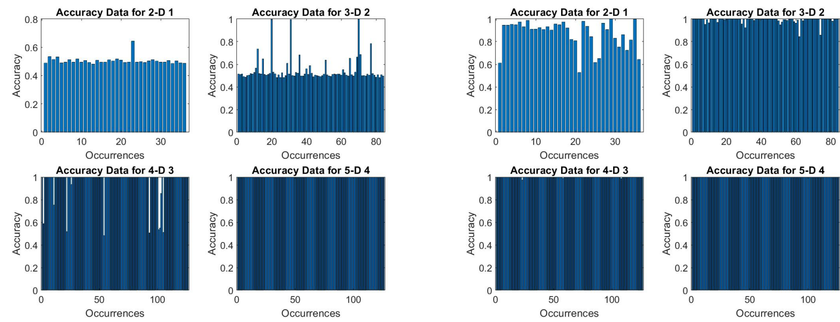
4.2.2 Experiment procedure

0	1	2	3	4	5	6	7	8
instr_cnt	clk_cnt	branch_cnt	btaken_cnt	jump_cnt	ld_cnt	ls_cnt	rcv_cnt	st_cnt

Table 7: Number of registers for PMCs

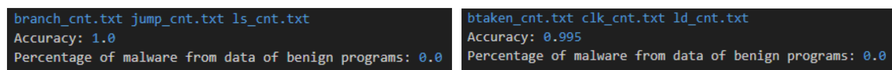
I collected research on feature selection on the sci-kit official website. There are three ways:

- Feature Importance Scores Tree-based models such as Random Forest and Decision Trees can provide feature importance scores.



(a) accuracy of 1st experiment (b) accuracy of 2nd experiment

Figure 28: The accuracy of 2 experiments



```
branch_cnt.txt jump_cnt.txt ls_cnt.txt
Accuracy: 1.0
Percentage of malware from data of benign programs: 0.0
```

```
btaken_cnt.txt clk_cnt.txt ld_cnt.txt
Accuracy: 0.995
Percentage of malware from data of benign programs: 0.0
```

(a) Combination 1 (b) Combination 2

```
instr_cnt.txt ld_cnt.txt rcv_cnt.txt
Accuracy: 1.0
Percentage of malware from data of benign programs: 0.0
```

(c) Combination 3

Figure 29: 3 register combinations that can achieve 100% accuracy

- Recursive Feature Elimination (RFE) RFE is a method that reduces the number of features incrementally by repeatedly training the model and eliminating the least important features.

- SelectKBest SelectKBest is a method for selecting the top k best features.

Based on these three procedures, we ranked the features of the data : As we can see though the figure (30), there is no unique ordering, every time it's different, and none of the answers are our correct answers.

```
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_feature.py
selected feature:  [0 7 8]
accuracy 0.4955
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_feature.py
selected feature:  [0 6 7]
accuracy 0.49875
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_feature.py
selected feature:  [0 2 7]
accuracy 0.5045
```

Figure 30: Feature Importance Scores

```
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_RFE.py
Optimal number of features: 4
Feature Ranking:
[6 4 3 1 1 1 1 2 5]
Accuracy on test set with top 3 features: 0.56125
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_RFE.py
Optimal number of features: 4
Feature Ranking:
[6 4 3 1 1 1 1 2 5]
Accuracy on test set with top 3 features: 0.56125
```

Figure 31: RFE

```
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_K.py
selected feature: [0 1 7]
Accuracy on test set with top 3 features: 0.4965
cristiano@cristiano-VirtualBox:~/Documents/pulp-sdk/tests/qsort$ python3 ml_9d_K.py
selected feature: [0 1 7]
Accuracy on test set with top 3 features: 0.4965
```

Figure 32: SelectKBest

By comparing the results, we can find that none of them are correct. Interestingly, most of the results had two items that were close to the correct answer.

