

Quantum Computing Methods for Malware Detection

Eliška Krátká^[0009-0000-5152-4670] and
Aurél Gábor Gábris^[0000-0002-2671-6328]

Abstract In this paper, we explore the potential of quantum computing in enhancing malware detection through the application of Quantum Machine Learning (QML). Our main objective is to investigate the performance of the Quantum Support Vector Machine (QSVM) algorithm compared to SVM. A publicly available dataset containing raw binaries of Portable Executable (PE) files was used for the classification. The QSVM algorithm, incorporating quantum kernels through different feature maps, was implemented and evaluated on a local simulator within the Qiskit SDK and IBM quantum computers. Experimental results from simulators and quantum hardware provide insights into the behavior and performance of quantum computers, especially in handling large-scale computations for malware detection tasks. The work summarizes the practical experience with using quantum hardware via the Qiskit interfaces. We describe in detail the critical issues encountered, as well as the fixes that had to be developed and applied to the base code of 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.

1 Introduction

Quantum computing has opened up new possibilities for addressing complex computational problems that classical computers struggle to solve. Quantum computers exploit the principles of quantum mechanics, such as superposition and entangle-

Eliška Krátká
Faculty of Information Technology, Czech Technical University in Prague, Prague, Czechia, e-mail: kratkeli@fit.cvut.cz

Aurél Gábor Gábris
Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Prague, Czechia, e-mail: gabris.aurel@fjfi.cvut.cz

ment, which allow them to perform parallel computations and potentially achieve exponential speedup for specific tasks.

In recent years, a significant milestone in quantum computing has been the development of noisy intermediate-scale quantum (NISQ) devices [26]. NISQ devices are the class of quantum computers characterized by their intermediate scale in the number of qubits. 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 the noise in the quantum hardware [4]. 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 [4].

One promising research area on the presently available NISQ computers is the combination of quantum computing and machine learning, known as Quantum Machine Learning (QML). Over the last decade, there have been significant advances in the QML field, including conventional machine learning algorithms that can be enhanced using quantum techniques and entirely new quantum machine learning algorithms explicitly designed to run on quantum computers [5].

In this chapter, we explore the potential of applying QML to a practical problem from information security: malware detection. Malware detection is the process of identifying malicious software. This task is typically framed as a binary classification problem, where the goal is to distinguish between two categories: malicious and benign (harmless) software [39]. Machine learning models are well-suited for solving this type of problem. Given the increasing volume and variety of new threats, malware detection based on machine learning has become a popular approach in modern antivirus programs [1, 17].

Our research focuses on the Quantum Support Vector Machine (QSVM) algorithm and its application to malware detection. A key part of the work involves running the QSVM on quantum computers from IBM. Executing the algorithm on real quantum hardware presents unique challenges, making the process more difficult than running the same calculations on quantum computer simulators.

The QSVM algorithm combines the conventional Support Vector Machine (SVM) with a quantum kernel. We study and implement the quantum kernel using a quantum computer. The SVM model is then fitted with the precomputed quantum kernel and trained on a classical computer. We assess the performance of the QSVM in terms of the model's accuracy and compare its results to SVM using conventional kernels.

We organize our work into three parts, with each covered in the following sections. In Section 2, we provide the necessary background on the quantum computing aspects of our research, explaining how quantum kernels in QSVM differ from conventional ones and how they are computed using quantum computers. We then introduce Qiskit [16] and IBM Quantum [14], highlighting their roles in implementing the algorithm and accessing quantum hardware. Section 3 focuses on our implementation, emphasizing the challenges faced during the development process for quantum processors and how we addressed them. Section 4 presents the performed experiments and the achieved results.

2 Background

In this section, we explain the core concepts and principles underlying quantum computation, which are necessary to understand before we focus on the QSVM algorithm. We examine the theoretical foundations of QSVM, describe how it operates on quantum computers and the advantages it offers over its classical counterpart.

Furthermore, we introduce the Qiskit and its machine learning module [16, 31]. Through Qiskit, researchers can develop quantum algorithms and access quantum computers from IBM, which are available through the IBM Quantum platform [14]. We discuss the role of Qiskit in our research in implementing QSVM and performing quantum experiments on real quantum processors.

2.1 Terminology

We follow the definitions and explanations of key terms laid down by Nielsen and Chuang in [23]. The only prerequisite is a basic understanding of elementary linear algebra and classical computing.

