Assignment of master's thesis

Title:	Quantum computing methods for malware classification
Student:	Bc. Eliška Krátká
Supervisor:	Aurél Gábor Gábris, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

The presently available near-term intermediate-scale quantum computers (NISQ) have been proven to be more powerful than any existing ordinary supercomputer. While these experiments demonstrated what is termed quantum supremacy, the types of algorithms that were used are not (known to be) related to any practical problem. Fortunately, there exist several algorithms that are suitable to be run on NISQ hardware, and the general expectation is that the capacity of available quantum computers may soon reach the level at which certain problems could be solved faster on them than on classical supercomputers. The aim of the project is to apply quantum computing methods to classify malware using publicly available datasets.

Instructions
1) Study and understand the specified classical and quantum algorithms applied to the malware classification problem
2) Implement a quantum machine learning based malware classification algorithm based on existing literature
3) Train the algorithm on publicly available datasets and evaluate it's performance on IBM quantum computers available by CTU subscription
4) Explore avenues optimising the algorithm, e.g. by considering alternative data preprocessing methods, and design of custom quantum feature maps

Literature:
[1] K. Bharti et al, Noisy intermediate-scale quantum (NISQ) algorithms, Rev. Mod. Phys.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 26 January 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

94, 015004 (2022) [https://doi.org/10.48550/arXiv.2101.08448]
[2] V. Havlíček et al, Supervised learning with quantum-enhanced feature spaces, Nature 567, 209–212 (2019) [https://www.nature.com/articles/s41586-019-0980-2]
[3] Grégoire Barrué, Tony Quertier, Quantum Machine Learning for Malware Classification, arXiv:2305.09674 [https://doi.org/10.48550/arXiv.2305.09674]

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 26 January 2024 in Prague.

Master's thesis

QUANTUM COMPUTING METHODS FOR MALWARE CLASSIFICATION

Bc. Eliška Krátká

Faculty of Information Technology
Department of Information Security
Supervisor: Aurél Gábor Gábris, Ph.D.
May 9, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Bc. Eliška Krátká. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Krátká Eliška. *Quantum Computing Methods for Malware Classification.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments v

Declaration vi

Abstract vii

List of Abbreviations ix

Introduction 1

1 Malware Classification Based on Machine Learning 4
- 1.1 Data Extraction and Preprocessing 4
- 1.2 Support Vector Machine 7

2 Quantum Computing and Quantum Machine Learning 11
- 2.1 Terminology 12
- 2.2 Quantum Machine Learning 14
- 2.3 IBM Quantum Platform 16

3 QSVM for Malware Classification 20
- 3.1 PEML Module 21
- 3.2 SVM Module 22
- 3.3 Implementation of QSVM 23

4 Experiments 29
- 4.1 Experimental Setup 30
- 4.2 Evaluation Metrics 33
- 4.3 Experiment Parameters 34
- 4.4 Findings 34
- 4.5 Discussion 42

5 Conclusion 44

A Benchmark Results on Simulator 46

Bibliography 48

Contents of Attached Media 54

List of Figures

1.1 Visualising Malware as an Image 6
1.2 Visualisation of SVM Decision Boundary and Support Vectors 8

2.1 Relationship Between Probability Amplitude and Classical Probability . 13

3.1 Simplified Project Layout With Implemented Modules and Classes . . . 21
3.2 Error Message on IBM Quantum Platform Due to Lack of Circuit Transpilation 25

List of Tables

2.1 Summary of the Standard Notation in Quantum Mechanics 14

4.1 Accuracy Comparison Between QSVMs With Different Feature Maps With 4 Qubits and Depth 2 36
4.2 Experiment Results: Accuracy Comparison Between QSVMs With Different Feature Maps With 4 Qubits and Depth 2 36
4.3 Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the 4 Qubits Used in QSVM 36
4.4 Experiment Results: Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the 4 Qubits Used in QSVM 37
4.5 Accuracy Comparison Between QSVMs With Different Feature Maps With Depth 2. 37
4.6 F1-score Comparison Between QSVMs With Different Feature Maps With Depth 2. 38
4.7 Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the Number Qubits Used in QSVM. 39
4.8 F1-score Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the Number Qubits Used in QSVM. 40
4.9 Experiment Results: QSVM Classification on IBM Quantum Systems With Various Datasets and Backends 42

A.2 Accuracy Comparison Between QSVMs With Different Feature Maps With Depth 2 [12]. 46
A.4 F1-score Comparison Between QSVMs With Different Feature Maps With Depth 2 [12]. 47

List of Code Listings

3.1 Error Message Due to Job Size Exceeding Maximum Limit on IBM Quantum Platform. 25

4.1 Versions of Used Python Packages 32
4.2 Example Usage of the Classification Scripts 33

I want to express my gratitude to my thesis supervisor, Aurél Gábor Gábris, Ph.D., for his guidance and feedback. I want to thank my partner, family and friends for their support. I especially want to thank my dear friends Linda, Tonda and Patrik because I would probably never have finished this thesis without them.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2024

Abstract

This thesis explores the potential of quantum computing in enhancing malware classification through the application of Quantum Machine Learning (QML). The main objective is to investigate the performance of the Quantum Support Vector Machines (QSVM) algorithm compared to classical approaches for malware classification. A custom Python module was developed to select, extract, and preprocess malware samples from the publicly available PE Malware Machine Learning dataset. The QSVM algorithm, incorporating quantum kernels through different feature maps, was implemented and evaluated on a local simulator within the Qiskit SDK and on real quantum computers from IBM. Experimental results from simulators and quantum hardware provide insights into the behaviour and performance of quantum computers, especially in handling large-scale computations for malware classification tasks. This thesis lays the groundwork for future research in quantum-enhanced malware classification, exploring the feasibility and potential benefits of QML in advancing cybersecurity. The thesis summarizes the practical experience with using quantum hardware via the Qiskit interfaces. We describe in detail the encountered critical issues, as well as the fixes that had to be developed and applied to the base code of the the Qiskit Machine Learning library. These issues include missing transpilation of the circuits submitted to IBM Quantum systems and exceeding the maximum job size limit due to the submission of all the circuits in one job.

Keywords Quantum Support Vector Machines, malware classification, Quantum Machine Learning, IBM Quantum, Qiskit

Abstrakt

Práce zkoumá potenciál kvantových počítačů v oblasti klasifikace malware prostřednictvím aplikace algoritmů kvantového strojového učení (QML). Hlavním cílem je prozkoumat výkon algoritmu Quantum Support Vector Machine (QSVM) ve srovnání s přístupy, nevyužívají kvantové výpočty. Byl vyvinut Python modul pro výběr, extrakci a preprocessing vzorků malware z veřejně dostupného datasetu PE Malware Machine Learning. Byl implementován algoritmus QSVM, který zahrnuje kvantová jádra s různými typy mapování příznaků. Následně byl otestován na lokálním simulátoru z Qiskit SDK a na kvantových počítačích od IBM. Výsledky experimentů poskytují vhled do chování a skutečného výkonu kvantových počítačů, zejména pokud jde o komplexní výpočty typické pro oblast klasifikace malware. Práce pokládá základy pro budoucí výzkum v oblasti klasifikace malware, demonstruje proveditelnost a přínosnost v rozvoji kybernetické bezpečnosti. Stěžejní přínos práce zahrnuje nalezení a nápravu kritických chyb v původní implementaci tříd pro výpočet kvantového jádra v knihovně Qiskit Machine

Learning. Chyby zahrnovaly neprovádění transpilace kvantových obvodů odeslaných ke zpracování kvantovému počítači nebo překročení limitu maximální velikosti výpočetní úlohy v důsledku zpracování všech obvodů v rámci jediné úlohy.

Klíčová slova Quantum Support Vector Machines, klasifikace malware, kvantové strojové učení, IBM Quantum, Qiskit

List of Abbreviations

NISQ	Noisy Intermediate-Scale Quantum
QML	Quantum Machine Learning
SVM	Support Vector Machine
QSVM	Quantum Support Vector Machine
SDK	Software Development Kit
PCA	Principal Component Analysis

Introduction

Malware refers to any software intentionally designed to cause harm to a user [1], for example, by stealing sensitive information or gaining unauthorised access to the system. Malware often infects computer systems without the user's knowledge or consent. It can be distributed through various communication channels, including infected websites, email attachments and removable media such as USB drives [2].

Malware detection is the process of identifying malicious software on a host system or network or distinguishing whether a specific program is malicious or benign. A benign program in the context of malware refers to a program which is harmless and "well-intentioned", the opposite of malicious software [3]. Malware detection is a crucial component of cybersecurity, helping to identify and mitigate the risks posed by malicious software to systems, networks, and data [4]. It is generally considered a binary classification problem of distinguishing between data of two classes, benign and malicious [5]. In the context of this thesis, we mainly refer to the malware detection as the malware classification. However, in a broader context, malware classification usually refers to a process of categorising malware samples into families based on their similarities and attack techniques [6].

Malware analysis is the process of understanding the behaviour and purpose of the malware. Static malware analysis refers to examining the malware sample without executing it. It studies the malware binary file structure and looks for patterns [1]. On the other hand, dynamic analysis involves executing the code. It observes the behaviour of the malicious program and how it interacts with the system [1].

Manual malware analysis involves human analysts examining suspicious files or observing anomalous behaviour within a system to determine if it poses a security risk. Analysts may also reverse-engineer malware to understand its inner workings, which can provide valuable insights for developing new detection techniques [7]. Manual analysis can help to identify previously unknown or sophisticated malware variants that may evade automated detection systems. However, it is a time-consuming process which requires highly skilled analysts who can interpret complex data and patterns effectively. Moreover, the amount of new malware samples generated daily makes relying solely on manual analysis for comprehensive threat detection impractical. While manual analysis is a crucial component of cybersecurity operations, especially for investigating advanced threats, it is often complemented by automated detection methods.

Most antivirus programs use automated detection techniques based on file signatures [1]. A file signature is a unique identifier generated from the content of a file. It can be computed using cryptographic algorithms such as MD5, SHA-1, or SHA-256, which produce a short fixed-size string of bytes (also known as hash). Signature analysis is a method that compares samples with a database of known malware signatures (for example, VirusTotal). If a match is found, it labels the file as potentially harmful. It is a relatively easy and effective method for detecting known malware. However, it has its limitations. For example, it cannot detect malware which is obfuscated (obscured or encrypted malware) or zero-day (malware that exploits vulnerabilities in software or hardware that are not yet known) [1].

The landscape of malware is continuously evolving, and traditional signature-based methods struggle to keep up with the volume and diversity of new threats. That is why modern antivirus programs also leverage machine learning to identify new and unknown threats [8, 9]. Malware detection based on machine learning algorithms offers a solution to the challenges in malware detection because it can detect previously unseen malware based on learned experience [1].

In recent years, advancements in quantum computing, particularly with the development of Noisy Intermediate-Scale Quantum (NISQ) devices, have opened up new possibilities for addressing complex computational problems that classical computers struggle to solve [10]. Quantum computers leverage the principles of quantum mechanics, such as superposition and entanglement, which allow them to perform parallel calculations and exponentially increase their processing power for specific problems. The combination of quantum computing and machine learning, Quantum Machine Learning (QML), is a promising research area on the presently available NISQ devices [10].

This thesis explores the potential of quantum computing, specifically QML, in malware detection. Over the last decade, there have been significant advances in the QML field, including classical machine learning algorithms that can be enhanced using quantum techniques and entirely new quantum machine learning algorithms designed to run on quantum computers [11]. Inspired by the work of [12], we focus on the Quantum Support Vector Machines (QSVM) algorithm.

QSVM algorithm combines the classical machine learning algorithm used in classification problems, Support Vector Machine (SVM) and a quantum kernel, which exploits quantum properties like superposition and entanglement [12]. In SVM, the kernel is a function that computes the similarity between pairs of data points in a higher-dimensional space and allows SVM to classify non-linearly separable data effectively [13]. We study the quantum kernels, which are based on the concept of quantum feature maps, introduced in [14]. Quantum kernels promise to outperform classical kernels by addressing challenges posed by high-dimensional feature spaces [14].

We implement the QSVM algorithm and use it to classify malware samples from a publicly available dataset on the real NISQ computer from IBM [15] and the local quantum computer simulator implemented in Qiskit SDK [16]. We evaluate the performance of our algorithm using a comparison with classical SVM kernels and with the results from the study presented in [12]. We structure our research into four parts, each described in one of the four chapters of this thesis.

In the first chapter, we start by introducing malware classification based on machine

learning. We explain why the data extraction and preprocessing steps are needed before putting the data into the machine learning algorithm. We specifically focus on the methods we use for work with our chosen dataset, the PE Malware Machine Learning dataset. Our dataset consists of binary samples of malicious and benign PE files. We describe the PE files and their structure and introduce the preprocessing method we use to extract features from the files before submitting them to the QSVM algorithm. Additionally, in the first chapter, we describe the classical SVM algorithm with a focus on the kernel function.

In the second chapter, we focus on the quantum computing part of our research. We provide an introduction to quantum computing and the terminology we use and then focus on the QSVM algorithm. We describe how the quantum kernels differ from the classical ones and how they are estimated on quantum computers. We then move to the IBM Quantum Platform [15] and Qiskit SDK [16], tools provided by IBM that we use during our research for the implementation of the algorithm and for access to quantum computers from IBM.

In the third chapter, we describe our implementation. We implement two independent Python modules, one for the data preprocessing and the second one for the QSVM classification. We describe how we implement the quantum kernels and the interface for the classification on IBM Quantum computers. We describe the challenges encountered during the implementation and how we fix the bugs in the Qiskit Machine Learning library.

Lastly, in the fourth chapter, we perform the experiments on both the simulator and a real quantum computer from IBM. We describe the purpose of the experiments and our findings and discuss the optimisation techniques we plan to explore in future in our following research.

Chapter 1

Malware Classification Based on Machine Learning

In computer science, the classical algorithm is a procedure for solving a problem consisting of a finite number of clearly defined steps. Algorithms are explicitly programmed to solve specific problems based on predetermined rules and logic. They follow a deterministic approach of executing predefined instructions to perform tasks. [17]

Machine learning is a field of computer science that focuses on developing algorithms that solve problems through learning from data. In contrast to classical algorithms, machine learning algorithms are not explicitly programmed to execute step-by-step instructions based on data input. They are designed to learn how to solve problems based on input data by identifying patterns, making predictions, and optimizing their performance.

In this chapter, we explore the application of machine learning in malware classification, a critical task in cybersecurity. While various machine learning algorithms have been applied to this problem domain [18], we emphasise the Support Vector Machine algorithm due to its widespread use and effectiveness in handling high-dimensional data and non-linear decision boundaries.

We begin by examining selected data extraction and preprocessing techniques relevant to our research, laying the groundwork for subsequent analysis. We then shift our focus to the introduction of the Support Vector Machine algorithm, where we explore its principles and functionalities. A particular emphasis is placed on the role of kernel functions, which enable the Support Vector Machine algorithm to operate effectively in high-dimensional feature spaces.