Possible outcomes:

1. Feature selection may have other ways to find out the rankings, except for the three methods I used.

2. For successful identification of harmful programs, in this case, a single register cannot give an exact ranking. That is to say, a feature of a single data cannot be found without feature selection, and a combination of at least two registers is required.

4.3 Experiment 3: Feature ranking

The primary objective of this phase of our study is to methodically rank the features within our dataset based on their contributions to the model's predictive capabilities. This ranking not only aims to identify the most influential features but also sets the stage for a deeper investigation into how these features interact with one another. We defined these specific feature groupings (2, 4, 6), (1, 3, 5), and (0, 5, 7) based on previous experiments(4.2.1). Following the establishment of these rankings, our analysis will extend to examining the correlations between features within these identified groupings.

In the pursuit of understanding the varying contributions of features within our dataset, we have ventured into utilizing Principal Component Analysis (PCA) as a method for dimensionality reduction [7]. The rationale behind this choice stems from our observations in previous experiments, where PCA's inherent mechanism of discarding less contributive data highlighted its potential for our objective.

The process begins with the application of PCA to reduce the dataset's dimensions. This reduction is not arbitrary; it is a calculated procedure where components with minimal contribution towards the variance in the dataset are

systematically removed. By setting different principal components, we aim to dissect the variance each feature contributes to the overall dataset.

With nine features under scrutiny, our method involves calculating the weight of each feature within the PCA-transformed space. These weights serve as indicators of the features' relative importance or contribution to the dataset's variance. The underlying assumption is that features with higher weights are of greater significance, thereby meriting a higher rank.

The following table 8 shows the feature weights corresponding to different principal component settings.

Analysis :

- Low Model Accuracy with 1 to 3 Principal Components: Further analysis revealed that models derived with principal component settings ranging from 1 to 3 exhibited lower accuracy. This reduction in model performance indicates a potential loss of critical information, making these settings unsuitable for effective feature ranking. As a result, configurations within this range were also set aside.

- Focus on Principal Component Settings Above 3: Based on the aforementioned insights, the analysis has been refined to consider only scenarios where the number of principal components exceeds 3. This decision is underpinned by the aim to ensure a balance between dimensionality reduction and the retention of significant information necessary for accurate feature ranking.

- Variance verification: finally we verified the results by variance calculation. When the component of PCA set to 1, the ranking is same as variance of feature.

Components	Weights ranking
1	1,0,7,2,5,3,8,4,6
2	1,0,7,8,3,5,2,4,6
3	0,7,1,8,3,2,6,5,4
4	7,0,1,2,3,5,8,4,6
5	0,7,1,8,3,2,5,6,4
6	0,7,2,5,1,3,8,6,4
7	8,0,3,5,4,7,2,1,6

Table 8: PCA feature ranking

Feature	Variance
1	156892296.43453774
0	11738311.840617886
7	2373639.2341436697
2	409135.59637775
5	338026.62665397907
3	232575.15949103888
8	173336.34102604105
4	154121.04762287906
6	83571.75457247745

Table 9: Variance ranking

5 Conclusion and Future Work

5.1 Research Contributions and Limitations

This study aimed to enhance AI-based anomaly detection in RISC-V architectures by exploring various machine learning methods and their effectiveness in identifying malicious software activities. Our investigation utilized a diverse dataset, focusing on the analysis of performance metrics collected from RISC-V processors. Among the methods tested, the DecisionTreeClassifier and RandomForestClassifier demonstrated exceptional accuracy in detecting anomalies, showcasing the potential of machine learning techniques in cybersecurity applications within RISC-V environments.

However, the study also identified limitations :

- Due to the limitations of RISC-V SDK library functions and file systems, many malware samples are difficult to simulate, making it challenging to conduct further testing.
- For some classification algorithm, such as the SVC-linear method's slow performance and challenges in handling collinear variables, which could affect the scalability and efficiency of these anomaly detection systems.
- We didn't find the rank for the single register.