The standard notation for linear algebra in quantum mechanics and quantum computing is known as bracket 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. An inner product of two vectors $|\psi\rangle$ and $|\varphi\rangle$ is denoted by $\langle\varphi|\psi\rangle$. The inner product is formally a map

$$\langle\cdot, \cdot\rangle : V \times V \rightarrow \mathbb{C},$$

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 [2]. Quantum computing operates within a finite-dimensional Hilbert space, which in this context is equivalent to a complex vector space $\mathbb{C}^n$ with the inner product.

A quantum bit, known as a 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 utilizes qubits.

A state of the qubit, the quantum state, is described by a unit vector in a two-dimensional Hilbert Space, which we further refer to as a quantum state 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

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

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

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

$$|\alpha|^2 + |\beta|^2 = 1.$$

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. The absolute squares of the probability amplitudes give the probabilities of the possible outcomes occurring when measured in the computational basis.

Measurement plays an essential role in quantum computing. While the state of a classical bit can be observed without altering it, the qubit in superposition cannot be directly measured without affecting its quantum state. Upon measurement, the qubit *collapses* into one of the basis states, giving an outcome of either $|0\rangle$ with a probability of $|\alpha|^2$ or $|1\rangle$ with a probability of $|\beta|^2$. 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, which means they allow for the exact reconstruction of the original input information after processing. When a quantum gate is applied to a set of qubits, the operation can be undone without any loss of information. Because quantum gates are reversible, they preserve the quantum information encoded in qubits.

In quantum computing, quantum gates are represented by the unitary operators. Unitary operators are mathematical operators represented by matrices that satisfy the condition

$$U^\dagger U = I,$$

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 the quantum computation.

Within quantum circuits, interference emerges as a 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 to the quantum circuit. By carefully designing the sequence of gates, the interference ef-

fects 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 specific 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 refers to a special relationship between qubits that allows them to become correlated in such a way that the state of one qubit directly influences the state of another, regardless of their individual locations within a quantum system. When two qubits are entangled, they form a single quantum state that cannot be described independently, which means that the measurement of one qubit will instantly determine the state of the other qubit, even if they are not physically connected. Entanglement enables quantum computers to perform calculations on multiple states simultaneously and exhibit non-local correlations, exponentially increasing processing power for certain problem domains.

2.2 Quantum Machine Learning

Quantum machine learning explores the potential benefits of using quantum algorithms and quantum computing hardware to enhance classical machine learning tasks [4]. We focus on enhancing the SVM algorithm, which is a widespread tool in the domain of machine learning-based malware detection [40], by combining it with a quantum kernel, estimated using a quantum computer.

The quantum advantage lies in using a kernel, which is hard to estimate classically [6]. QSVM is based on quantum circuits that are hard to simulate due to their unique quantum properties, such as entanglement and superposition. QSVM promises to achieve better accuracy than conventional SVM across various problem domains, including malware detection [3].

In this section, we explain the concept of kernels in SVM and introduce the quantum kernel. We also provide an overview of the tools used to implement and run software on quantum computers, specifically Qiskit and IBM Quantum. Additionally, we present related work in the field and discuss how it connects to our research.

2.2.1 QSVM

In SVM classification, the algorithm seeks to find an optimal decision boundary 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. Many real-world datasets are not inherently linearly separable, which is why kernels are used in SVM. Kernels map the input features to a new, possibly higher-dimensional space where the data may become more easily separable.

A feature map $\phi(x)$ is a function which maps each data point x from the original input feature space to a new transformed feature space with a higher dimensionality. The kernel function

$$k(x, y) = (\phi(x) \cdot \phi(y))$$

computes the dot product between two vectors x and y in the higher-dimensional feature space. Instead of explicitly computing the transformed vectors $\phi(x)$ and $\phi(y)$, the kernel function computes the dot product directly from the original input space without explicitly performing the mapping, which allows SVM to operate efficiently in high-dimensional space [34].

There are various types of SVM kernels, such as polynomial, RBF, and sigmoid kernels. Different kernel functions define different ways of projecting the data and measuring similarity between points. QSVM combine SVM with a quantum kernel, computed using a quantum computer. The SVM model is then fitted with the precomputed quantum kernel and trained on a classical computer.

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 computational framework. The kernel 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 data to be transformed into quantum state space before processing. In the context of QSVM, we refer to this transformation as data encoding. Once the data are encoded, the quantum kernel function is applied to compute correlations between the quantum states. Therefore, estimating the 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 $\phi(x)$. It is a parameterized quantum circuit that maps a classical feature vector x to its 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 [25]. The index $\phi(x)$ in the $U_{\phi(x)}$ refers to the specific parameterization of the operation U , which depends on the classical feature vector x . Quantum gates and operations can be parameterized by certain variables, which affect 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.