1.1 Data Extraction and Preprocessing

Data extraction and preprocessing are the initial steps in the machine learning workflow. In this stage, raw binary files representing executable programs across various platforms and architectures are transformed into structured data suitable for machine learning algorithms. Directly feeding raw binary files into machine learning algorithms is

impractical due to their unstructured nature and the volume of data. Unstructured data lacks the organisation and formatting necessary for practical analysis, and the amount of information in raw binary files makes it challenging for machine learning algorithms to extract meaningful patterns.

The feature extraction process is crucial because it pulls only the relevant information from binary files to construct informative feature vectors. The feature extraction can be done through static or dynamic analysis techniques, depending on the types of features. Static analysis involves examining the structure of binary files, such as header fields, section names, and import libraries, without executing the code [1]. Dynamic analysis involves executing the code within a controlled environment to observe its behaviour and interactions with the system [1].

The selection of preprocessing techniques significantly impacts the quality and effectiveness of machine learning models. Our discussion centres on the conversion of the binary samples into grayscale images. The method is chosen for its simplicity and effectiveness in capturing essential characteristics of binary files [19]. We discuss subsequent preprocessing steps, such as Principal Component Analysis, a dimensionality reduction technique that retains key information while reducing the complexity of the dataset. [1, 20]

PE File Format

A PE file stands for Portable Executable file, a binary architecture-independent file format used in Microsoft Windows operating systems for executable files, object code, and DLLs (Dynamic Link Libraries). PE files contain information about how a program should be loaded and executed, such as code, data, resources, and metadata the operating system loader requires. The significance of the PE file format in malware detection lies in its role as the primary format for executable files in the Microsoft Windows operating system. Statistics from AV-TEST, an independent agency that evaluates and rates antivirus and security software, show that Windows is the most targeted operating system by malware attacks [21]. Malware often disguises itself as legitimate software, therefore understanding the structure and contents of PE files is crucial for detecting and analyzing malicious programs.

PE files have a specific fixed structure, which consists of two main parts: header and sections [22]. The header contains information about the functions and sizes of structures in the file. It consists of the DOS header, the PE signature (PE magic number, the Common Object File Format (COFF) file header, the optional header, and the section headers. The DOS header contains the MZ signature, a two-byte marker indicating the DOS executable format. The PE signature, consisting of the characters 'P' and 'E' followed by two null bytes, marks the beginning of the Portable Executable format. These signatures are crucial for identifying and validating the PE file format. The sections encapsulate information needed to manage executable code, such as code, data, resources, and other elements required for program execution. Each section typically serves a specific purpose, such as storing executable instructions, initialized data, uninitialized data, or resources like strings. Sections provide the necessary granularity for the operating system loader to map the contents of the PE file into memory and

manage its execution efficiently.

Grayscale Image Conversion

In the context of preprocessing malware samples for machine learning, grayscale image conversion refers to transforming raw PE binary samples into grayscale representations. The method is simple, effective and does not require code execution or disassembling for the classification while also appearing resilient to obfuscation techniques such as encryption [19]. The motivation behind adopting this method in malware detection and classification stemmed from recognising that malware binaries from the same family appear very similar in layout and texture [19].

Figure 1.1 shows the process of visualizing malware as an image. The PE file samples are essentially binary files composed of zeros and ones. Each byte or group of bytes within the binary data corresponds to a pixel in the grayscale image, with pixel intensity determined by the value of the byte(s). For example, a byte with a value of 0 translates to a black pixel, while a byte with a value of 255 signifies a white pixel. These binary data, organised as 1D vectors, can be reshaped into 2D matrices and viewed as grayscale images. The resulting size of these images can be either fixed or dynamically adjusted based on the length of the binary data. Once transformed, further applied image processing techniques can extract features such as texture, shape, or spatial distribution patterns within the image. These extracted features or raw images can serve as input for training ML models for malware classification.

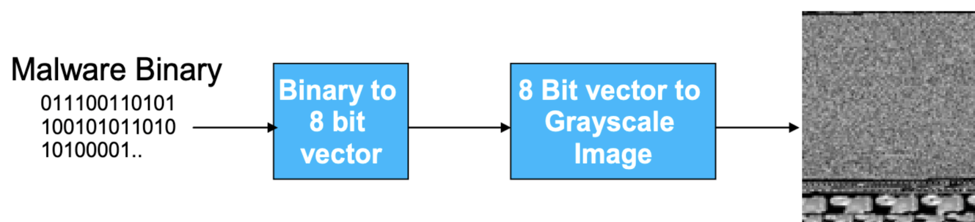

Figure 1.1 Visualising Malware as an Image [19].

Principal Component Analysis

Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of large datasets while retaining most of the original information [23]. PCA effectively extracts the most informative features from the data by transforming correlated variables into a smaller set of variables known as principal components. This preprocessing step is particularly useful for machine learning algorithms, as it helps mitigate the 'curse of dimensionality' by reducing model complexity [23]. The curse of dimensionality refers to the negative impact on model performance caused by adding each new feature. Through projection into a smaller feature space, PCA addresses issues such as overfitting, which can arise in high-dimensional datasets. PCA finds applications in various domains, for example, in image processing and pattern recognition.

1.2 Support Vector Machine

In this section, we delve into the Support Vector Machine (SVM) algorithm, a widespread tool in the domain of machine learning-based malware classification [18]. Our discussion provides an overview of the principles and functionalities of SVM, with a particular emphasis on kernel functions. Kernel functions play an essential role in SVM by enabling the algorithm to operate effectively in high-dimensional feature spaces and capture complex relationships between the data samples and their features. The insights presented in this section are based on the works of [13], [24], [25], and [14], which have significantly contributed to our understanding of SVM and the importance of kernel functions.

Support Vector Machine (SVM) is a supervised machine learning algorithm for classification and regression tasks. In SVM classification, the algorithm seeks to find an optimal decision boundary within the feature space that separates the data points into different classes. Once the decision boundary is established, new data points can be classified by determining which side of the boundary they fall on.

The feature space refers to the space defined by the input features of the dataset. Each data point in the dataset is represented as a feature vector in the feature space, where the total dimension corresponds to the number of features in the vector. Each element in the vector represents a feature of the dataset, and the dimension of the feature space corresponds to a number of features in the vector. The decision boundary is a conceptual line that separates the different classes in a classification problem. In binary classification, it is typically a line or curve in two-dimensional space or a hyperplane in higher-dimensional space that divides the feature space into regions corresponding to each class in the classification problem.

The core principle of SVM is to find a hyperplane that best separates the data into different classes while maximising the margin, which refers to the distance between the hyperplane and the nearest data point from each class. In other words, there is the widest possible margin on both sides of the hyperplane without any points. The hyperplane is characterised by the support vectors, which are data points lying closest to it. The support vectors are crucial because they directly influence the position and orientation of the hyperplane. Moreover, they define the margin because the optimal hyperplane is positioned to maximise the distance between them. Support vectors play a significant role in SVM because they determine the effectiveness of the classifier and its ability to generalise to unseen data. Figure 1.2 visually represents the decision boundary, maximised margin, and support vectors.

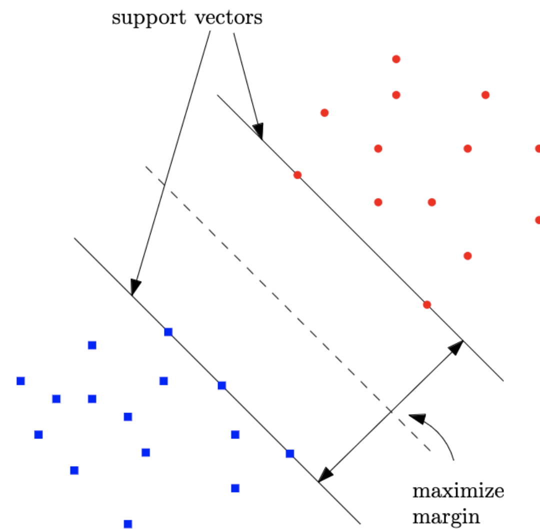

Figure 1.2 Visualisation of SVM Decision Boundary and Support Vectors [26].

SVM can handle both linearly separable and non-linearly separable data. Linearly separable data refers to a scenario where two classes of data points can be perfectly separated by at least one straight line (in two dimensions), plane (in three dimensions) or hyperplane (in higher dimensions). Many real-world datasets are not inherently linearly separable, which is why techniques such as kernel functions are utilised in SVM. When we talk about transforming the feature space using a kernel function in SVM, we are essentially mapping the input features to a new, possibly higher-dimensional space where the data may become linearly separable or more easily separable by a hyperplane. This transformation allows SVM to find complex decision boundaries that may not be feasible to compute directly in the original feature space.

A feature map $\phi(x)$ is a function which explicitly transforms the input data into a higher-dimensional space. The function maps each data point x from the original feature space to a new transformed feature space with a higher dimensionality. Kernel function, defined as

$$k(x, y) = (\phi(x) \cdot \phi(y)), \tag{1.1}$$

uses the feature map to compute the dot product between two vectors x and y from the input data space. It refers to the distance between the data points (vectors x and y) in the original input space.

However, the transformed feature space is high-dimensional, so the righthand side of the equation (1.1) may be computationally expensive (especially if the dimensionality of the feature space is very large or even infinite). There comes a method known as the kernel trick. Instead of explicitly computing the transformed feature vectors $\phi(x)$ and $\phi(y)$, the kernel trick allows us to compute $k(x, y)$ directly from the original input space without ever explicitly computing $\phi(x)$ and $\phi(y)$. The implicit transformation into higher-dimensional spaces facilitated by the kernel function is a key aspect of the kernel trick, enabling SVM to handle non-linear relationships in the data efficiently.

For example, consider a linear kernel function

$$k(x, y) = x \cdot y, \tag{1.2}$$

where x and y are the input vectors in the original feature space and $\cdot$ denotes the dot product operation. Consider a dataset with two-dimensional input vectors $x = (x_1, x_2)$ and $y = (y_1, y_2)$ and take two data points $x = (1, 2)$ and $y = (3, 4)$. To compute the linear kernel function $k(x, y)$, we take the dot product of the two vectors

$$k(x, y) = (1 * 3) + (2 * 4) = 11. \tag{1.3}$$

In the original feature space, the data points x and y are represented by their individual components. However, if we were to transform these data points into a higher-dimensional space using a feature map for a linear kernel, the feature map would be the identity function

$$\phi(x) = x \tag{1.4}$$

$$\phi(y) = y. \tag{1.5}$$

In the context of the linear kernel, the output of the kernel function is equivalent to computing the dot product of the original input vectors x and y without any explicit transformation into a higher-dimensional feature space.

Various kernel functions, such as polynomial, RBF, and sigmoid kernel, are suitable for different data types and classification tasks. Kernel functions play a crucial role in SVM by enabling it to handle non-linear relationships between the input data. First, the kernel function $k(x, y)$ is applied to all pairs of training samples x_i, x_j to compute the kernel matrix

$$K_{ij} = k(x_i, x_j). \tag{1.6}$$

The kernel matrix represents the pairwise distances between the training samples in the original feature space. Next, SVM uses the kernel function to solve an optimisation problem of finding the optimal hyperplane that maximises the margin between the classes.

Various kernel functions, such as polynomial, RBF, and sigmoid kernels, are employed in SVM to handle different data types and classification tasks. These kernel functions play a crucial role in SVM by enabling it to capture non-linear relationships between input data effectively. During training, SVM computes the kernel function for all pairs of training samples to construct a kernel matrix, which represents pairwise distances in the original feature space. Next, SVM uses the kernel function to solve the optimisation problem of finding the optimal hyperplane that maximises the margin between the classes. The problem is described by the decision function

$$f(x) = \operatorname{sgn}\left(\sum_{i=1}^{n} y_i \alpha_i k(x, x_i) + b\right) \tag{1.7}$$

where n is the number of samples in the dataset, α_i are the Lagrange multipliers, y_i are the labels of the training samples, and b is a constant, which shifts the decision

boundary away from the origin $(0, 0)$ of the coordinate system in the feature space [13]. During the training phase of SVM, b is optimised along with the Lagrange multipliers α_i. Only the samples with non-zero Lagrange multipliers α (support vectors) contribute to the decision boundary. In order to classify a new data point x, its similarity with the support vectors is computed using the kernel function $k(x, x_i)$, and the decision function $f(x)$ is evaluated. If $f(x)$ is positive, the data point is classified as one class, and if it is negative, it is classified as the other class.

Chapter 2

Quantum Computing and Quantum Machine Learning

Quantum computing is a field that leverages the principles of quantum mechanics to perform computations. In recent years, quantum computing has emerged as a promising tool for solving complex computational problems intractable to classical computers with the development of NISQ (Noisy Intermediate-Scale Quantum) devices. NISQ devices are the class of quantum computers characterised by their intermediate scale. Unlike universal fault-tolerant quantum computers, which are still a theoretical goal, NISQ devices operate with a limited number of qubits and suffer from errors due to noise in the quantum hardware [10]. They typically have tens to hundreds of qubits, larger than what can be simulated classically but smaller than required for error correction and fault tolerance [10]. Nevertheless, they still offer the potential for exponential speedups over classical algorithms in various domains, including optimisation, cryptography, and machine learning [10].

The intersection of quantum computing and machine learning, Quantum Machine Learning (QML), has gained attention for its ability to leverage quantum computational advantages to enhance traditional machine learning algorithms. This chapter introduces quantum computing and its application to machine learning, focusing on the Quantum Support Vector Machine (QSVM) algorithm. We provide an overview of quantum computing fundamentals, explaining the core concepts and principles underpinning quantum computation, such as quantum bits (qubits), superposition and entanglement. We examine the theoretical foundations of QSVM, describe how they operate on quantum computers and the advantages they offer over their classical counterparts.

Furthermore, we introduce the IBM Quantum Platform, which is a leading platform for quantum computing research and development, along with the Qiskit software development kit (SDK) and its machine learning module. Through Qiskit, researchers can develop quantum algorithms and access IBM Quantum hardware. We discuss the role of Qiskit in our research in implementing QSVM and performing quantum experiments on real quantum processors.

2.1 Terminology

We rely on a [27] for definitions and explanations of key terms. The only prerequisite is a basic understanding of elementary linear algebra and classical computing.