5.2 Areas for Improvement and Future Work

- Analyze the registers that present useful data and identify which registers have a higher contribution to the experiment.
- Analyze the registers that present useful data and determine which registers are correlated in this experiment.

- Exploration of Additional Data Sources and Features: Expanding the dataset to include a wider range of performance metrics and exploring the use of feature selection techniques could improve the models' ability to distinguish between benign and malicious activities.

- Optimization of Machine Learning Algorithms: Further research should aim to enhance the efficiency and accuracy of machine learning models, particularly in handling large datasets and reducing computational overhead.

- Some papers [9] suggest that imposing a time constraint on data collection on HPCs without using machine learning can provide more information about harmful software. In this experiment, we can try to add a time constraint. However, we have not validated the feasibility of this method due to limitations of the SDK tools.

- Development of Real-Time Anomaly Detection Systems: Future studies could focus on implementing these machine learning models in real-time detection systems, assessing their practicality and effectiveness in live RISC-V environments.

- Cross-Architecture Validation: Validating the effectiveness of the proposed anomaly detection methods across different RISC-V architectures and comparing their performance with other processor architectures could provide insights into the generalizability of these techniques.

- This research contributes to the growing body of knowledge in cybersecurity for RISC-V architectures, providing a foundation for further exploration and development in this critical area. By addressing the challenges and exploring the suggested future directions, the field can move closer to realizing robust,

AI-driven security solutions that can protect against increasingly sophisticated threats.

References

[1] Nazareno Bruschi, Germain Haugou, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. Gvsoc: A highly configurable, fast and accurate full-platform simulator for risc-v based iot processors. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 409–416, 2021.

[2] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. Dory: Automatic end-to-end deployment of real-world dnns on low-cost iot mcus. *IEEE Transactions on Computers*, 70(8):1253–1268, 2021.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3), jul 2009.

[4] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2, 07 2007.

[5] Cristiano Pegoraro Chenet, Alessandro Savino, and Stefano Di Carlo. A survey of hardware-based malware detection approach, 2023.

[6] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *SIGARCH Comput. Archit. News*, 41(3):559–570, jun 2013.

[7] Kayikkalthop M. George. An application of pca to rank problem parts. In *International Conference on Computational Science, Engineering and Information Technology*, 2012.

[8] Stavros Kalapothas, Manolis Galetakis, Georgios Flamis, Fotis Plessas, and Paris Kitsos. A survey on risc-v-based machine learning ecosystem. *Information*, 14(2):64, 2023.

[9] Abraham Peedikayil Kuruvila, Sayar Karmakar, and Kanad Basu. Time series-based malware detection using hardware performance counters. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 102–112, 2021.

[10] Huamin Li, George Linderman, Arthur Szlam, Kelly Stanton, Yuval Kluger, and Mark Tygert. Algorithm 971: An implementation of a randomized algorithm for principal component analysis. *ACM Transactions on Mathematical Software*, 43:1–14, 01 2017.

[11] Gilles Louppe. Understanding random forests: From theory to practice. *arXiv: Machine Learning*, 2014.

[12] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, STC '11, pages 71–76, New York, NY, USA, 2011. Association for Computing Machinery.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] PULP Project. Pulp platform. Available at https://pulp-platform.org/ (2023/09/25).

[15] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. Unsupervised anomaly-based malware detection using hardware features. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 109–129, Cham, 2014. Springer International Publishing.