The quantum kernel function

$$k(x, y) = \langle\phi(x) | \phi(y)\rangle = |\langle\phi(x) | \phi(y)\rangle|^2$$

is defined as the state overlap of the two data-encoded feature vectors from the quantum state space and represents the similarity between them [6]. A larger value of $k(x, y)$ indicates that the classical data points x and y are close in feature space [25].

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

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

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

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

where the feature map $\phi(x)$ is described as the unitary operation $U_{\phi(x)}$ applied to the initial state $|0^n\rangle$ [6, 25].

2.2.2 Qiskit

In our work, we rely on Qiskit [16] to implement the QSVM algorithm. Qiskit is an open-source software development kit for Python that enables users to design and implement algorithms for quantum computers at the level of quantum circuits. These algorithms can be executed locally on simulators or on quantum computers from IBM.

IBM provides access to the quantum computers, known as IBM Quantum systems, via cloud through the IBM Quantum platform [14], allowing researchers to experiment with real quantum hardware without needing specialized infrastructure. IBM processors fall under the NISQ devices category, meaning they operate with a limited number of qubits and suffer from errors due to the noise in the quantum hardware [4, 26]. As of September 2024, eleven quantum processors are available on the IBM Quantum platform. Three quantum processors are freely available to the public, while the remainder is accessible via a premium plan.

Qiskit Machine Learning [31] is a module within Qiskit which provides tools for quantum machine learning tasks, including classical machine learning algorithms that can be enhanced using quantum computing techniques and entirely new quantum machine learning algorithms designed to run on quantum computers. We focus on introducing the classes implementing the quantum kernel within the Qiskit Machine Learning module. Understanding those classes is essential for effective integration of quantum-based kernels into the SVM.

The quantum kernel interface is abstractly defined by the `BaseKernel` [32] class. It specifies the `evaluate` method for constructing a kernel matrix from a given dataset, which is compatible with the Quantum Support Vector Classifier within Qiskit Machine Learning, as well as other kernel-based machine learning algorithms in established classical frameworks (for example, `scikit-learn` [24]). Each entry in the kernel matrix is the result of the kernel function, defined as

$$K(x, y) = \langle f(x) | f(y) \rangle,$$

where x, y are n -dimensional inputs and f is a map from an n -dimensional to an m -dimensional space. The quantum kernel algorithm computes the kernel matrix given the datapoints x and y , and the feature map f , all of dimension n .

The `FidelityQuantumKernel` [33] implements the `BaseKernel` interface. The kernel function is defined as the overlap of two quantum states x and y ,

$$K(x, y) = |\langle \phi(x) | \phi(y) \rangle|^2,$$

constructed using the feature map $\phi(x)$. The `FidelityQuantumKernel` requires a fidelity primitive, which computes the fidelity between quantum states based on the `BaseStateFidelity` [27] algorithm introduced in Qiskit.

The `BaseStateFidelity` class is an interface that calculates state fidelities (state overlaps) for pairs of (parameterized) quantum circuits. The fidelity calculation is generally defined as the state overlap

$$|\langle \psi(x) | \phi(x) \rangle|^2,$$

where ψ and ϕ represent the states, and x and y are optional parameterizations of these states. The default fidelity primitive in the `FidelityQuantumKernel` is the `ComputeUncompute` [28], which implements the `BaseStateFidelity` interface.

The data encoding process allows the quantum kernel to generate correlations between variables that are difficult to achieve using classical methods alone. The feature map must be based on quantum circuits that are hard to simulate classically [6] to obtain the quantum advantage over conventional kernels used in SVM. We describe feature maps based on the work of Havlicek et al. [6] and implemented in Qiskit, which we later use in our experiments, namely `PauliFeatureMap` [8], `ZZFeatureMap` [10] and `ZFeatureMap` [9].

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, specified 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. Data encoding is achieved by applying the unitary operation $U_{\phi(x)}$ to the initial state, which in the case of `PauliFeatureMap` is defined as

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

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

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

is the data mapping function, which can be customized.

The `ZZFeatureMap` is a special case of the `PauliFeatureMap`, where the `ZZ` denotes the use of 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 specifically chosen as Pauli-Z matrices, resulting in a product term that captures the `ZZ` interaction between qubits.