Quantum computing operates within a finite-dimensional Hilbert space $\mathcal{H}$. In this context, the Hilbert space equals a complex vector space $\mathbb{C}^n$ with an inner product. The inner product is formally a map

$$\langle \cdot, \cdot \rangle : V \times V \to \mathbb{C}, \tag{2.1}$$

where V is a vector space over $\mathbb{C}$, which satisfies the following three properties for all vectors $x, y, z \in V$ and all scalars $\alpha \in \mathbb{C}$:

1. $\langle x \,|\, \alpha y + z \rangle = \alpha \langle x \,|\, y \rangle + \langle x \,|\, z \rangle$ *(linearity in the second argument),*
2. $\langle x \,|\, y \rangle = \langle y \,|\, x \rangle^*$ *(conjugate symmetry),*
3. $\langle x \,|\, x \rangle \geq 0$ with equality if and only if $|x\rangle = 0$ *(positive definiteness),*

where $*$ is a complex conjugate and 0 is a zero vector [28]. The standard notation for linear algebra in quantum mechanics and quantum computing is a braket notation, which consists of two elements, bra and ket. The ket, written as $|\psi\rangle$, denotes a vector in the vector space. The bra, written as $\langle \psi|$, represents a dual vector to the ket. The inner product of two vectors $|\psi\rangle$ and $|\varphi\rangle$ is denoted by $\langle \varphi \,|\, \psi \rangle$.

A quantum bit, shortly qubit, serves as the fundamental unit of information in quantum computing. While classical computing processes information using bits, which are binary variables capable of holding values 0 or 1, quantum computing leverages qubits.

A state of the qubit, the quantum state, is described by a unit vector in a two-dimensional Hilbert Space. The states $|0\rangle$ and $|1\rangle$ denote the fundamental computational basis states of the qubit, forming an orthonormal basis. Any quantum state of the qubit can be expressed as a linear combination of $|0\rangle$ and $|1\rangle$, meaning a qubit can exist in a superposition of these states. For example, the state

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{2.2}$$

represents the qubit in the superposition of $|0\rangle$ and $|1\rangle$.

The complex numbers α and β referred to as probability amplitudes satisfy

$$|\alpha|^2 + |\beta|^2 = 1. \tag{2.3}$$

They encode the probability of each outcome and the associated phase information. In contrast to a classical probability distribution, which only considers the real numbers, probability amplitudes incorporate both magnitude and phase. Figure 2.1 visually represents the relationship between probability amplitude and classical probability. The absolute squares of the probability amplitudes give the probabilities of the possible outcomes occurring when measured in the computational basis.

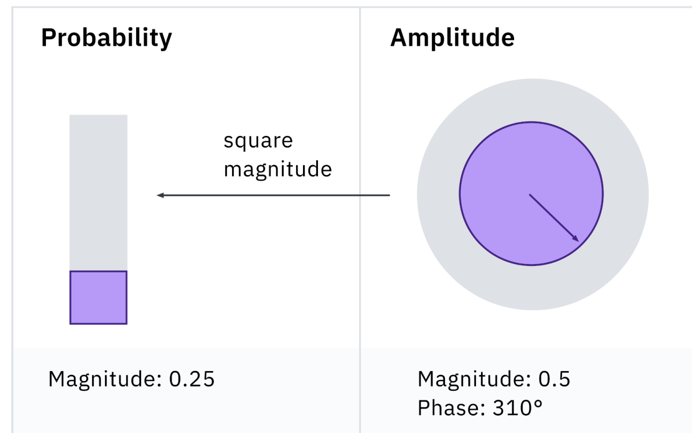

Figure 2.1 Relationship Between Probability Amplitude and Classical Probability [29].

Measurement plays an essential role in quantum computing. While a classical bit's state can be observed without altering it, qubits in superposition cannot be directly measured without affecting their quantum state. Upon measurement, qubits 'collapse' into one of the basis states, giving the outcome either $|0\rangle$ with probability $|\alpha|^2$ or $|1\rangle$ with the probability $|\beta|^2$. Consequently, quantum states inherently embody non-determinism, as their measurement is probabilistic and fundamentally different from classical systems.

The building blocks of quantum computing are quantum gates and circuits. Quantum gates are basic operations that manipulate qubits, similar to classical logic gates. They come in various types, such as single-qubit and two-qubit gates, each designed to perform specific transformations on quantum states. Quantum gates are reversible transformations, preserving the quantum information encoded in qubits. In quantum computing, quantum gates are represented by unitary operators. Unitary operators are mathematical operators represented by matrices that satisfy the condition

$$U^\dagger U = I, \tag{2.4}$$

where $U^\dagger$ is the adjoint (conjugate transpose) of U, and I is the identity matrix.

Quantum circuits are composed of sequences of quantum gates applied to qubits to perform specific computational tasks. Just as classical circuits are constructed from interconnected logic gates, quantum circuits are built by connecting quantum gates. They describe the flow of information and operations in a quantum computation. Quantum circuits are visually represented with qubits depicted as lines and quantum gates as boxes or symbols acting on those lines. The connections between gates indicate which qubits are involved in each operation and in what order they are applied.

Within quantum circuits, interference emerges as a fundamental phenomenon where the probability amplitudes of different quantum states combine and interact. Transition amplitudes describe the probability amplitude for a qubit to transition from one quantum state to another under the influence of a quantum gate or operation. In quantum algorithms, transition amplitudes are manipulated by applying quantum gates in a quantum circuit. By carefully designing the sequence of gates, the interference effects can be exploited to enhance the probability of obtaining the desired output state while

minimizing the probability of undesired outcomes. The interference can be constructive, where probability amplitudes increase the probability of a particular outcome, or destructive, where probability amplitudes cancel each other out, reducing the probability of certain outcomes. The ability to control transition amplitudes is a key feature that enables quantum computers to solve specific problems more efficiently than classical computers.

State overlap and operator fidelity play a crucial role in quantifying the similarity between quantum states. State overlap quantifies the extent to which two quantum states share common elements or characteristics, providing insight into their similarity. Operator fidelity quantifies the accuracy of a quantum operation or transformation by measuring the closeness between the input and output states. Maximizing fidelity ensures the reliability and effectiveness of quantum algorithms, enhancing their computational performance and accuracy.

Entanglement stands as another unique feature of quantum computing. Entangled qubits are interconnected so that the state of one qubit depends on the state of another, regardless of the distance between them. Alongside superposition and interference, entanglement empowers quantum computers to perform parallel calculations and exhibit non-local correlations, exponentially increasing processing power for certain problem domains.

Furthermore, complementing the introduction on quantum computing, Table 2.1 summarises the notation presented in this section.

Notation	Description
z^*	Complex conjugate of the complex number z.
$\|\psi\rangle$	Vector. Also known as a *ket*.
$\langle\psi\|$	Vector dual to $\|\psi\rangle$. Also known as a *bra*.
$\langle\varphi \| \psi\rangle$	Inner product between the vectors $\|\varphi\rangle$ and $\|\psi\rangle$.
A^*	Complex conjugate of the A matrix.
A^T	Transpose of the A matrix.
$A^\dagger$	Adjoint of the A matrix, $A^\dagger = (A^T)^*$.
U	Unitary operator.

Table 2.1 Summary of the Standard Notation in Quantum Mechanics [27].

2.2 Quantum Machine Learning

Quantum machine learning (QML) is an interdisciplinary field that combines principles from quantum computing and machine learning. It explores the potential benefits of using quantum algorithms and quantum computing hardware to enhance various aspects of machine learning tasks, including data processing, pattern recognition, optimisation, and predictive modelling [10].

In the first chapter, we laid the groundwork for understanding the Support Vector Machine (SVM) algorithm and its role in machine learning, particularly in classification problems. Building upon that foundation, we now focus on a quantum advancement of

the classical SVM, the Quantum Support Vector Machine algorithm (QSVM). We explore the QSVM algorithms, leveraging the principles of quantum mechanics to enhance the capabilities of SVM for classification problems.

Recent advancements in quantum computing, as discussed in [14], have introduced the concept of quantum advantage in QSVM over classical SVM approaches. The advantage hinges on exploiting quantum properties such as entanglement and superposition, which classical computers cannot simulate efficiently. While estimating the complexity of quantum kernels using classical methods may pose challenges, the unique quantum properties enable QSVM to outperform classical SVM.

By integrating quantum computing techniques with SVM algorithms, QSVM promises to achieve better accuracy and efficiency than classical SVM algorithms across various problem domains, including malware detection [12]. We investigate how quantum algorithms can optimise SVM performance and address the challenges posed by high-dimensional feature spaces. Ultimately, our goal is to provide a comprehensive understanding of the QSVM algorithm and its potential impact on malware classification.

QSVM

The QSVM algorithm is presented in two versions in [14]. The first version resembles the classical SVM but operates on a quantum computer using a variational quantum circuit, a type of quantum circuit used for optimization tasks to implement the quantum classifier. We focus on the second version, which combines a classical SVM classifier with a quantum kernel function. The quantum kernel function is estimated on a quantum computer to compute the kernel matrix for use in the conventional SVM.

The key difference between classical and quantum kernels lies in how the data are processed. In a classical kernel, the data are processed directly in the original form within the classical computational framework. The kernel function computes the dot product between feature vectors in the original input space. This computation is done explicitly, without any transformation of the data into a different space.

In contrast, the quantum kernel requires the data to be transformed into a quantum state space $\mathcal{H}$ before being processed. The data transformation leverages principles of quantum mechanics such as entanglement and interference. In the context of QSVM, we refer to this transformation as a data encoding. Once the data are encoded, a quantum kernel function is applied to compute the correlations between the quantum states. The data encoding process allows the quantum kernel to generate correlations between variables that are difficult to achieve using classical methods alone. The advantage of quantum kernels lies in their ability to construct complex circuits that are hard to compute classically. [14, 11]

The estimation of the kernel matrix using a quantum kernel involves two main components: the encoding of classical data and the application of the quantum kernel function.

The data encoding process is done through a quantum feature map, denoted as $\phi(x)$, which represents a parametrised quantum circuit that maps classical feature vectors x to the corresponding quantum state $|\phi(x)\rangle\langle\phi(x)|$. The mapping is done by applying the unitary operation $U_{\phi(x)}$ to the initial state $|0^n\rangle$, where n is the number of qubits

used for encoding. The index $\phi(x)$ in the $U_{\phi(x)}$ refers to a specific parameterization of the operation U, which depends on the classical feature vector x. Quantum gates and operations can be parametrized by certain variables, affecting how they transform quantum states. Different values of x lead to different parameterizations of the unitary operation, resulting in different quantum states after the transformation. [30]

The quantum kernel function

$$k(x, y) = \langle \phi(x) \,|\, \phi(y) \rangle = |\langle \phi(x) \,|\, \phi(y) \rangle|^2, \tag{2.5}$$

is defined as a state overlap of the two data-encoded feature vectors from the quantum state space and represents the similarity between them [14]. A larger value of $k(x, y)$ indicates that the classical data points x and y are close in feature space [30].

When applied to all datapoints, quantum kernel function generates a quantum kernel matrix

$$K_{i,j} = k(x_i, x_j) = |\langle \phi(x_i) \,|\, \phi(x_j) \rangle|^2, \tag{2.6}$$

where the entries represent the fidelities between different feature vectors. The fidelities can be computed efficiently on a quantum computer by calculating the transition amplitude between the states

$$K_{i,j} = k(x_i, x_j) = |\langle \phi(x_i) \,|\, \phi(x_j) \rangle|^2 = |\langle 0^n | U^{\dagger}_{\phi(x_i)} U_{\phi(x_j)} | 0^n \rangle|^2, \tag{2.7}$$

where the feature map $\phi(x)$ is described as the unitary operation $U_{\phi(x)}$ applied to the initial state $|0^n\rangle$. [14, 30]

2.3 IBM Quantum Platform

In our quantum computing research, we heavily rely on the comprehensive suite of tools provided by IBM, such as the Qiskit SDK and the real quantum computers available through the cloud. IBM plays a significant role in advancing the field of quantum computing through research, development, and the provision of accessible tools and resources. They have made significant contributions to the development of quantum hardware, including the design and fabrication of superconducting qubits, the building blocks of quantum processors. They developed the first commercially available quantum computer, IBM Q System One. While the development of the IBM Q System One marked a significant milestone, their broader efforts in advancing quantum hardware technology are equally noteworthy.

IBM provides access to real quantum processors, known as IBM Quantum systems, through the cloud via the IBM Quantum Platform [15], allowing researchers and developers to experiment with real quantum hardware without needing specialized infrastructure. As of April 2024, 15 quantum processors and five simulators are available on the IBM Quantum Platform. Four quantum processors and all simulators are freely available to the public, while the remainder is accessible via a premium plan. Additionally, the platform offers learning resources to help users understand the principles of quantum computing and how to leverage IBM's tools effectively.

IBM is developing Qiskit [16], an open-source software development kit (SDK) for

working with quantum circuits and algorithms. It allows users to create, simulate, and execute programs for quantum computers using Python, a widely used high-level programming language. Python is known for its simplicity and readability, which makes it an excellent choice for a framework like Qiskit, which aims to be accessible to both beginners and experienced users in the quantum computing field. Qiskit allows users to implement algorithms for quantum computers at the level of quantum circuits. These algorithms can be executed locally on simulators or real quantum devices available through the IBM Quantum Platform.

Qiskit Machine Learning

Qiskit Machine Learning [31] is a module within the Qiskit SDK, which provides a comprehensive set of tools for quantum-enhanced machine learning tasks, including classical machine learning algorithms that can be enhanced using quantum techniques and entirely new quantum machine learning algorithms designed to run on quantum computers.

It offers building blocks such as Quantum Kernels and Quantum Neural Networks, allowing users to explore the intersection of quantum computing and machine learning without requiring deep quantum computing knowledge. This section focuses mainly on introducing the Quantum Kernels within the Qiskit Machine Learning module, specifically on the `FidelityQuantumKernel` [32]. Understanding the functionality and usage of the `FidelityQuantumKernel` [32] class is essential for effectively integrating quantum-based kernels into the Support Vector Machine algorithms.

The quantum kernel interface is abstractly defined by the `BaseKernel` class [33] in Qiskit Machine Learning. It specifies the `evaluate` method, which constructs kernel matrices from given datasets. These matrices are compatible with the Quantum Support Vector Classifier within Qiskit Machine Learning or other kernel-based machine learning algorithms in established classical frameworks. Each entry in the kernel matrix is the result of the kernel function defined as

$$K(x, y) = \langle f(x) \,|\, f(y) \rangle, \tag{2.8}$$

where x, y are n-dimensional inputs and f is a map from n-dimension to m-dimension space. The quantum kernel algorithm computes a kernel matrix given datapoints x and y and feature map f, all of n dimension. This matrix can then be used in classical machine learning algorithms such as support vector classification. A feature map, a parameterized circuit serving as input to the kernel function, is required. The default feature map is a `ZZFeatureMap` with two qubits.

The `FidelityQuantumKernel` implements the `BaseKernel` interface. Here, the kernel function is defined as the overlap of two quantum states x and y),

$$K(x, y) = |\langle \phi(x) \,|\, \phi(x) \rangle|^2, \tag{2.9}$$

defined by the feature map $\phi(x)$.