6 Appendix

1. AES encryption

```
#include <string.h> // CBC mode, for memset
#include "aes.h"
#include "pmsis.h"

#define Nb 4

#if defined(AES256) && (AES256 == 1)
    #define Nk 8
    #define Nr 14
#elif defined(AES192) && (AES192 == 1)
    #define Nk 6
    #define Nr 12
#else
    #define Nk 4        // The number of 32 bit words in a key.
    #define Nr 10       // The number of rounds in AES Cipher.
#endif

#ifndef MULTIPLY_AS_A_FUNCTION
  #define MULTIPLY_AS_A_FUNCTION 0
#endif


typedef uint8_t state_t[4][4];


```

```
26 static const uint8_t sbox[256] = {
27   //0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
28   0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01,
       0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
29   0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4,
       0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
30   0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5,
       0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
31   0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12,
       0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
32   0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b,
       0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
33   0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb,
       0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
34   0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9,
       0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
35   0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6,
       0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
36   0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7,
       0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
37   0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee,
       0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
38   0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3,
       0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
39   0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56,
       0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
40   0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd,
       0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
```

```
41   0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35,
         0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
42   0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e,
         0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
43   0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99,
         0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };
44
45 #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
46 static const uint8_t rsbox[256] = {
47   0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40,
         0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
48   0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e,
         0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
49   0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c,
         0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
50   0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b,
         0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
51   0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4,
         0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
52   0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15,
         0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
53   0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4,
         0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
54   0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf,
         0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
55   0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2,
         0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
56   0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9,
         0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
```

```
57    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7,
         0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