The `ZFeatureMap` is another specific case of the `PauliFeatureMap`. Unlike 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 may mean that the `ZFeatureMap` does not provide a quantum advantage for certain tasks, its effectiveness still depends on the specific problem being addressed.

The last feature map we later use in our experiments is not implemented in Qiskit directly. However, it is based on the `ZZFeatureMap` with a custom data mapping function, defined as

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

where S is a set of qubit indices that describes the connections in the feature map [25]. We later refer to this feature map as the `ZZphiFeatureMap`.

All the feature maps mentioned can have a custom circuit depth specified by the `depth` parameter, which refers to the number of layers of quantum gates or operations applied to the input qubits to transform classical data into a quantum state. In the `PauliFeatureMap`, each layer typically consists of single-qubit rotations and entangling gates involving Pauli matrices. The depth of the `PauliFeatureMap` is determined by the number of these layers applied to the input qubits. The depth of a `PauliFeatureMap`, or any quantum circuit, represents the complexity of the circuit and the number of sequential operations used to encode classical data into a quantum state. A deeper circuit may capture more complex patterns in the data but may also require more computational resources.

2.3 Related Work

The inspiration for our research is laid by the work of Barrué and Quertier [3], which provides insights into the performance of quantum machine learning algorithms in the context of malware detection. Notably, to date, this is the only paper that addresses malware detection through quantum computing methods while also performing experiments on IBM quantum computers rather than solely relying on simulators. Their work investigates QSVM alongside Quantum Neural Networks,

and their findings underscore the potential of QSVM to outperform SVM with conventional kernels, mainly when operating with smaller datasets. Their research is heavily focused on experiments using only Qiskit’s simulator. In contrast, our approach differs by concentrating on experiments with real quantum computers, which allows us to assess the practical challenges and performance of QSVM in a more realistic setting.

However, we encountered several challenges when replicating their results due to the paper’s lack of detailed experimental descriptions and parameter specifications. More importantly, they do not specify how many qubits and shots were used or which processors were utilized when conducting experiments on IBM Quantum devices. Additionally, they are not consistent with their metrics, such as not consistently measuring the F1-score, and if so, it is not clear to which parameters it belongs.

3 Implementation

Our implementation consists of two main Python modules: the `peml` module, which is responsible for preprocessing the chosen dataset, and the `svm` module, which implements the SVM classification interface with both quantum and classical kernels. These modules are designed to function independently. The `peml` module focuses on preprocessing the specified dataset. The `svm` module can classify any dataset that adheres to the input format. The source code, along with detailed documentation, is available on GitLab [18].

The QSVM class within the `svm` module implements the interface for QSVM classification using both the local simulator and quantum computers from IBM. Our implementation is based on two main classes from the Qiskit Machine Learning module [31], `ComputeUncompute` [28] and `FidelityQuantumKernel` [33], which we previously described in detail in Section 2. However, a significant limitation of these classes, and the Qiskit Machine Learning module as a whole, is that they are designed to run only on Qiskit’s local quantum computer simulators. We aim to apply QSVM on real quantum hardware, specifically IBM’s quantum computers.

In this section, we explain the challenges encountered when running the code on actual quantum hardware and detail how and why we modified the source code of these two classes to overcome these obstacles, enabling execution on quantum devices. While the challenges are explained in the context of QSVM, they are universal to any large-scale practical quantum machine learning problem, not limited to QSVM, that requires substantial data processing on quantum hardware. For instance, similar issues would arise when implementing other models, such as neural networks.

3.1 Modifications for Quantum Hardware

The implementation of the `ComputeUncompute` and `FidelityQuantumKernel` classes has three significant limitations that prevent the code from running on quantum hardware: inability to split the evaluation process, lack of transpilation for fidelity circuits and submission of all fidelity circuits in a single computational job, which exceeds the maximum job size limit. In the modified versions of the classes, we address these issues. Our improvements enable efficient resource utilization, ensure compatibility with IBM Quantum hardware, and enhance scalability for real-world machine learning applications.

However, a major ongoing challenge is that Qiskit and its Qiskit IBM Runtime module constantly evolve, but often without maintaining minimal backward compatibility, which makes it difficult to keep the implementation up to date, and parts of the project may become outdated even in terms of few months. Nonetheless, as mentioned earlier, these three problems are not specific only to the QSVM implementation in Qiskit Machine Learning. They are general issues that need to be considered when working with real IBM Quantum hardware and should be accounted for in any project design.