The `FidelityQuantumKernel` requires a fidelity primitive. This primitive computes the fidelity between quantum states based on the `BaseStateFidelity` algorithm in-

troduced in Qiskit. `BaseStateFidelity` class [34] is an interface that calculates state fidelities (state overlaps) for pairs of (parametrized) quantum circuits. The specific method of fidelity calculation depends on the implementation of the fidelity method. However, it can generally be defined as the state overlap

$$|\langle\psi(x)\,|\,\phi(x)\rangle|^2, \tag{2.10}$$

where x and y are optional parameterizations of the states ψ and ϕ. The default fidelity value is an instance of the `ComputeUncompute` class built on top of the `Sampler` primitive [35].

There are numerous options for selecting a suitable feature map $\phi(x)$ for the computation of `FidelityQuantumKernel`, as described in [14]. We introduce those implemented in Qiskit SDK, which we later use in our experiments, such as `PauliFeatureMap` [36], `ZZFeatureMap` [37] and `ZFeatureMap` [38].

The `PauliFeatureMap` is based on the Pauli matrices, which are fundamental operators in quantum mechanics. The Pauli matrices include the X, Y and Z matrices, each representing a different type of quantum operation. In the `PauliFeatureMap`, combinations of these matrices, set by the `paulis` parameter, are applied to the input qubits to generate entanglement and capture features of the input data. The `PauliFeatureMap` typically consists of layers of single-qubit rotations and entangling gates involving Pauli matrices, with parameters that can be optimized during training to learn an adequate representation of the data for classification tasks. The data encoding is done by applying the unitary operation $U_{\phi(x)}$ to the initial state, which in the case of `PauliFeatureMap` is defined as

$$U_{\phi(x)} = \exp\left(i \sum_{S \in \mathcal{I}} \phi_S(x) \prod_{i \in S} P_i\right), \tag{2.11}$$

where S is a set of qubit indices that describes the connections in the feature map, $\mathcal{I}$ is a set containing all these index sets, P_i refers to the chosen Pauli matrix and

$$\phi_S(x) = \begin{cases} x_i & \text{if } S = \{i\} \\ \prod_{j \in S}(\pi - x_j) & \text{if } |S| > 1 \end{cases} \tag{2.12}$$

refers to the data mapping function. The data mapping function can be changed to a custom one via the parameter `data_map_func`. [36]

The `ZZFeatureMap` is a special case of the PauliFeatureMap, where the 'ZZ' refers to the Pauli-Z matrices. These matrices represent the ZZ interaction between qubits, contributing to the entanglement in the quantum circuit. In the `ZZFeatureMap`, the Pauli matrices P_i are explicitly chosen as Pauli-Z matrices (ZZ), resulting in the product term representing the ZZ interaction between qubits. [37]

The `ZFeatureMap` is another particular case of the `PauliFeatureMap`. In contrast to the `ZZFeatureMap`, it consists solely of Pauli Z matrices without entangling operations between qubits. As a result, the encoding produced by the `ZFeatureMap` does not exhibit entanglement. While this lack of entanglement means that the `ZFeatureMap` may not offer a quantum advantage for specific tasks, its effectiveness depends on the problem being addressed. [38]

All the mentioned feature maps can have a custom circuit depth set by the 'depth'

parameter, which refers to the number of layers of quantum gates or operations applied to the input qubits to transform the classical data into a quantum state. Each layer in the `PauliFeatureMap` typically consists of single-qubit rotations and entangling gates involving Pauli matrices. The depth of the `PauliFeatureMap` is determined by the number of such layers applied to the input qubits. The depth of a `PauliFeatureMap`, or any quantum circuit, represents the complexity or the number of sequential operations applied to the input qubits to encode the classical data into a quantum state. A deeper circuit may capture more complex patterns in the data but may also require more computational resources. [30]

Chapter 3

QSVM for Malware Classification

In this chapter, we explore the practical application of the Quantum Support Vector Machine (QSVM) algorithm, including the dataset we use and the preprocessing steps we take to prepare the data. The source codes with detailed documentation are available on our project's GitLab repository[1]. We drew inspiration from the concepts introduced in [12] for our implementation. Additionally, the initial ideas for the implementation of the QSVM module were influenced by the work of [30].

Our implementation of the QSVM algorithm consists of two main Python modules: the `peml` module, responsible for preprocessing the dataset, and the `svm` module, which implements SVM algorithm with both quantum and classical kernels. These modules are designed to function independently. While the `peml` module focuses on preprocessing of the specified dataset we use, the `svm` module can classify any preprocessed datasets in the specific input format we explain later in the chapter.

Additionally, we implemented a separate `data_objects` module, which provides essential data structures utilised throughout the project. These structures include configurations for project directories, kernel matrix parameters, and SVM classification specifications.

Figure 3.1 depicts the structure of our implemented source codes, highlighting the modular organization and the classes contained within each module.

[1] `https://gitlab.fit.cvut.cz/kratkeli/ni-dip`

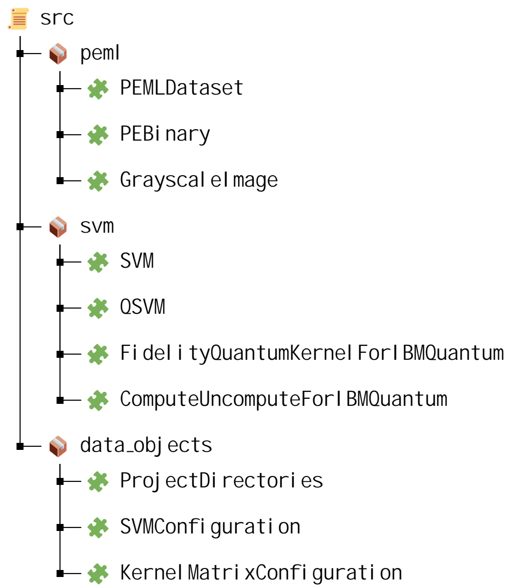

Figure 3.1 Simplified Project Layout With Implemented Modules and Classes

3.1 PEML Module

We developed the `peml` module specifically for the preprocessing of the PE Malware Machine Learning dataset [39] we intend to use in our experiments with QSVM classification on the simulator and IBM Quantum Systems. The PE Malware Machine Learning dataset consists of raw labelled PE file binaries containing benign and malware samples. It is distributed in an encrypted zip folder, with file extensions removed from the individual samples to prevent accidental execution.

The `peml` module enables randomly selecting, extracting, and preprocessing samples from the zipped dataset folder to obtain training and testing feature vectors for the QSVM classification. It consists of three classes: `PEMLDataset`, `PEBinary` and `GrayscaleImage`. The `PEMLDataset` class represents the dataset and utilises `PEBinary` and `GrayscaleImage` classes.

The `PEMLDataset` class facilitates the extraction and preprocessing of selected samples within the zipped dataset folder. It operates with a metadata file (*samples.csv*) that organises the dataset's structure. The class offers methods to randomly select training and testing samples from the dataset, ensuring an equal distribution of benign and malicious files. Additionally, it provides functionality to extract feature vectors from selected binary files by converting them into grayscale images and flattening them into 1D feature vectors. It also enables saving and loading dataset components, such as IDs, labels, and feature vectors, using HDF5 files [40]. The `extract_samples` class method executes 7zip commands [41] to extract selected samples from the dataset folder. This approach allows for the extraction of only the necessary samples from the dataset, thus bypassing the need to unzip the entire dataset at once. The dataset has a considerable size of 43.8GB compressed and uncompressed 117GB, so computational and memory resources are effectively managed as the samples are selectively extracted as needed.

The `PEBinary` class encapsulates functionality for handling Portable Executable (PE) binary files. It is designed to convert a binary file into a grayscale image, represented by the `GrayscaleImage` class. It utilises helper methods to load binary data, validate PE file structure, and calculate the appropriate image width. When initialised with a file path pointing to a PE binary, it loads the binary data and verifies its integrity as a PE file. It checks the presence of the 'MZ' signature at the beginning of the binary, and the 'PE' signature at the specified offset. The `get_image` method converts the binary data into a grayscale image, adjusting its width based on the size of the binary content. This process involves reshaping the binary data into a 2D array, with dimensions determined by predefined size ranges introduced in [19].

The `GrayscaleImage` class complements the functionality of the `PEBinary` class by providing utilities for working with grayscale representations of PE binaries. It provides methods for resizing the image while maintaining its aspect ratio, extracting a feature vector from it, and plotting it for visualisation. The class also offers a property to retrieve the size of the image, allowing access to its dimensions.

3.2 SVM Module

The `svm` module is our implementation of the QSVM algorithm. The module consists of four classes: `SVM`, `QSVM`, `FidelityQuantumKernelForIBMQuantum` and `ComputeUncomputeForIBMQuantum`.

The `SVM` class provides a flexible framework for SVM classification, allowing users to choose between quantum and classical methods and customise the classification process with various feature maps and kernels. It introduces the `SVMConfiguration` data object, a container for storing and accessing parameters required for SVM classification, including dataset sizes, feature dimensions, quantum circuit specifications, and runtime configuration for executing jobs on IBM Quantum systems.

During initialisation, the `SVM` class loads already preprocessed feature vectors of samples from the dataset represented by the `PEMLDataset` class. It subjects them to further preprocessing using the private helper method `__preprocess_data`, which preprocesses the training and testing samples by reducing their dimensions to match the number of qubits used in the quantum SVM, using Principal Component Analysis. It then normalises the data by eliminating the mean and scaling it to unit variance, employing `StandardScaler` and `MinMaxScaler` from the `scikit-learn` library [42], respectively. These steps ensure that the data is appropriately prepared for classification.

The `classify` method operates using both quantum and classical methods. It accepts lists of quantum feature maps and classical kernels as optional parameters. For each provided quantum feature map, it initialises a `QSVM` object and performs classification. Similarly, for each specified kernel, it performs classical classification using the `__classical_classification` method.

The `__classical_classification` method executes classification utilising a chosen classical kernel for the SVM. It begins by accepting the type of classical kernel as input and initialising an SVM classifier (`SVC` from `scikit-learn`) with the designated classical kernel. Following this, it measures the training time by recording the start and end times of the training process. The classifier is then trained using the provided training samples

and their corresponding labels. Once trained, it predicts labels for the test samples. It computes the classification metrics, accuracy and F1 score based on the predicted and actual labels. Finally, it prints the training time to the standard output and returns the classification metrics as a tuple. We explain these metrics in the following chapter when discussing experiments.

3.3 Implementation of QSVM

In the `QSVM` class, which is a part of the `svm` module, we implement the interface for the QSVM classification on both the local simulator and IBM Quantum systems. The `QSVM` object is created inside the `SVM.classify` method and requires two parameters inherited from the `SVM` object: `project_directories` and `config`. The first parameter is an object specifying the project directories, while the second is a configuration for the classification.

Quantum-based classification involves specific parameters distinct from classical SVM, which utilise classical kernels. Firstly, there is the `simulator` parameter, a boolean value indicating whether to perform classification using a quantum-computer simulator or IBM Quantum system. Secondly, the `ibm_backend` parameter specifies the backend's name in the case of classification using real quantum computers. Lastly, the `runtime_jobs_completed` parameter indicates whether the jobs on the IBM Quantum system are finished.

The `runtime_jobs_completed` parameter is needed for the classification on the real quantum hardware, which can be divided into two parts. The initial part involves submitting jobs to the IBM Quantum to calculate the entries of the kernel matrices. Since these jobs may require several hours to execute on a quantum computer, the classification script need not run continuously. It can only submit the jobs and save the configuration of kernel matrices. Therefore, it can save local computing resources. After the quantum computer finishes executing the jobs (observable on the IBM Quantum Platform), the second part of the classification process follows. It involves loading the kernel matrix configuration, processing the finished jobs, and evaluating the kernel matrices. The subsequent script or program can execute this task with the `runtime_jobs_completed` parameter set to true.

The kernel matrix configuration plays a crucial role in the classification process, providing information for evaluating the kernel matrices, which are fundamental to support vector classification.

The `classify` method is the core functionality of the `QSVM` class. It performs the support vector classification using a specified quantum kernel. This method accepts `feature_map_type` as an argument, denoting the type of quantum feature map to use. Supported types include `ZZFeatureMap` [37], `PauliFeatureMap` [36], `ZZphiFeaturemap`, and `ZFeatureMap` [38].

The `ZZphiFeaturemap` [12] is a modified version of the `ZZFeatureMap` with a custom data mapping function

$$\phi_S(x) = \begin{cases} x_i & \text{if } S = \{i\} \\ \sin(\pi - x_i)\sin(\pi - x_j) & \text{if } S = \{i, j\}\,, \end{cases} \tag{3.1}$$

where S is a set of qubit indices that describes the connections in the feature map [12, 14]. The custom data mapping function is implemented in the `_custom_data_map_func` method [30].

The number of qubits and depth of the feature map quantum circuits are specified in the `SVMConfiguration` data object provided during initialization.

The classify method utilizes the `scikit-learn` library [42] for the support vector classification, similar to the `SVM` class. It initializes the `sklearn.SVC` object [42, 43], representing the machine learning model and fits the model with the precomputed train matrix along with corresponding labels. It measures the training time and prints it to the standard output. Subsequently, it computes the classification metrics, accuracy and F1 score using a precomputed test matrix and corresponding labels. The method returns a tuple containing the classification metrics. However, if the runtime jobs necessary for classification are incomplete, it returns `(None, None)` to indicate that the results are not yet available.

The quantum kernel evaluation, whether using a simulator or IBM Quantum hardware, is internally managed by two private methods: `__evaluate_on_simulator` and `__evaluate_on_ibm_hardware`, executed within the `classify` method. While the evaluation process is generally similar in both cases, there are slight differences.

Both methods return the precomputed train and test kernel matrices. They rely on sampler, kernel and fidelity objects. The fidelity object leverages the sapler primitive to compute the state fidelity of two parametrized quantum circuits. The kernel object constructs the kernel matrix and uses the fidelity object to evaluate the entries in the kernel matrix. The sampler primitive varies based on the evaluation type: for simulator evaluation, it is sourced from the `qiskit.primitives` module [35], while for hardware evaluation, it is obtained from the `qiskit_ibm_runtime` module [44] and is instantiated within the fidelity object exclusively.

The kernel object is an instance of the `FidelityQuantumKernelForIBMQuantum` class. This class is a modified version of the `FidelityQuantumKernel` class [32] from the `Qiskit Machine Learning` module [31]. It extends the functionality of the `FidelityQuantumKernel` class by providing additional methods and handling fidelity computation specifically for use with IBM Quantum systems. The fidelity object is an instance of the `ComputeUncomputeForIBMQuantum` class, which is, similar to the kernel, an extended version of the `ComputeUncompute` class [45] within the `Qiskit Machine Learning` module [31]. It supports running computations on IBM Quantum systems, including handling hardware-specific considerations and job execution.

Modifications for Execution on IBM Quantum

The original implementation of the `FidelityQuantumKernel` [32] and `ComputeUncompute` [45] classes from the `Qiskit Machine Learning` [31] module shows three significant issues.