58    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb,
         0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
59    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12,
         0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
60    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5,
         0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
61    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb,
         0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
62    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69,
         0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };
63  #endif
64
65  static const uint8_t Rcon[11] = {
66    0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36
         };
67
68
69  static uint8_t getSBoxValue(uint8_t num)
70  {
71    return sbox[num];
72  }
73
74  #define getSBoxValue(num) (sbox[(num)])
75
76  static void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key)
77  {
78    unsigned i, j, k;
```

```
79  uint8_t tempa[4]; // Used for the column/row operations
80
81  // The first round key is the key itself.
82  for (i = 0; i < Nk; ++i)
83  {
84    RoundKey[(i * 4) + 0] = Key[(i * 4) + 0];
85    RoundKey[(i * 4) + 1] = Key[(i * 4) + 1];
86    RoundKey[(i * 4) + 2] = Key[(i * 4) + 2];
87    RoundKey[(i * 4) + 3] = Key[(i * 4) + 3];
88  }
89
90  // All other round keys are found from the previous round keys.
91  for (i = Nk; i < Nb * (Nr + 1); ++i)
92  {
93    {
94      k = (i - 1) * 4;
95      tempa[0]=RoundKey[k + 0];
96      tempa[1]=RoundKey[k + 1];
97      tempa[2]=RoundKey[k + 2];
98      tempa[3]=RoundKey[k + 3];
99
100   }
101
102   if (i % Nk == 0)
103   {
104     // [a0,a1,a2,a3] becomes [a1,a2,a3,a0]
105
106     // Function RotWord()
107     {
```

```
108         const uint8_t u8tmp = tempa[0];
109         tempa[0] = tempa[1];
110         tempa[1] = tempa[2];
111         tempa[2] = tempa[3];
112         tempa[3] = u8tmp;
113       }
114
115
116       // Function Subword()
117       {
118         tempa[0] = getSBoxValue(tempa[0]);
119         tempa[1] = getSBoxValue(tempa[1]);
120         tempa[2] = getSBoxValue(tempa[2]);
121         tempa[3] = getSBoxValue(tempa[3]);
122       }
123
124       tempa[0] = tempa[0] ^ Rcon[i/Nk];
125     }
126 #if defined(AES256) && (AES256 == 1)
127     if (i % Nk == 4)
128     {
129       // Function Subword()
130       {
131         tempa[0] = getSBoxValue(tempa[0]);
132         tempa[1] = getSBoxValue(tempa[1]);
133         tempa[2] = getSBoxValue(tempa[2]);
134         tempa[3] = getSBoxValue(tempa[3]);
135       }
136     }
```

```
137 #endif
138     j = i * 4; k=(i - Nk) * 4;
139     RoundKey[j + 0] = RoundKey[k + 0] ^ tempa[0];
140     RoundKey[j + 1] = RoundKey[k + 1] ^ tempa[1];
141     RoundKey[j + 2] = RoundKey[k + 2] ^ tempa[2];
142     RoundKey[j + 3] = RoundKey[k + 3] ^ tempa[3];
143   }
144 }
145 
146 void AES_init_ctx(struct AES_ctx* ctx, const uint8_t* key)
147 {
148   KeyExpansion(ctx->RoundKey, key);
149 }
150 #if (defined(CBC) && (CBC == 1)) || (defined(CTR) && (CTR == 1))
151 void AES_init_ctx_iv(struct AES_ctx* ctx, const uint8_t* key,
      const uint8_t* iv)
152 {
153   KeyExpansion(ctx->RoundKey, key);
154   memcpy (ctx->Iv, iv, AES_BLOCKLEN);
155 }
156 void AES_ctx_set_iv(struct AES_ctx* ctx, const uint8_t* iv)
157 {
158   memcpy (ctx->Iv, iv, AES_BLOCKLEN);
159 }
160 #endif
161 
162 
163 static void AddRoundKey(uint8_t round, state_t* state, const
      uint8_t* RoundKey)
```

```
164 {
165   uint8_t i,j;
166   for (i = 0; i < 4; ++i)
167   {
168     for (j = 0; j < 4; ++j)
169     {
170       (*state)[i][j] ^= RoundKey[(round * Nb * 4) + (i * Nb) + j];
171     }
172   }
173 }
174
175
176 static void SubBytes(state_t* state)
177 {
178   uint8_t i, j;
179   for (i = 0; i < 4; ++i)
180   {
181     for (j = 0; j < 4; ++j)
182     {
183       (*state)[j][i] = getSBoxValue((*state)[j][i]);
184     }
185   }
186 }
187
188 static void ShiftRows(state_t* state)
189 {
190   uint8_t temp;
191
192   // Rotate first row 1 columns to left
```

```
193   temp          = (*state)[0][1];
194   (*state)[0][1] = (*state)[1][1];