3.1.1 Evaluation Process Must Wait for the Job Completion

The original implementation of the classes lacks the ability to split the evaluation process into two distinct parts: submitting the computational jobs to IBM Quantum and processing the completed jobs. As a result, the classification process must run continuously while awaiting job execution on the IBM Quantum platform, which can take several days, depending on the job queue. This inefficiency not only consumes unnecessary resources but also restricts the scalability of the evaluation process, particularly when dealing with large datasets.

To address this limitation, we introduced a solution that divides the process into two parts by adding helper methods to handle job submission and post-processing separately. In the first part, jobs are submitted to IBM Quantum to calculate the entries of the kernel matrix. Once the quantum jobs are completed, the second phase processes the results and evaluates the kernel matrices using the saved configuration.

3.1.2 Missing Transpilation

Another issue is the absence of transpilation for the fidelity circuits before submitting computational jobs to IBM Quantum, which is a critical flaw in the original implementation. Transpilation refers to transforming quantum circuits to use only instructions supported by the targeted quantum processor. This transformation ensures compatibility and efficient execution on the actual quantum hardware. As of March 1, 2024, IBM Quantum introduced a significant update to improve the speed and efficiency of quantum computation [11, 15]. Circuits and observables must now be

transformed to use only the instruction set architecture (ISA) supported by the target quantum system, meaning that all circuits must be transpiled before submission for execution.

Without transpilation, the fidelity circuits in QSVM cannot be executed on IBM processors, which makes the `ComputeUncompute` and `FidelityQuantumKernel` classes unusable for real-world applications. It is worth noting that the transpilation issue is known and tracked by the Qiskit community, affecting several other classes beyond those discussed here, yet as of the completion of this work, it remains unresolved [29, 30].

To address this issue, we added logic to ensure the fidelity circuits are transpiled before submission to the target quantum processor. However, while transpilation is necessary for executing quantum circuits on IBM hardware, it is not straightforward. It involves a series of optimizations that can sometimes alter the properties of the original circuit. During transpilation, circuits are transformed to match the constraints of the target system, such as available gates and qubit connectivity. However, this can result in issues such as increased circuit depth, which directly impacts execution time and noise levels.

Additionally, circuits might be mapped to sub-optimal qubits for the specific computation, further degrading performance. In some cases, the original structure of the circuit, which was carefully designed for a specific behavior, may be lost or compromised during the transpilation process. These challenges make transpilation a problem of its own, requiring careful consideration when working with real quantum hardware, as the efficiency and accuracy of the quantum computation can be significantly affected.

3.1.3 Exceeding Maximum Job Size Limit

The original classes submit all the fidelity circuits in a single computational job. While this approach works for local simulation, it becomes impractical for larger datasets on IBM Quantum systems. The job size often exceeds the maximum allowed limit [13], preventing circuit execution and significantly limiting the usability of these classes, especially with larger datasets.

To address this limitation, we implemented a one-job-per-kernel-entry approach, where each fidelity circuit responsible for computing one entry of the kernel matrix is computed in an individual job. We avoid unnecessary queuing delays by submitting these jobs in a session, enhancing overall efficiency and scalability.

4 Experiments

This section describes the experiments we performed and presents our results. We categorize the experiments into two types: those run on Qiskit’s local simulator and those executed on IBM Quantum processors. First, we outline the dataset and

evaluation metrics used, followed by a detailed description of the experiments within each category.

4.1 Dataset

We used the publicly available PE Malware Machine Learning Dataset [19] for our experiments. A key benefit of this dataset is that it provides the raw binary files themselves rather than just metadata extracted from the samples.

The dataset consists of raw binaries of PE files, such as .exe or .dll files, and contains 201,549 labeled 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 are sourced primarily from platforms like VirusShare [41], MalShare [20], and the-Zoo [22]. Most of the legitimate files come from various instances of Windows 7, featuring a variety of installed software. However, there is a potential bias towards files associated with Microsoft products among them.

Directly feeding raw binary files into the model is impractical due to their unstructured nature and the volume of data. Unstructured data lacks the organization and formatting necessary for practical analysis, and the amount of information in raw binary files makes it challenging for the model to extract meaningful patterns. Therefore we applied preprocessing techniques such as conversion to grayscale images [21] and Principal Component Analysis [7] to transform the raw binaries into informative feature vectors from which the model can learn.