Firstly, these classes lack the ability to split the evaluation process into two distinct parts: submitting jobs to calculate kernel matrix entries on IBM Quantum and processing the finished jobs. Consequently, the classification script must run continuously while awaiting job execution on the IBM Quantum system, which can take several days

depending on the job queue length. This inefficiency not only consumes resources but also restricts the scalability of the evaluation process, particularly with large datasets. We aim to address this issue to optimize resource usage and enhance scalability.

The absence of transpilation for fidelity circuits before submission to the IBM Quantum presents a critical flaw in the original implementation. Transpilation refers to the process of transforming quantum circuits to use only instructions supported by the underlying quantum hardware. This transformation ensures compatibility and efficient execution on real IBM Quantum hardware. As of 1 March 2024, IBM Quantum introduced a significant change to improve the speed and efficiency of quantum computation [46, 47]. Circuits and observables are now required to undergo transformation to only use the instruction set architecture (ISA) supported by the quantum system. It means that circuits must be transpiled before being submitted to the Qiskit Runtime primitives for execution. Without the transpilation, the fidelity circuits cannot be executed on the IBM Quantum hardware, which makes the classes unusable in real-world scenarios. An illustration of the transpilation error we encountered on the IBM Quantum Platform is shown in Figure 3.2. It is worth noting that the transpilation issue is known and tracked by the Qiskit community, affecting several classes beyond those discussed here, yet at the time of finishing this thesis, it has not been resolved [48, 49].

Status: ⊗ Failed - Circuits do not match the target definition (non-ISA circuits). -- \n Transpile your circuits for the target before submitting a primitive query. For\n example, you can use the following code block given an IBMBackend object `backend`\n and circuits of type `List[QuantumCircuit]`:\n from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager\n pm = generate_preset_pass_manager(optimization_level=1, target=backend.target)\n isa_circuits = pm.run(circuits)\n Then pass `isa_circuits` to the Sampler or Estimator.\n -- https://ibm.biz/error_codes#1517

Figure 3.2 Error Message on IBM Quantum Platform Due to Lack of Circuit Transpilation

Moreover, the original classes submit all fidelity circuits in a single job to the IBM Quantum interface. While this approach is suitable for local simulation, it proves impractical for larger datasets on IBM Quantum systems. The resulting job size often exceeds the maximum limit [50], preventing the circuit submission for execution and severely limiting class usability, particularly with larger datasets. An illustration of the error message we encountered due to this limitation is provided in Listing 3.1.

```
IBMRuntimeError: Failed to run program: \ 413 Client Error: Payload Too Large
↪ for url: https://api.quantum.ibm.com/runtime/jobs.
↪ {"statusCode":413,"message":"request entity too large"}\
```

Code listing 3.1 Error Message Due to Job Size Exceeding Maximum Limit on IBM Quantum Platform.

In the extended versions of the classes (`ComputeUncomputeForIBMQuantum` and `FidelityQuantumKernelForIBMQuantum`), we address these issues by introducing enhancements to the initialisation and methods. These improvements enable efficient resource utilisation, ensure compatibility with IBM Quantum hardware, and enhance scalability for real-world machine learning applications. In the following sections we

compare the original and our extended classes, highlighting differences and explaining how we address the identified issues.

ComputeUncomputeForIBMQuantum

In the ComputeUncomputeForIBMQuantum class, we implement the BaseStateFidelity algorithm [34] designed for compatibility with IBM Quantum Platform. It leverages the sampler primitive to compute the state fidelity between two parametrised quantum circuits (feature maps), following the compute-uncompute method [45, 14].

We introduce additional parameters such as simulator, backend, and shots upon initialisation for compatibility with the IBM Quantum Platform. The simulator is a boolean value specifying whether to run the computation on a simulator or IBM Quantum system. The backend parameter specifies the quantum hardware's name, while shots specify the number of measurement shots performed during circuit execution. The sampler parameter is optional in contrast to the original ComputeUncompute class [45]. That is because, in the case of the hardware evaluation, the sampler is instantiated within the object exclusively. It is retrieved from the qiskit_ibm_runtime module [51, 44] and depends on the backend and session configurations. The session feature [52] of Qiskit Runtime streamlines the execution of multi-job iterative workflows on quantum computers. Using sessions helps avoid delays caused by queuing each job separately, which can be particularly useful for iterative tasks involving frequent communication between classical and quantum resources [52].

We introduces five additional methods in the ComputeUncomputeForIBMQuantum class: __run_on_simulator, __run_on_ibm_hardware, run_jobs, get_fidelities, and _construct_circuits (a modified version of the method from the BaseStateFidelity class [34]) and alongside it edits to the _run and _call methods. In the ComputeUncomputeForIBMQuantum class, the _run and _call methods, while present, serve only as placeholders, so the class adhere to the BaseStateFidelity interface requirements. The run_jobs and get_fidelities methods replace the logic provided by _run and _call methods.

The rub_jobs method is responsible for running the jobs for computing the state overlap (fidelity) between two parametrised circuits. It is mostly similar to the original _run method in ComputeUncompute [45]. However, it includes additional logic to differentiate between running on a simulator or IBM Quantum hardware. To do so, it uses two helper methods __run_on_simulator and __run_on_ibm_hardware. The method also incorporates logic for transpilation, ensuring fidelity circuits are optimised for IBM Quantum hardware execution via the _construct_circuits method.

The _construct_circuits method is a modified version of the _construct_circuits method from the Basestatefidelity class, which is responsible for constructing the list of fidelity circuits to be evaluated. Additionally, it considers whether to run the circuits on a simulator or IBM Quantum system. In the case of the simulator, the process remains unchanged from the original method. In the case of the evaluation on hardware, it transpiles the fidelity circuits to be run on the desired backend using the PassManager [53, 54].

Helper methods __run_on_simulator and __run_on_ibm_hardware facilitate execution on the local simulator and IBM Quantum hardware, respectively. The __run_on_simulator mimics the last step from the original _run method from

ComputeUncompute [45]. It executes all circuits in a single job using the Sampler from qiskit.primitives [35], returning a list containing the submitted job. The job is returned in a list due to the simplification of processing the following jobs in the get_fidelites method. The __run_on_ibm_hardware method submits the jobs to the IBM Quantum hardware. As we explained earlier, submitting the fidelity circuits in one job would quickly exceed the maximum limit for job size. Therefore, each fidelity circuit is submitted in a separate job using the sampler primitive obtained from the qiskit_ibm_runtime module [44]. The jobs are submitted in session to avoid delays caused by separately queuing each job.

The get_fidelities method processes job results and computes fidelities, replacing the _call method from the original ComputeUncompute class [45]. It includes additional logic to handle results differently based on whether the evaluation was performed on the simulator (in a single job) or the IBM Quantum system (in multiple jobs). The method also allows for splitting the evaluation process. It can be run after the jobs submitted to IBM Quantum are finished.

In summary, ComputeUncomputeForIBMQuantum extends ComputeUncompute [45] by addressing the three significant issues mentioned previously. It enables the splitting of the evaluation process, replacing the original _run and _call methods into two independent methods, run_jobs and get_fidelities. It provides the transpilation of the fidelity circuits before submitting them to the IBM Quantum system via the _construct_circuits method. Moreover, it divides the submitted job into smaller jobs, fixing the issue by exceeding the max job size limit given by the IBM Quantum Platform. This enhanced functionality enhances the usability and scalability of the algorithm for real-world applications.

FidelityQuantumKernelForIBMQuantum

In the FidelityQuantumKernelForIBMQuantum class we implement the quantum kernel interface based on the BaseStateFidelity algorithm [34] designed for compatibility with IBM Quantum. We use the fidelity computations provided by ComputeUncomputeForIBMQuantum class to construct kernel matrices for the quantum kernel in QSVM algorithm.

We introduce the KernelMatrixConfiguration data object in the FidelityQuantumKernelForIBMQuantum class to encapsulate the configuration for computing the kernel matrix entries. The KernelMatrixConfiguration object includes information such as whether the matrix is symmetric, its shape, the number of circuits, indices, and associated jobs for evaluating the kernel entries on the quantum computer. It abstracts away the details about submitted jobs, providing a clean interface for kernel matrix evaluation in the helper private methods __get_kernel_matrix or __get_kernel_entries.

We divide the original evaluate method in FidelityQuantumKernel [32] into two distinct phases: obtaining the kernel matrix configuration and computing the matrix using this configuration. Initially, the FidelityQuantumKernelForIBMQuantum class submits jobs to retrieve kernel entries through the get_kernel_matrix_config method. Once these jobs are completed, the evaluate_matrix method processes the results to compute the kernel matrix. Users have the flexibility to execute the evaluation process either as a single operation using the evaluate method or as separate steps using the

get_kernel_matrix_config and evaluate_matrix methods.

The get_kernel_matrix_config is similar to the original evaluate method [32]. It validates the input, determines the kernel shape and computes all the combinations needed to evaluate the kernel entries via the helper parameterization methods (__get_parameterization, __get_symmetric_parameterization). However, instead of directly constructing the kernel matrix, it only submits the jobs for evaluating the kernel entries via the helper __run_jobs method and returns the matrix configuration.

The __run_jobs method is a helper method inspired by the original get_kernel_entries method. It is responsible for submitting jobs to calculate the kernel matrix entries via the underlying fidelity instance (ComputeUncomputeForIBMQuantum).

The evaluate_matrix method constructs the kernel matrix using the provided configuration and the fidelity computation results. It utilizes the helper methods __get_kernel_matrix and __get_symmetric_kernel_matrix and the modified __get_kernel_entries method, which executes the get_fidelities method from the underlying fidelity instance (ComputeUncomputeForIBMQuantum).

Our modifications enable integration of the fidelity-based quantum kernel with the IBM Quantum Platform and allow for efficient computation of kernel matrices on real quantum hardware.

Chapter 4
Experiments

In this chapter, we perform experiments to test and evaluate our implementation of the QSVM algorithm we described in the previous chapter. The groundwork for our experiments is laid by the work of [12], which provides insights into the performance of quantum machine learning algorithms, particularly in the context of malware classification. Reproducing their results serves as a key benchmark for our experiments.

In the initial phase of our experiments, we perform QSVM classification using a local simulator on datasets of varying sizes, ranging from 500 train samples and 100 test samples to 8000 train samples and 4000 test samples. Subsequently, we perform SVM classification using classical kernels for comparison analysis. We aim to directly compare our results with those presented in the [12] by replicating their experimental setup as closely as possible. Their findings highlight the efficiency of QSVM algorithms, particularly in extracting information from smaller datasets, suggesting their potential in cybersecurity applications. By comparing the results obtained from the simulator with those of SVM classification using classical kernels and referencing the findings of [12], we aim to evaluate the relative performance of QSVM algorithms across different dataset sizes.

The second part of the experiments refers to experiments on IBM Quantum systems. Inspired by the promise of NISQ quantum computers, our initial objective was to implement and evaluate quantum machine learning algorithms, particularly the QSVM algorithm, on IBM Quantum computers. However, during the implementation process we encountered challenges that significantly impacted the direction of our experiments. These challenges primarily rise from the limitations inherent in the Qiskit Machine Learning module, specifically relating to transpilation requirements and the constraints imposed by job sizes on IBM Quantum systems.

To address these challenges, we devoted considerable effort to fixing the limitations within the Qiskit Machine Learning module, as we further described in Chapter 3. It was essential to fix those issues, as without resolving them, we would not have been able to execute any code on the IBM Quantum computers. However, despite our efforts, we are still constrained by suboptimal execution times on the IBM Quantum systems. This limitation underscores the need for further research and optimisation, a pursuit we intend to undertake in the future. Consequently, we find ourselves severely restricted in

the size of datasets we can operate with on the IBM Quantum systems. As a result, we must adjust the original purpose of our experiments.

Our focus now shifts towards exploring how the limitations imposed by the Qiskit Machine Learning module affect our experimental setup. Instead of solely evaluating the accuracy and F1 score of our classification algorithm, we aim to investigate how the IBM Quantum systems behave under the workload of numerous jobs.

This chapter describes the detailed setup, execution, and evaluation of our quantum machine learning experiments. First, we present our experimental setup, which utilises the hardware, primarily IBM Quantum computers, and the software components, including our custom-implemented Python modules. We describe the used dataset and provide a clear breakdown of the configuration of our experiments, detailing parameters, dataset sizes, and computational resources at our disposal.

We then discuss the evaluation metrics employed to assess the performance of our classification algorithm. We describe the reasoning behind our choice of metrics, such as accuracy, F1 score, and “quantum time”, clarifying how each metric contributes to our understanding of the algorithm’s effectiveness despite the constraints of small dataset sizes. Finally, we run the experiments and present our findings, acknowledging the challenges posed by the limitations of the Qiskit Machine Learning module and the constraints of working with small datasets on IBM Quantum systems.

4.1 Experimental Setup

In both classical and quantum computing experiments, the experimental setup includes the configuration of hardware, software, and environmental conditions essential for conducting the experiment and gathering data. The comprehensive description of the experimental environment ensures the reproducibility of all tests and measurements and provides a roadmap for others to replicate the experiments effectively.

We rely on the PE Malware Machine Learning Dataset [39] for our experiments. It consists of raw binaries of PE files, such as .exe or .dll files and contains 201,549 labelled samples, with 86,812 benign and 114,737 malware samples. It is distributed in an encrypted zip folder, with file extensions removed from the individual samples to prevent accidental execution. Most malicious samples come primarily from the platforms VirusShare[1], MalShare[2] and TheZoo[3], which are free malware repositories intended for security researchers. The legitimate files come from instances of various versions of Windows 7 and above with a variety of different software downloaded and installed. However, there is a bias towards files associated with Microsoft products among the benign samples. The main benefit of the dataset is that it provides the raw binary files themselves instead of just metadata that has already been extracted from the samples.

As our experiments involve analysing and classifying malware samples from the dataset, ensuring a secure and controlled environment for handling these potentially harmful files is important. All the experiments, including data extraction, preprocessing and classification, are undertaken in a sandboxed environment on a virtual machine provided by FIT CTU through the CloudFIT [55] platform.

[1] https://virusshare.com/
[2] https://malshare.com/
[3] https://github.com/ytisf/theZoo

Hardware

In the realm of hardware, our experiments utilize various resources, including a local simulator, IBM Quantum processors, and the resources provided through the FIT CTU computational server. A crucial aspect of the experimental design involves comparing the benefits and disadvantages of using the simulator and hardware. While the simulator offers flexibility and ease of use, it may not fully capture the complexities of quantum behaviour. In contrast, IBM Quantum computers provide a glimpse into real-world quantum processing, yet their capabilities are bounded by factors such as limited computational time and queue constraints. The selection of specific IBM Quantum computers for this experiment is driven by their respective capabilities and limitations, with careful consideration for factors such as queue lengths and computational resources. Additionally, including the CTU subscription further enhances the experiment's access to quantum computing resources.