195   (*state)[1][1] = (*state)[2][1];
196   (*state)[2][1] = (*state)[3][1];
197   (*state)[3][1] = temp;
198
199   // Rotate second row 2 columns to left
200   temp          = (*state)[0][2];
201   (*state)[0][2] = (*state)[2][2];
202   (*state)[2][2] = temp;
203
204   temp          = (*state)[1][2];
205   (*state)[1][2] = (*state)[3][2];
206   (*state)[3][2] = temp;
207
208   // Rotate third row 3 columns to left
209   temp          = (*state)[0][3];
210   (*state)[0][3] = (*state)[3][3];
211   (*state)[3][3] = (*state)[2][3];
212   (*state)[2][3] = (*state)[1][3];
213   (*state)[1][3] = temp;
214 }
215
216 static uint8_t xtime(uint8_t x)
217 {
218   return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
219 }
220
221 static void MixColumns(state_t* state)
```

```
222 {
223   uint8_t i;
224   uint8_t Tmp, Tm, t;
225   for (i = 0; i < 4; ++i)
226   {
227     t   = (*state)[i][0];
228     Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^
            (*state)[i][3] ;
229     Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);
            (*state)[i][0] ^= Tm ^ Tmp ;
230     Tm  = (*state)[i][1] ^ (*state)[i][2] ; Tm = xtime(Tm);
            (*state)[i][1] ^= Tm ^ Tmp ;
231     Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);
            (*state)[i][2] ^= Tm ^ Tmp ;
232     Tm  = (*state)[i][3] ^ t ;              Tm = xtime(Tm);
            (*state)[i][3] ^= Tm ^ Tmp ;
233   }
234 }
235
236 #if MULTIPLY_AS_A_FUNCTION
237 static uint8_t Multiply(uint8_t x, uint8_t y)
238 {
239   return (((y & 1) * x) ^
240        ((y>>1 & 1) * xtime(x)) ^
241        ((y>>2 & 1) * xtime(xtime(x))) ^
242        ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^
243        ((y>>4 & 1) * xtime(xtime(xtime(xtime(x)))))); /* this last
              call to xtime() can be omitted */
244   }
```

```
245 #else
246 #define Multiply(x, y)                          \
247       (  ((y & 1) * x) ^                        \
248       ((y>>1 & 1) * xtime(x)) ^                 \
249       ((y>>2 & 1) * xtime(xtime(x))) ^          \
250       ((y>>3 & 1) * xtime(xtime(xtime(x)))) ^   \
251       ((y>>4 & 1) * xtime(xtime(xtime(xtime(x)))))) \
252
253 #endif
254
255 #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
256
257 #define getSBoxInvert(num) (rsbox[(num)])
258
259 static void InvMixColumns(state_t* state)
260 {
261   int i;
262   uint8_t a, b, c, d;
263   for (i = 0; i < 4; ++i)
264   {
265     a = (*state)[i][0];
266     b = (*state)[i][1];
267     c = (*state)[i][2];
268     d = (*state)[i][3];
269
270     (*state)[i][0] = Multiply(a, 0x0e) ^ Multiply(b, 0x0b) ^
        Multiply(c, 0x0d) ^ Multiply(d, 0x09);
271     (*state)[i][1] = Multiply(a, 0x09) ^ Multiply(b, 0x0e) ^
        Multiply(c, 0x0b) ^ Multiply(d, 0x0d);
```

```
272     (*state)[i][2] = Multiply(a, 0x0d) ^ Multiply(b, 0x09) ^
            Multiply(c, 0x0e) ^ Multiply(d, 0x0b);
273     (*state)[i][3] = Multiply(a, 0x0b) ^ Multiply(b, 0x0d) ^
            Multiply(c, 0x09) ^ Multiply(d, 0x0e);
274   }
275 }
276
277
278 static void InvSubBytes(state_t* state)
279 {
280   uint8_t i, j;
281   for (i = 0; i < 4; ++i)
282   {
283     for (j = 0; j < 4; ++j)
284     {
285       (*state)[j][i] = getSBoxInvert((*state)[j][i]);
286     }
287   }
288 }
289
290 static void InvShiftRows(state_t* state)
291 {
292   uint8_t temp;
293
294   // Rotate first row 1 columns to right
295   temp = (*state)[3][1];
296   (*state)[3][1] = (*state)[2][1];
297   (*state)[2][1] = (*state)[1][1];
298   (*state)[1][1] = (*state)[0][1];
```

```
299    (*state)[0][1] = temp;
300
301    // Rotate second row 2 columns to right
302    temp = (*state)[0][2];
303    (*state)[0][2] = (*state)[2][2];
304    (*state)[2][2] = temp;
305
306    temp = (*state)[1][2];
307    (*state)[1][2] = (*state)[3][2];
308    (*state)[3][2] = temp;
309
310    // Rotate third row 3 columns to right
311    temp = (*state)[0][3];
312    (*state)[0][3] = (*state)[1][3];
313    (*state)[1][3] = (*state)[2][3];
314    (*state)[2][3] = (*state)[3][3];
315    (*state)[3][3] = temp;
316  }
317  #endif // #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB
         == 1)
318
319  // Cipher is the main function that encrypts the PlainText.
320  static void Cipher(state_t* state, const uint8_t* RoundKey)
321  {
322    uint8_t round = 0;
323
324    // Add the First round key to the state before starting the
           rounds.
325    AddRoundKey(0, state, RoundKey);
```

```
326
327   // There will be Nr rounds.
328   // The first Nr-1 rounds are identical.
329   // These Nr rounds are executed in the loop below.
330   // Last one without MixColumns()
331   for (round = 1; ; ++round)
332   {
333     SubBytes(state);
334     ShiftRows(state);
335     if (round == Nr) {
336       break;
337     }
338     MixColumns(state);
339     AddRoundKey(round, state, RoundKey);
340   }
341   // Add round key to last round
342   AddRoundKey(Nr, state, RoundKey);
343 }
344
345 #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB == 1)
346 static void InvCipher(state_t* state, const uint8_t* RoundKey)
347 {
348   uint8_t round = 0;
349
350   // Add the First round key to the state before starting the
        rounds.
351   AddRoundKey(Nr, state, RoundKey);
352
353   // There will be Nr rounds.
```

```
354    // The first Nr-1 rounds are identical.
355    // These Nr rounds are executed in the loop below.
356    // Last one without InvMixColumn()
357    for (round = (Nr - 1); ; --round)
358    {
359      InvShiftRows(state);
360      InvSubBytes(state);
361      AddRoundKey(round, state, RoundKey);
362      if (round == 0) {
363        break;
364      }
365      InvMixColumns(state);
366    }
367
368  }
369  #endif // #if (defined(CBC) && CBC == 1) || (defined(ECB) && ECB
       == 1)
370
371  #if defined(ECB) && (ECB == 1)
372
373
374  void AES_ECB_encrypt(const struct AES_ctx* ctx, uint8_t* buf)
375  {
376    Cipher((state_t*)buf, ctx->RoundKey);
377  }
378
379  void AES_ECB_decrypt(const struct AES_ctx* ctx, uint8_t* buf)
380  {
381    InvCipher((state_t*)buf, ctx->RoundKey);
```

```
382 }
383
384
385 #endif // #if defined(ECB) && (ECB == 1)
386
387
388
389
390
391 #if defined(CBC) && (CBC == 1)
392
393
394 static void XorWithIv(uint8_t* buf, const uint8_t* Iv)
395 {
396   uint8_t i;
397   for (i = 0; i < AES_BLOCKLEN; ++i) // The block in AES is always
        128bit no matter the key size
398   {
399     buf[i] ^= Iv[i];
400   }
401 }
402
403 void AES_CBC_encrypt_buffer(struct AES_ctx *ctx, uint8_t* buf,
      size_t length)
404 {
405   size_t i;
406   uint8_t *Iv = ctx->Iv;
407   for (i = 0; i < length; i += AES_BLOCKLEN)
408   {
```

```
409     XorWithIv(buf, Iv);
410     Cipher((state_t*)buf, ctx->RoundKey);
411     Iv = buf;
412     buf += AES_BLOCKLEN;
413   }
414
415   memcpy(ctx->Iv, Iv, AES_BLOCKLEN);
416 }
417
418 void AES_CBC_decrypt_buffer(struct AES_ctx* ctx, uint8_t* buf,
      size_t length)
419 {
420   size_t i;
421   uint8_t storeNextIv[AES_BLOCKLEN];
422   for (i = 0; i < length; i += AES_BLOCKLEN)
423   {
424     memcpy(storeNextIv, buf, AES_BLOCKLEN);
425     InvCipher((state_t*)buf, ctx->RoundKey);
426     XorWithIv(buf, ctx->Iv);
427     memcpy(ctx->Iv, storeNextIv, AES_BLOCKLEN);
428     buf += AES_BLOCKLEN;
429   }
430
431 }
432
433 #endif // #if defined(CBC) && (CBC == 1)
434
435
436
```

```
437 #if defined(CTR) && (CTR == 1)
438
439
440 void AES_CTR_xcrypt_buffer(struct AES_ctx* ctx, uint8_t* buf,
        size_t length)
441 {
442   uint8_t buffer[AES_BLOCKLEN];
443
444   size_t i;
445   int bi;
446   for (i = 0, bi = AES_BLOCKLEN; i < length; ++i, ++bi)
447   {
448     if (bi == AES_BLOCKLEN) /* we need to regen xor compliment in
            buffer */
449     {
450
451       memcpy(buffer, ctx->Iv, AES_BLOCKLEN);
452       Cipher((state_t*)buffer,ctx->RoundKey);
453
454       /* Increment Iv and handle overflow */
455       for (bi = (AES_BLOCKLEN - 1); bi >= 0; --bi)
456       {
457     /* inc will overflow */
458         if (ctx->Iv[bi] == 255)
459     {
460           ctx->Iv[bi] = 0;
461           continue;
462         }
463         ctx->Iv[bi] += 1;
```

```
464         break;
465       }
466       bi = 0;
467     }
468
469     buf[i] = (buf[i] ^ buffer[bi]);
470   }
471 }
472
473 #endif // #if defined(CTR) && (CTR == 1)
```