When comparing the simulator and hardware, weighing the benefits and disadvantages of each is important. The simulator allows for prototyping and debugging. However, it can only simulate quantum computing to some extent and at a higher computational cost.

On the other hand, IBM Quantum computers offer the advantage of executing computations on real quantum hardware and enabling algorithm validation in real-world conditions. However, using IBM Quantum computers introduces challenges such as limited access to computational time, variability in queue lengths, and the potential for errors arising from hardware imperfections.

In practice, selecting specific IBM Quantum computers depends on their capabilities and constraints. For example, when choosing backends, we prioritize those with shorter queue lengths to minimize waiting times. We have access to the IBM Quantum computers thanks to the CTU license, which has only 400 minutes of 'quantum time' per month available for the computations. Quantum time refers to the duration, in seconds, a quantum system is committed to fulfilling a user request [56]. We explore this constraint later when discussing the experiments.

Software

On the software side, Python scripts play a crucial role in executing the experiments, along with utilising a sandboxed environment on the FIT CTU computational server. The sandboxed environment ensures a controlled and secure setting for experimentation, while the dataset used for classification undergoes preprocessing through dedicated scripts. These scripts facilitate the preprocessing of the dataset and execute the classification process, enabling the systematic evaluation of malware classification using quantum SVM techniques.

The data extraction and preprocessing are handled by two Python scripts, `get_ids_labely.py` and `get_fv.py`, demonstrating the practical usage of the `peml` module, described in Chapter 3. The first script randomly selects training and testing samples from the dataset based on the command line parameters `train_size` and `test_size`. It saves their IDs and labels to an HDF5 file and verifies the correctness of the saved data. The second script, `get_fv.py`, extracts feature vectors from the

selected samples. Besides the size of training and testing samples, it requires two additional command line parameters, image_width and image_height, which specify the sizes of the grayscale images representing the binaries. After loading IDs and labels from a saved HDF5 file, the script extracts the samples from the zipped dataset folder, initialises the image size, and retrieves feature vectors. It saves the feature vectors to another HDF5 file and verifies their correctness.

The SVM classification, coordinated by the svm module described in Chapter 3, is handled by three Python scripts, which differentiate based on computational environment (simulator or quantum hardware) and method (classical SVM or QSVM). The scripts set up the SVMConfiguration based on the input parameters, execute the classification process, and save the results to output files. All three scripts require parameters from the command line, such as train_size, test_size, n_features, n_qubits, and depth. train_size, test_size and n_features specify the feature vectors required for the classification. n_qubits, and depth specify the number of qubits for the quantum classification and the depth of the data encoding feature map.

The classify_on_hardware.py script facilitates QSVM classification using IBM Quantum hardware resources. It accepts additional command-line arguments specifying the number of shots, the type of feature map, IBM Quantum backend, and runtime job completion. In contrast, the classify_on_simulator.py script performs SVM classification using the quantum computer simulator. It accepts command-line arguments for feature maps and shots but does not require IBM backend or job completion specifications. Unlike the previous scripts, classify_with_classical_kernel.py focuses on SVM classification using classical kernels, namely linear, polynomial, SBF and sigmoid.

Listing 4.2 shows the usage of the scripts to perform classification on the debugging dataset with four training samples and two testing samples. Listing 4.1 lists the versions of Python packages we use in our implementation.

```
qiskit: 1.0.1
qiskit-algorithms: 0.3.0
qiskit-ibm-runtime: 0.20.0
qiskit-machine-learning: 0.7.2
scikit-learn: 1.4.2
numpy: 1.26.4
```

Code listing 4.1 Versions of Used Python Packages

```
# Step 1: Randomly select four training samples and two testing samples from the
↪  dataset and save their IDs and labels
python3 working_dir/scripts/server/get_ids_labels.py 4 2

# Step 2: Load the IDs and labels of the training and testing samples, then extract
↪  the samples from the zipped dataset folder
# Transform the binary samples into 64x64 grayscale images and save their feature
↪  vectors
python3 working_dir/scripts/server/get_fv.py 4 2 64 64

# Step 3: Load the feature vectors and submit jobs for evaluating the quantum kernel
↪  matrix in QSVM classification on IBM Quantum processor ibm_cairo
# Parameters: train_size test_size n_features n_qubits depth shots feature_map
↪  runtime_jobs_completed ibm_backend
python3 working_dir/scripts/server/classify_on_hardware.py 20 10 4096 4 2 1000 zzPhi
↪  1 ibm_cairo

# Step 4: Load the feature vectors, process finished jobs, and complete QSVM
↪  classification
python3 working_dir/scripts/server/classify_on_hardware.py 20 10 4096 4 2 1000 zzPhi
↪  1 ibm_cairo

# Step 5: Load the feature vectors and perform QSVM classification on the simulator
# Parameters: train_size test_size n_features n_qubits depth shots
python3 working_dir/scripts/server/classify_on_simulator.py 4 2 4096 4 2 1000

# Step 6: Load the feature vectors and perform SVM classification using classical
↪  kernels
# Parameters: train_size test_size n_features n_qubits
python3 working_dir/scripts/server/classify_with_classical_kernel.py 4 2 4096 4
```

Code listing 4.2 Example Usage of the Classification Scripts

4.2 Evaluation Metrics

Our primary metrics for evaluating the performance of the QSVM algorithm are accuracy and F1 score.

Accuracy represents the proportion of correctly classified samples out of the total number of samples. It provides a straightforward indication of the model's correctness in classification tasks.

In addition to accuracy, we use the F1 score, similar to the authors in [12]. Their work is a benchmark for our simulator experiments, so we use the same metrics.

The F1 score can be interpreted as a harmonic mean of precision and recall. It reaches its best value at 1 and worst at 0. It is defined as

$$F1 = \frac{2 \times TP}{2 \times TP + FP + FN} \tag{4.1}$$

where TP is the number of true positives, FN is the number of false negatives, and FP is the number of false positives [57]. If there are no TP, FN or FP samples, the default value of the F1 score is 0.

4.3 Experiment Parameters

The main parameters of our experiments are the types of quantum kernel feature maps and the classical kernels we use for comparison.

We consider the number of qubits on the quantum computer, a fundamental parameter influencing the preprocessing steps and the quantum feature map. Specifically, the number of qubits dictates the length of input feature vectors and the dimensionality of the quantum feature space. To maintain consistency between classical and quantum kernels, we align the PCA in the preprocessing part with the chosen number of qubits, ensuring compatibility and fair comparison between classical and quantum representations of the data.

Additionally, we control the depth of the feature map. The depth of the quantum circuit, in our case, represented by the feature map, is a measure of how many 'layers' of quantum gates executed in parallel it takes to complete the computation defined by the circuit [58]. It corresponds to the time it takes the quantum computer to execute the circuit, which is an important aspect we must consider in our experiments.

We set the number of shots on the quantum computer, which is the number of repetitions of each circuit for sampling. Increasing the number of shots influences the statistical significance of the quantum measurements but at the cost of the computational time.

We specify the size of the grayscale image when we convert binary samples to the feature vectors using the grayscale conversion.

Lastly, we specify the IBM backend when classifying the IBM Quantum computer.

4.4 Findings

In this section, we present our findings from the experiments. We first run QSVM classification on the local simulator, utilising datasets of varying sizes ranging from 500 train samples and 100 test samples to 8000 train samples and 4000 test samples. We then compare our results with those presented in [12] to assess the performance of our implementation. Subsequently, we transition to QSVM classification on IBM Quantum computers, employing tiny datasets comprising up to 20 train and 10 test samples. The primary objective of these experiments is to investigate how IBM Quantum systems behave under the flood of numerous jobs required to evaluate the kernel matrix. Through this exploration, we aim to gain insights into the capabilities and limitations of IBM Quantum systems in practical applications, including scalability and resource utilisation considerations.

Simulator

As we mentioned in the previous chapter, [12] provides a comprehensive exploration of quantum machine learning algorithms, mainly focusing on Quantum Support Vector Machines (QSVM) and Quantum Neural Networks (QNN). Moreover, the study investigates two preprocessing techniques, namely the 'Ordered Importance Features method' and the 'Grayscale method'. Notably, while [12] explores both QSVM and QNN to-

gether with both preprocessing techniques, our research narrows its focus to the QSVM algorithm coupled specifically with the 'Grayscale method', aligning with the objectives of the thesis.

The study by [12] makes observations regarding the efficiency of QSVM algorithms, particularly concerning their performance relative to classical counterparts. Notably, their findings underscore the potential of QSVM algorithms to outperform classical approaches, mainly when operating with smaller datasets. With the ZZFeatureMap, quantum circuits demonstrated a notable accuracy improvement of up to 2.5% in specific configurations, which suggests the capability of QSVM to extract more information from limited data, a critical aspect in cybersecurity applications.

However, we encountered several challenges when replicating their results due to the paper's lack of detailed experimental descriptions and parameter specifications. They do not specify library versions, which is crucial given the rapid evolution of the Qiskit SDK environment. More importantly, they do not specify how many qubits and shots and which processor they used when conducting experiments on IBM Quantum devices. Additionally, they are not consistent with their metrics, such as not constantly measuring the F1-score, and if so, it is not clear to which parameters it belongs. More clarity is needed to ensure accurate replication and comparison of results.

Table 4.1 and Table 4.3 present the results we compiled from the paper [12], focusing on experiments conducted with four qubits. We omitted the F1 score metric, as the authors do not always report it for their experiments. However, the complete results presented in [12], including F1-scores for different qubit configurations, can be found in Appendix A. Our experimental results, which mimic those presented in Table 4.1 and Table 4.3, are shown in Table 4.2 and Table 4.4.

Table 4.1 displays the accuracies obtained using QSVM with different types of quantum kernels, distinguished by the utilised feature map. The feature maps, namely ZZFeatureMap (ZZ), PauliFeatureMap (Pauli), ZZphiFeatureMap (ZZphi), and ZFeatureMap (Z), were introduced and discussed in detail in earlier sections of the thesis. The parameters for the QSVM experiments include four qubits and a depth of two for each feature map, with varying sizes for the training and testing datasets, as indicated in the table. It is worth noting that further elaboration on these feature maps and their significance can be found in the preceding chapter, providing readers with a comprehensive understanding of the experimental setup.

Table 4.2 mirrors the structure of Table 4.1 but showcases the results obtained from our experiments on a simulator. These experiments aimed to replicate the conditions outlined in the referenced paper, utilizing the same parameter settings, including four qubits and a depth of two for the feature maps. We conducted these experiments with 1000 shots, as the paper did not specify the shot count. We used grayscale images of size 64×64, consistent with the specifications provided in the paper. The input data were obtained by preprocessing binary samples, converting them into grayscale images, resizing them to 64×64 dimensions, and converting them into 1D feature vectors.

Table 4.3 presents the accuracies achieved by SVM with classical kernels. The parameters for the classical kernel experiments include four qubits, which are utilised for preprocessing the data through PCA. PCA transforms the binary data into 1D feature vectors, aligning with the approach employed in the QSVM experiments. Similar to

Table 4.1, the dataset sizes vary, with different combinations of training and testing samples. A detailed explanation of the preprocessing steps and their implications can be found in Chapter 3, ensuring clarity regarding the experimental methodology.

Table 4.4 parallels Table 4.3 but presents the results obtained from our experiments with classical SVM kernels. Like Table 4.3, these experiments aimed to replicate the conditions outlined in the referenced paper, ensuring a fair comparison between classical and quantum approaches.

The results in 4.1 and 4.3 showcase our findings from experiments conducted on a simulator, replicating the conditions outlined in the referenced paper. Despite our efforts to maintain consistency with the original experiments, we observed slight variations in accuracy compared to the results reported in the referenced paper.

Data (Train/Test)	ZZ	Pauli	ZZphi	Z
500/100	0.78	0.75	0.82	0.81
1000/200	0.815	0.79	0.815	0.805
2000/400	0.8125	0.79	0.8075	0.8
4000/800	0.7925	0.786	0.811	0.8075
8000/1600	0.805	0.7975	0.814	0.808

Table 4.1 Accuracy Comparison Between QSVMs With Different Feature Maps With 4 Qubits and Depth 2 [12].

Data (Train/Test)	ZZ	Pauli	ZZphi	Z
500/100	0.73	0.78	0.8	0.79
1000/200	0.73	0.66	0.735	0.735
2000/400	0.748	0.743	0.775	0.767
4000/800	0.806	0.821	0.771	0.775
8000/1600	0.812	0.792	0.804	0.806

Table 4.2 Experiment Results: Accuracy Comparison Between QSVMs With Different Feature Maps With 4 Qubits and Depth 2.

Data (Train/Test)	Linear	Poly	RBF	Sigmoid
500/100	0.77	0.78	0.8	0.61
1000/200	0.8	0.81	0.79	0.55
2000/400	0.77	0.78	0.8	0.54
4000/800	0.78	0.8	0.82	0.55
8000/1600	0.77	0.79	0.82	0.55

Table 4.3 Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the 4 Qubits Used in QSVM [12].

Data (Train/Test)	Linear	Poly	RBF	Sigmoid
500/100	0.74	0.72	0.79	0.54
1000/200	0.705	0.72	0.74	0.58
2000/400	0.718	0.685	0.765	0.585
4000/800	0.771	0.637	0.791	0.637
8000/1600	0.781	0.662	0.822	0.63

Table 4.4 Experiment Results: Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the 4 Qubits Used in QSVM.

The following tables provide an overview of the results obtained on a simulator with various qubit configurations, including the F1 scores.

Data (Train/Test)	Qubits	ZZ	Pauli	ZZphi	Z
500/100	3	0.74	0.79	0.8	0.78
	4	0.73	0.78	0.8	0.79
	6	0.66	0.72	0.81	0.81
	7	0.7	0.8	0.82	0.83
1000/200	3	0.725	0.675	0.735	0.72
	4	0.73	0.66	0.735	0.735
	6	0.745	0.76	0.78	0.775
	7	0.79	0.735	0.78	0.78
2000/400	3	0.71	0.73	0.748	0.757
	4	0.748	0.743	0.775	0.767
	6	0.777	0.728	0.77	0.78
	7	0.782	0.767	0.802	0.78
4000/800	3	0.799	0.784	0.771	0.777
	4	0.806	0.821	0.771	0.775
	6	0.83	0.812	0.816	0.8
	7	0.838	0.805	0.824	0.821
8000/1600	3	0.783	0.779	0.797	0.796
	4	0.812	0.792	0.804	0.806
	6	0.835	0.806	0.819	0.818
	7	0.851	0.812	0.831	0.821

Table 4.5 Accuracy Comparison Between QSVMs With Different Feature Maps With Depth 2.