2. format of HPCs

```
Number of Instructions: 469764456
Clock Cycles: 8454
Number of cycles the core was active: 29265
Number of load data hazards:29264
Number of jump register data hazards:591
Cycles waiting for instruction fetches:0
Number of data memory loads executed:0
Number of data memory stores executed:1442
Number of unconditional jumps:1068
Number of branches:947
Number of taken branches:1394
Number of compressed instructions executed:968
Number of memory loads to EXT executed:0
Number of memory stores to EXT executed:0
Cycles used for memory loads to EXT:0
```

```
Cycles used for memory stores to EXT:0
Cycles wasted due to TCDM/log-interconnect contention:0
```

3. random combination of data and run(2-D example)

```
1  import subprocess
2
3  # Define the list of file names
4  file_names = ["branch_cnt.txt", "btaken_cnt.txt", "clk_cnt.txt",
       "instr_cnt.txt", "jump_cnt.txt", "ld_cnt.txt", "ls_cnt.txt",
       "rcv_cnt.txt", "st_cnt.txt"]
5
6  # Define the paths
7  path_local =
       "/home/cristiano/Documents/pulp-sdk/tests/qsort/scripts/"
8  path_encrypt_learn =
       "/home/cristiano/Documents/pulp-sdk/tests/qsort
9  /encrypt/parameters_learn/"
10 path_noencrypt_learn =
       "/home/cristiano/Documents/pulp-sdk/tests/qsort
11 /noencrypt/parameters_learn/"
12 path_encrypt_test = "/home/cristiano/Documents/pulp-sdk/tests/qsort
13 /encrypt/parameters/"
14 path_noencrypt_test =
       "/home/cristiano/Documents/pulp-sdk/tests/qsort
15 /noencrypt/parameters/"
16 path_results =
       "/home/cristiano/Documents/pulp-sdk/tests/qsort/results_benign/"
17
18 # Open a log file to record the results
```

```
19 # with open("2_reg_machine_learning_benign_results.log", "w") as
     log_file:
20 with open(path_results +
     "2_reg_machine_learning_benign_results.log", "w") as log_file:
21
22   # Loop through all pairs of files
23   for i in range(len(file_names)):
24     for j in range(i+1, len(file_names)):
25
26       # Print the file names to the log
27       log_file.write(f"{file_names[i]} {file_names[j]}\n")
28
29       # Run the first command : encrypt learn
30       subprocess.check_call([path_encrypt_learn + "2d10000sel_enc",
           path_encrypt_learn + file_names[i], path_encrypt_learn +
           file_names[j]])
31
32       # Run the second command : noencript learn
33       subprocess.check_call([path_noencrypt_learn +
           "2d10000sel_noenc", path_noencrypt_learn + file_names[i],
           path_noencrypt_learn + file_names[j]])
34
35       # Run the third command : encrypt test
36       subprocess.check_call([path_encrypt_test + "2d_test_enc",
           path_encrypt_test + file_names[i], path_encrypt_test +
           file_names[j]])
37
38       # Run the next command
39       subprocess.check_call([path_noencrypt_test + "2d_test_noenc",
```

```
        path_noencrypt_test + file_names[i], path_noencrypt_test
        + file_names[j]])
40
41      # Print the results of the last command to the log
42      result = subprocess.check_output(["python3", path_local +
            "ml_benign.py"])
43      log_file.write(result.decode("utf-8"))
44      log_file.write("\n")
```