Data (Train/Test)	Qubits	ZZ	Pauli	ZZphi	Z
500/100	3	0.736	0.79	0.797	0.777
	4	0.729	0.779	0.797	0.788
	6	0.649	0.716	0.808	0.81
	7	0.69	0.795	0.819	0.829
1000/200	3	0.723	0.675	0.732	0.717
	4	0.729	0.658	0.732	0.731
	6	0.744	0.756	0.779	0.775
	7	0.787	0.731	0.779	0.78
2000/400	3	0.707	0.728	0.747	0.756
	4	0.746	0.741	0.774	0.767
	6	0.776	0.726	0.769	0.78
	7	0.781	0.764	0.802	0.78
4000/800	3	0.797	0.783	0.769	0.775
	4	0.805	0.821	0.77	0.773
	6	0.83	0.811	0.816	0.799
	7	0.837	0.803	0.823	0.821
8000/1600	3	0.783	0.779	0.796	0.794
	4	0.812	0.792	0.803	0.805
	6	0.835	0.806	0.818	0.818
	7	0.851	0.812	0.83	0.821

Table 4.6 F1-score Comparison Between QSVMs With Different Feature Maps With Depth 2.

Data (Train/Test)	Qubits	linear	poly	rbf	sigmoid
500/100	3	0.75	0.69	0.76	0.51
	4	0.74	0.72	0.79	0.54
	6	0.74	0.74	0.81	0.56
	7	0.74	0.75	0.85	0.62
1000/200	3	0.705	0.73	0.745	0.575
	4	0.705	0.72	0.74	0.58
	6	0.735	0.745	0.79	0.64
	7	0.73	0.775	0.78	0.64
2000/400	3	0.718	0.672	0.77	0.603
	4	0.718	0.685	0.765	0.585
	6	0.74	0.735	0.782	0.595
	7	0.743	0.743	0.795	0.583
4000/800	3	0.766	0.639	0.787	0.671
	4	0.771	0.637	0.791	0.637
	6	0.772	0.804	0.83	0.608
	7	0.771	0.791	0.84	0.616
8000/1600	3	0.779	0.619	0.804	0.633
	4	0.781	0.662	0.822	0.63
	6	0.779	0.734	0.84	0.616
	7	0.776	0.746	0.845	0.608

Table 4.7 Accuracy Comparison Between SVMs With Different Classical Kernels and Preprocessing Corresponding to the Number Qubits Used in QSVM.

Data (Train/Test)	Qubits	linear	poly	rbf	sigmoid
500/100	3	0.746	0.662	0.754	0.51
	4	0.736	0.7	0.787	0.54
	6	0.736	0.729	0.808	0.56
	7	0.736	0.738	0.849	0.62
1000/200	3	0.7	0.728	0.742	0.574
	4	0.7	0.719	0.737	0.579
	6	0.73	0.738	0.789	0.64
	7	0.725	0.771	0.779	0.64
2000/400	3	0.716	0.651	0.769	0.601
	4	0.716	0.67	0.764	0.584
	6	0.739	0.729	0.782	0.595
	7	0.742	0.737	0.795	0.582
4000/800	3	0.764	0.612	0.786	0.671
	4	0.769	0.621	0.79	0.637
	6	0.771	0.803	0.83	0.607
	7	0.769	0.79	0.84	0.616
8000/1600	3	0.778	0.589	0.804	0.633
	4	0.779	0.646	0.822	0.63
	6	0.778	0.731	0.84	0.616
	7	0.775	0.743	0.845	0.607

Table 4.8 F1-score Comparison Between SVMs With Different Classical Kernels and Pre-processing Corresponding to the Number Qubits Used in QSVM.

IBM Quantum Systems

The second phase of our experiments consists of QSVM classification on IBM Quantum systems. Unlike the simulator experiments, where we benchmarked our results against those presented in the [12] paper, we opted for a different approach for the hardware experiments. There are several key reasons behind this decision.

Firstly, the [12] paper lacks crucial details regarding the experimental setup for their IBM Quantum experiments. Parameters such as the number of qubits, shots, and specific processor configurations used in their experiments remain unspecified. Without these essential parameters, it becomes impractical, if not impossible, to replicate their results accurately.

Secondly, the paper fails to provide information about the software environment and library versions used in their experiments. Given the rapid evolution of quantum computing frameworks like Qiskit, precise details about library versions are indispensable for reproducibility and validation purposes.

Moreover, the [12] paper was released in June 2023, predating significant updates to the IBM Quantum platform introduced in March 2024 [46, 47]. One such update mandates the transpilation of quantum circuits to conform to the instruction set architecture (ISA) supported by the target quantum system. However, since the paper

predates these updates, it is unclear whether the authors encountered similar transpilation requirements or faced the challenges posed by the updated IBM Quantum platform requirements.

As detailed in Chapter 3, transpilation is a critical requirement when executing quantum circuits on IBM Quantum systems. Failure to transpile circuits renders them incompatible with the underlying hardware and, therefore, unusable for real-world applications. While the issue is well-documented and acknowledged within the Qiskit community, it remains unresolved at the time of this study [48, 49]. Consequently, any comparison with the results presented in [12] would be inherently flawed due to the different experimental conditions.

Due to these challenges, we faced limitations when running experiments on real quantum computers. To address the issues outlined earlier, we implemented a fix involving the addition of transpilation and adopting a one-job-per-kernel-entry approach, as detailed in Chapter 3. Transpilation, a critical requirement when executing quantum circuits on IBM Quantum systems, involves adapting circuits to conform to the target quantum system's instruction set architecture (ISA). Our fix resolved the critical challenges, although more optimal and efficient solutions exist, as discussed later in this chapter. Time constraints during the master's thesis project limited our ability to explore these alternatives fully. Consequently, we could only conduct tests on tiny datasets, comprising a maximum of 20 train and 10 test samples.

QSVM classification necessitates two quantum kernel matrices: one for training and one for testing. The training matrix is symmetric with size $n \times n$, where n refers to the number of training samples. The test matrix is $m \times n$, where m refers to the number of testing samples. For instance, considering the 20 train and 10 test samples dataset, our one-job-per-kernel-entry approach translates to 390 jobs on the quantum computer.

During the debugging phase, we experimentally evaluated the "quantum time" it takes to run on a single job. One job consists of a parametrized quantum circuit (chosen feature map) with a concrete sample (feature vector) as a parameter. We experimented with the number of shots and different backends (quantum computers) and found out that, in our case, it takes approximately 15 'quantum seconds' to execute one job. Therefore, the total time required to evaluate the small dataset containing 20 train and 10 test samples is approximately 97.5 minutes on the quantum computer. These limitations are further compounded by the constraints of our CTU license, which grants us access to only 400 'quantum minutes' per month.

Transitioning from the evaluation of quantum time to the impact of job queue dynamics, we observed significant variations in execution time due to the dynamics of the job queue. Backend workload levels vary, leading to fluctuating queue lengths. Therefore, we initially opted for a combination of a least busy backend and a busier backend (in our case, `ibm_torino`) to examine the impact on queue wait times.

Additionally, we experimented with the number of jobs sent in a single session. Sessions allow all jobs to be executed consecutively, minimizing queue wait times. However, as the number of jobs in a session and the used quantum minutes approach the limit imposed by the license, the queue wait time increases exponentially. Consequently, even small datasets (e.g., 20 train and 10 test samples) could queue for up to approximately 14 days on the `ibm_torino` backend, leading us to explore alternative backends.

Table 4.9 presents the results of our experiments conducted on IBM Quantum systems using various backend configurations. The table showcases the performance metrics obtained from running Quantum Support Vector Machine (QSVM) classification tasks on datasets of different sizes. We began with smaller datasets of 4 train and 2 test samples, gradually increasing the dataset size to 8 train and 4 test samples, and finally evaluating performance on a larger dataset with 20 train and 10 test samples.

We executed experiments on different IBM Quantum backends for each dataset size, including `ibm_torino`, `ibm_algiers`, `ibm_cairo`, and `ibm_kyoto`. The 'job time' column specifies the average time each job executes on the respective backend, measured in 'quantum seconds'. While the accuracy and F1-score metrics are provided for completeness, it is important to note that due to the small dataset sizes, these metrics may not accurately reflect the performance of the QSVM algorithm. However, they offer insights into the relative performance across different backends and dataset sizes, providing a basis for comparison. Overall, the table illustrates the iterative nature of our experiments, starting from smaller datasets and progressively scaling up to larger ones while also exploring the performance variation across different IBM Quantum backends.

Data (Train/Test)	Backend	Job Time	Accuracy	F1-Score
4/2	`ibm_torino`	15s	0.5	0.333
	`ibm_algiers`	18s	0.5	0.333
8/4	`ibm_torino`	18s	1	1
	`ibm_algiers`	15s	0.75	0.733
20/10	`ibm_cairo`	16s	0.6	0.6
	`ibm_kyoto`	17s	0.6	0.524

Table 4.9 Experiment Results: QSVM Classification on IBM Quantum Systems With Various Datasets and Backends.

4.5 Discussion

In this section, we discuss the results of the experiments performed on the simulator and IBM Quantum systems and propose optimisation techniques that could elevate the performance of our QSVM algorithm.

The results we obtained from the simulator are remarkably similar to those reported in the referenced paper [12]. Although our results show slightly lower values than the results in [12], it may be attributed to differences in the chosen methodology for sample selection. The authors of the referenced paper [12] did not specify their sample selection process, dataset balance, or if they used the same samples across various dataset sizes. Because of that, we could not precisely mimic the setup conditions and probably have a difference in the used data, which might be one of the reasons why our results differ. We have chosen to select samples randomly, focusing on having the same number of benign and malicious samples to avoid any biases as much as we can. We do not know if their samples do not have any bias, which could propagate to the results.

When we compare the performance of QSVM with quantum feature maps to that of SVM with classical kernels, it is evident that QSVM consistently achieves higher or

similar accuracy. In this case, we have the same input preprocessed data, and the only difference is the chosen kernel. We can see there is a slight increase in accuracy and F1 score with the increase in the number of qubits, which aligns with the hypothesis presented in the reference paper [12] that precision increases with the number of qubits, particularly on small datasets.

Regarding hardware experimentation, our focus was exploring the behaviour of IBM Quantum systems. As mentioned earlier, we have access to the IBM Quantum Platform thanks to the CTU license. We are one of the first users of the IBM Quantum Platform under this licence, so a part of our research process was to explore how the IBM Quantum environment behaves under the workload of jobs.

Despite the relatively small size of our individual jobs in terms of data volume and quantum processing time per job, due to the nature of machine learning tasks, a considerable number of jobs are required for our classification, especially with our current implementation, where one job is required per kernel entry.

Our experimentation involved testing various backends, and for managing a larger number of jobs, we opted to submit them all within a single session. When selecting the least busy system available, we typically encountered queue times of only a few minutes. However, with the busiest system, wait times could extend to several hours, even with a relatively small number of jobs per session. While the quantum processing time required to execute the jobs remained consistent across various backends, differing only by a few seconds, these differences had a notable impact when considering our limited resources and the larger volume of jobs we needed to process.

There are various paths to explore when it comes to optimising our implementation of the QSVM algorithm. We can further experiment with the design of feature maps and modify the gates used in the circuit. We can also combine different data mapping functions.

Additionally, instead of combining classical SVM with a quantum kernel, we could explore a fully quantum version, the quantum variational classifier, which uses a variational quantum circuit to classify in a direct analogy to conventional SVM [14]. However, this approach may face challenges due to the limited number of quantum computational minutes available on the IBM Quantum Platform through the CTU license. If we were to perform training on the quantum computer, not just the evaluation of the kernel matrix, it could consume more quantum processing time.

We can also explore different preprocessing methods for malware samples, such as the 'Ordered Importance Features' method, which was suggested in [12]. Furthermore, we could explore entirely different quantum algorithms for classification, such as Quantum Neural Networks [12].

From the implementation standpoint, we can optimise the number of quantum circuits sent within a single job to the quantum computer. Currently, we send one job per kernel entry, but maximising the job size could reduce the number of jobs and potentially decrease the quantum processing time. Additionally, further exploration of the topology of IBM quantum computers could be helpful. Each computer has a different layout of qubits, and specifying the qubits we want to use for computation may reduce computation time.

Chapter 5
Conclusion

This thesis explores the application of quantum computing for malware classification. The main objectives were to investigate the potential of quantum machine learning in malware classification and to evaluate the performance of the Quantum Support Vector Machine (QSVM) algorithm compared to classical approaches.

Firstly, we introduced the machine learning-based malware classification. We described our data extraction and preprocessing techniques, such as converting binaries to grayscale images and Principal Component Analysis. We presented the Support Vector Machine algorithm with a focus on the kernel functions. We provided a simple introduction to quantum computing, highlighting the key aspects that need to be understood to work with the QSVM algorithm, making it accessible to researchers without a quantum computing background. We studied the QSVM algorithm, which combines a classical SVM algorithm with a quantum kernel, and compared it to the SVM.

We used a publicly available PE Malware Machine Learning dataset for the malware classification, for which we implemented a custom Python module that handles the selection, extraction, and preprocessing of the malware samples. We implemented the QSVM algorithm.

We addressed issues in the original implementation of classes for quantum kernel evaluation in Qiskit Machine Library, namely the inability to split the evaluation process into distinct parts, the absence of transpilation for fidelity circuits and the issue with submitting all the fidelity circuits in one single job to IBM Quantum leading to exceeding the maximum limit for job size. The absence of transpilation is a known issue within the Qiskit community and, at the time of finishing this thesis, has not yet been resolved. These critical issues would prevent the algorithm's execution on the IBM Quantum systems. We managed to fix those issues, as explained in Chapter 3. Our enhancements address critical flaws in the original implementation and pave the way for more efficient and practical utilisation of quantum computing resources in malware classification tasks.

Besides evaluating the algorithm on a local simulator, we also run it on a real quantum computer provided by the IBM Quantum Platform. We evaluated its performance using a publicly available PE Malware Machine Learning dataset. We tested how the real IBM Quantum computers work under the workload of many computation jobs in combination with the license provided to CTU from IBM. We were one of the first users

from the entire university to perform computations on the IBM Quantum computers under this license. Our experiments on IBM hardware provide insights into the behaviour and performance of quantum computers, especially in handling large-scale computations for malware classification tasks.

Lastly, we presented the paths for further exploration and optimisation. As this work represents initial steps in leveraging quantum computing for malware detection, there is wide scope for improving our implemented algorithm and its performance and exploring new quantum machine learning techniques.

This thesis lays the groundwork for future research in quantum-enhanced malware classification. Our implementation is compatible with the IBM Quantum interface, allowing anyone to advance the quantum machine learning field further.

Appendix A

Benchmark Results on Simulator

In this appendix, we present the benchmark results obtained from the experiments conducted by [12], focusing on the accuracy and F1-score comparisons of QSVMs with different feature maps. The tables provided here showcase the performance metrics for various datasets and qubit configurations, as detailed in Chapter 4. Additionally, it is important to note that in the Table A.4, we omit the lines where the authors did not provide the F1-score metric, ensuring clarity and consistency in the presentation of results.

Data (Train/Test)	Qubits	ZZ	Pauli	ZZphi	Z
500/100	4	0.78	0.75	0.82	0.81
1000/200	4	0.815	0.79	0.815	0.805
	6	0.81	0.715	0.79	0.785
	7	0.825	0.765	0.785	0.78
2000/400	3	0.765	0.7575	0.8	0.7825
	4	0.8125	0.79	0.8075	0.8
4000/800	4	0.7925	0.786	0.811	0.8075
8000/1600	4	0.805	0.7975	0.814	0.808

Table A.2 Accuracy Comparison Between QSVMs With Different Feature Maps With Depth 2 [12].

Data (Train/Test)	Qubits	ZZ	Pauli	ZZphi	Z
1000/200	6	0.828	0.739	0.817	0.815
	7	0.844	0.791	0.81	0.81
2000/400	3	0.79	0.774	0.819	0.804
	4	0.827	0.814	0.826	0.819

Table A.4 F1-score Comparison Between QSVMs With Different Feature Maps With Depth 2 [12].

Bibliography

1. JUREČEK, Martin. *Algoritmy informační bezpečnosti: Detekce Malware* [online]. 2023. [visited on 2024-04-24]. Available from: https://courses.fit.cvut.cz/NI-AIB/lectures/files/ni_aib_pr8.pdf.
2. MONNAPPA, K. A. *Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware*. Packt Publishing, 2018. ISBN 9781788392501.
3. CyberWire: benign. In: [online]. N2K Networks, Inc., 2024 [visited on 2024-05-08]. Available from: https://thecyberwire.com/glossary/benign.
4. KATZENBEISSER, Stefan; KINDER, Johannes; VEITH, Helmut. Malware Detection. In: *Encyclopedia of Cryptography and Security*. Boston, MA: Springer US, 2011, pp. 752–755. ISBN 978-1-4419-5905-8. Available from DOI: 10.1007/978-1-4419-5906-5_838.
5. SHAHZAD, R.K. *Automated Malware Detection and Classification Using Supervised Learning*. Blekinge Tekniska Högskola, 2024. Blekinge Institute of Technology Doctoral Dissertation Series. ISBN 9789172954755. Available also from: https://books.google.cz/books?id=HEGWOAEACAAJ.
6. MAHAJAN, Ginika; SAINI, Bhavna; ANAND, Shivam. Malware Classification Using Machine Learning Algorithms and Tools. In: *2019 Second International Conference on Advanced Computational and Communication Paradigms (ICACCP)*. IEEE, 2019, pp. 1–8. ISBN 978-1-5386-7989-0. Available from DOI: 10.1109/ICACCP.2019.8882965.
7. ZAHRADNICKÝ, Tomáš; KOKEŠ, Josef. *Reverzní inženýrství: Úvod do reverzního inženýrství, analýza zásobníku* [online]. 2023. [visited on 2024-05-08]. Available from: https://courses.fit.cvut.cz/MI-REV/media/lectures/rev01cz.pdf.
8. AI and machine learning. In: [online]. Gen Digital Inc., 2024 [visited on 2024-05-08]. Available from: https://www.avast.com/technology/ai-and-machine-learning.

9. Machine Learning in Cybersecurity. In: [online]. AO Kaspersky Lab, 2024 [visited on 2024-05-08]. Available from: https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity.

10. BHARTI, Kishor; CERVERA-LIERTA, Alba; KYAW, Thi Ha; HAUG, Tobias; ALPERIN-LEA, Sumner; ANAND, Abhinav; DEGROOTE, Matthias; HEIMONEN, Hermanni; KOTTMANN, Jakob S.; MENKE, Tim; MOK, Wai-Keong; SIM, Sukin; KWEK, Leong-Chuan; ASPURU-GUZIK, Alán. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics* [online]. 2022, vol. 94, no. 1 [visited on 2024-01-29]. ISSN 0034-6861. Available from DOI: 10.1103/RevModPhys.94.015004.

11. GUJJU, Yaswitha; MATSUO, Atsushi; RAYMOND, Rudy. *Quantum Machine Learning on Near-Term Quantum Devices: Current State of Supervised and Unsupervised Techniques for Real-World Applications* [online]. 2023. [visited on 2024-01-29]. Available from: https://doi.org/10.48550/arXiv.2307.00908.

12. BARRUÉ, Grégoire; QUERTIER, Tony. Quantum Machine Learning for Malware Classification. *ArXiv.org* [online]. 2023, p. 30 [visited on 2023-12-09]. Available from: https://doi.org/10.48550/arXiv.2305.09674.

13. SCHÖLKOPF, Bernhard; SMOLA, Alexander J. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. The MIT Press, 2018. ISBN 9780262256933. Available from DOI: 10.7551/mitpress/4175.001.0001.

14. HAVLÍČEK, Vojtěch; CÓRCOLES, Antonio D.; TEMME, Kristan; HARROW, Aram W.; KANDALA, Abhinav; CHOW, Jerry M.; GAMBETTA, Jay M. Supervised learning with quantum-enhanced feature spaces. *Nature*. 2019, vol. 567, no. 7747, pp. 209–212. ISSN 0028-0836. Available also from: https://doi.org/10.48550/arXiv.1804.11326.

15. QISKIT CONTRIBUTORS. *IBM Quantum Platform* [online]. 2024. [visited on 2024-05-04]. Available from: https://quantum.ibm.com/.

16. QISKIT CONTRIBUTORS. *Qiskit: An Open-source Framework for Quantum Computing* [online]. 2024. [visited on 2024-05-04]. Available from: https://github.com/Qiskit.

17. BALÍK, Miroslav; TRÁVNÍČEK, Jan; VAGNER, Ladislav; VOGEL, Josef. *Programování a algoritmizace 1: Algoritmy a programy, základní podpora vývoje* [online]. 2023. [visited on 2024-04-29]. Available from: https://courses.fit.cvut.cz/BI-PA1/@master/media/lectures/l01-alg-cz.pdf.

18. UCCI, Daniele; ANIELLO, Leonardo; BALDONI, Roberto. Survey of machine learning techniques for malware analysis. *Computers & Security*. 2019, vol. 81, pp. 123–147. ISSN 01674048. Available from DOI: 10.1016/j.cose.2018.11.001.

19. NATARAJ, L.; KARTHIKEYAN, S.; JACOB, G.; MANJUNATH, B. S. Malware Images: Visualization and Automatic Classification. In: *Proceedings of the 8th International Symposium on Visualization for Cyber Security* [online]. New York, NY, USA: ACM, 2011, pp. 1–7 [visited on 2024-01-29]. ISBN 9781450306799. Available from DOI: 10.1145/2016904.2016908.

20. JUREČEK, Martin. *Algoritmy informační bezpečnosti: Techniky detekce malwaru založené na strojovém učení I* [online]. 2023. [visited on 2024-04-24]. Available from: https://courses.fit.cvut.cz/NI-AIB/lectures/files/ni_aib_pr9.pdf.
21. Distribution of Malware and PUA by Operating System. In: *AV-ATLAS* [online]. AV-TEST, 2024 [visited on 2024-04-25]. Available from: https://portal.av-atlas.org/malware.
22. PE Format. In: *Microsoft Learn* [online]. Microsoft, 2024 [visited on 2024-04-25]. Available from: https://learn.microsoft.com/en-us/windows/win32/debug/pe-format.
23. What is principal component analysis (PCA)? In: [online]. IBM, 2024 [visited on 2024-05-08]. Available from: https://www.ibm.com/topics/principal-component-analysis.
24. SCHÖLKOPF, B.; MIKA, S.; C.J.C., Burges.; KNIRSCH, P.; MULLER, K.R.; RATSCH, G.; SMOLA, A.J. Input space versus feature space in kernel-based methods. *IEEE Transactions on Neural Networks*. 1999, vol. 10, no. 5, pp. 1000–1017. ISSN 10459227. Available from DOI: 10.1109/72.788641.
25. HEARST, M.A.; DUMAIS, S.T.; OSUNA, E.; PLATT, J.; SCHOLKOPF, B. Support vector machines. *IEEE Intelligent Systems and their Applications*. 1998, vol. 13, no. 4, pp. 18–28. ISSN 1094-7167. Available from DOI: 10.1109/5254.708428.
26. JUREČEK, Martin. *Algoritmy informační bezpečnosti: Techniky detekce malwaru založené na strojovém učení II* [online]. 2023. [visited on 2024-04-24]. Available from: https://courses.fit.cvut.cz/NI-AIB/lectures/files/ni_aib_pr10.pdf.
27. NIELSEN, Michael A.; CHUANG, Isaac L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. ISBN 9780511976667. Available from DOI: 10.1017/CBO9780511976667.
28. DOMBEK, Daniel; KALVODA, Tomáš; KLEPRLÍK, Luděk; KLOUDA, Karel. *Lineární algebra: Studijní text* [online]. 2020. [visited on 2024-03-30]. Available from: https://kam.fit.cvut.cz/deploy/bi-lin//lin-text.pdf.
29. QISKIT CONTRIBUTORS. *Qiskit Textbook: Introduction course* [online]. Github, 2023 [visited on 2024-05-04]. Available from: https://github.com/Qiskit/textbook/blob/main/notebooks/intro/what-is-quantum.ipynb.
30. PHAN, Anna. *Qiskit Global Summer School 2021: Introduction to Quantum Kernels and SVMs* [online]. 2021. [visited on 2023-12-09]. Available from: https://github.com/Qiskit/platypus/blob/main/notebooks/summer-school/2021/resources/lab-notebooks/lab-3.ipynb.
31. QISKIT CONTRIBUTORS. *Qiskit Machine Learning* [online]. 2024. [visited on 2024-05-04]. Available from: https://qiskit-community.github.io/qiskit-machine-learning/.

32. QISKIT CONTRIBUTORS. *Qiskit Machine Learning: FidelityQuantumKernel* [online]. 2024. [visited on 2024-05-08]. Available from: https://qiskit-community.github.io/qiskit-machine-learning/stubs/qiskit_machine_learning.kernels.FidelityQuantumKernel.html.

33. QISKIT CONTRIBUTORS. *Qiskit Machine Learning: BaseKernel* [online]. 2024. [visited on 2024-05-08]. Available from: https://qiskit-community.github.io/qiskit-machine-learning/stubs/qiskit_machine_learning.kernels.BaseKernel.html#qiskit_machine_learning.kernels.BaseKernel.

34. QISKIT CONTRIBUTORS. *Qiskit Algorithms: BaseStateFidelity*. 2024. Available also from: https://qiskit-community.github.io/qiskit-algorithms/stubs/qiskit_algorithms.state_fidelities.BaseStateFidelity.html#qiskit_algorithms.state_fidelities.BaseStateFidelity.

35. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Sampler* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit/qiskit.primitives.Sampler.

36. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: PauliFeatureMap* [online]. 2023. [visited on 2023-12-10]. Available from: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.PauliFeatureMap.

37. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: ZZFeatureMap* [online]. 2023. [visited on 2023-12-10]. Available from: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.ZZFeatureMap.

38. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: ZFeatureMap* [online]. 2023. [visited on 2023-12-10]. Available from: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.ZFeatureMap.

39. LESTER, Michael. *PE Malware Machine Learning Dataset* [online]. [visited on 2024-01-29]. Available from: https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/.

40. THE HDF GROUP. *The HDF5 Library & File Format* [online]. 2006. [visited on 2024-05-08]. Available from: https://www.hdfgroup.org/solutions/hdf5/.

41. PAVLOV, Igor. *7-Zip* [online]. 2024. [visited on 2024-05-08]. Available from: https://www.7-zip.org/.

42. PEDREGOSA, F.; VAROQUAUX, G.; GRAMFORT, A.; MICHEL, V.; THIRION, B.; GRISEL, O.; BLONDEL, M.; PRETTENHOFER, P.; WEISS, R.; DUBOURG, V.; VANDERPLAS, J.; PASSOS, A.; COURNAPEAU, D.; BRUCHER, M.; PERROT, M.; DUCHESNAY, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

43. SCIKIT-LEARN DEVELOPERS. *scikit-learn: sklearn.svm.SVC* [online]. 2024. [visited on 2024-05-08]. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

44. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: SamplerV2* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.SamplerV2.

45. QISKIT CONTRIBUTORS. *Qiskit Algorithms: ComputeUncompute* [online]. 2024. [visited on 2024-05-08]. Available from: https://qiskit-community.github.io/qiskit-algorithms/stubs/qiskit_algorithms.state_fidelities.ComputeUncompute.html#qiskit_algorithms.state_fidelities.ComputeUncompute.
46. QISKIT CONTRIBUTORS. *IBM Quantum Platform: Update to Qiskit Runtime Primitives* [online]. 2024. [visited on 2024-04-27]. Available from: https://docs.quantum.ibm.com/announcements/product-updates/2024-02-14-qiskit-runtime-primitives-update#update-to-qiskit-runtime-primitives.
47. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Configure runtime compilation for Qiskit Runtime* [online]. 2024. [visited on 2024-04-27]. Available from: https://docs.quantum.ibm.com/run/configure-runtime-compilation.
48. QISKIT CONTRIBUTORS. *Qiskit Runtime IBM Client: Sampler fails to run FidelityKernel even if circuits are transpiled* [online]. 2024. [visited on 2024-04-27]. Available from: https://github.com/Qiskit/qiskit-ibm-runtime/issues/1519.
49. QISKIT CONTRIBUTORS. *Qiskit Algorithms: ISA circuit support for latest Runtime* [online]. 2024. [visited on 2024-04-27]. Available from: https://github.com/qiskit-community/qiskit-algorithms/issues/164.
50. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Maximum execution time for a Qiskit Runtime job or session* [online]. 2024. [visited on 2024-04-27]. Available from: https://docs.quantum.ibm.com/run/max-execution-time.
51. QISKIT CONTRIBUTORS. *Qiskit Runtime IBM Client: Qiskit Runtime* [online]. 2024. [visited on 2024-05-04]. Available from: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/runtime_service.
52. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Session* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.Session.
53. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: PassManager* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.PassManager.
54. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Preset Passmanagers* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit/transpiler_preset#generate_preset_pass_manager.
55. CloudFIT. In: [online]. Faculty of Information Technology, CTU in Prague, 2024 [visited on 2024-05-05]. Available from: https://help.fit.cvut.cz/cloud-fit/index.html.
56. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Estimate job run time* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/run/estimate-job-run-time.
57. SCIKIT-LEARN DEVELOPERS. *scikit-learn: sklearn.metrics.f1_score* [online]. 2024. [visited on 2024-05-08]. Available from: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1%7B%5C_%7Dscore.html.

58. QISKIT CONTRIBUTORS. *IBM Quantum Documentation: Quantum Circuits* [online]. 2024. [visited on 2024-05-08]. Available from: https://docs.quantum.ibm.com/api/qiskit/circuit.

Contents of Attached Media

README.txt .... a brief summary of the contents
src .... the source codes directory