We converted the samples into grayscale images, adjusting their width based on the size of the binary content according to the predefined size ranges from Nataraj et al.[21]. The images were resized to a uniform size while maintaining their aspect ratio and flattened into one-dimensional feature vectors. To align the dimensionality of the feature vectors with the number of qubits used in our experiments, we applied Principal Component Analysis (PCA) for dimensionality reduction. Although it may seem counterintuitive to convert binary files to images before applying PCA, we followed this approach to replicate the setup and results presented in the paper by Barrué and Quertier [3], described in Section 2. However, the image-construction process might not be necessary, and directly applying PCA to the binary data could have avoided the resizing and flattening steps. We randomly selected samples for the training and testing groups, ensuring an equal number of benign and malicious samples to create balanced datasets for our experiments.

4.2 Evaluation Metrics

We adopt two metrics for evaluating the performance of models, accuracy and F1 score. Both metrics rely on the following terms, true positives, true negatives, false positives, and false negatives.

- True positives (**TP**) refer to the number of malware samples that are correctly classified as malware.
- True negatives (**TN**) represent the number of benign samples correctly classified as benign.
- False positives (**FP**) refer to the number of benign samples that are incorrectly classified as malware.
- False negatives (**FN**) represent the number of malware samples that are incorrectly classified as benign (missed malware detections).

Accuracy represents the proportion of correctly classified samples (both malware and benign) out of the total number of classifications [35]. It provides a straightforward indication of the model's overall correctness, reflecting how often it gets the classification right.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

F1 score is defined as a harmonic mean of precision and recall [36]. Precision measures how many of the samples classified as malware are truly malware [37]. For example, in malware detection, precision tells us what fraction of the files the model flagged as malware are actually malicious. Recall measures how many of the actual malware samples were correctly classified [38]. It tells us how well the model performs in detecting malware. It indicates the proportion of all malware samples that the model successfully identifies.

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

The F1 score combines precision and recall into a single metric, which can be especially useful when false positives and false negatives carry different consequences. In the context of malware detection, a high F1 score ensures that the model is not only accurate but also balances identifying actual malware and avoiding false positives, which can be critical when both false negatives (undetected malware) and false positives (benign files flagged as malware) are undesirable.

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 \times TP}{2 \times TP + FP + FN}$$

If there are no TP, FN, or FP samples (for example, in cases where no malware samples were predicted), the F1 score defaults to zero to avoid division errors.

4.3 Experimental Results

Our primary focus was on testing and assessing performance on real quantum hardware. While simulators are flexible and convenient, they do not fully capture the complexity of quantum behavior under real conditions. They cannot fully emulate the effects of quantum noise in real quantum systems and come at a higher computational cost. However, testing the implementation first on a simulator is a crucial part of any quantum computing experiment. Simulators serve as a benchmark, helping to verify that the quantum circuit is correctly implemented.

IBM Quantum computers offer the opportunity to validate algorithms under real-world conditions. Despite this advantage, running experiments on quantum computers introduces several challenges that affect the consistency and scalability of the results, as discussed in the previous section. Due to these limitations, the experiments conducted on IBM Quantum processors differ from those run on simulators. We could not run as many experiments on the hardware as on the simulator due to implementation constraints and limited access to computational resources.

On both platforms, our goal was to evaluate the performance of our QSVM implementation and compare it with conventional SVM using kernels like polynomial or RBF. We focused primarily on the model's accuracy and investigated whether the QSVM demonstrated any quantum advantage in improved performance over classical methods.

4.3.1 Simulator

On the local simulator in Qiskit, we tested QSVM classification with datasets of various sizes, ranging from 500 training samples and 100 test samples to 8000 training samples and 4000 test samples. For comparison, we performed SVM classification using classical kernels to evaluate how QSVM performs against conventional methods. Our goal was to replicate the experimental setup from Barrué and Quertier [3] as closely as possible and determine whether our implementation achieved similar performance improvements, particularly on smaller datasets.

A notable finding from Barrué and Quertier [3] is that quantum kernels, especially the `ZZFeatureMap`, demonstrated up to a 2.5% improvement in accuracy over conventional SVM kernels in specific configurations. Their results suggest that QSVM may have an advantage in scenarios with limited dataset size. We aimed to verify these claims by comparing the performance of QSVM with classical SVM kernels across various dataset sizes.

In the experiments, we used quantum kernels with different feature maps: `ZZFeatureMap` (ZZ), `PauliFeatureMap` (Pauli), `ZZphiFeatureMap` (ZZphi), and `ZFeatureMap` (Z), with the depth of the circuits set to 2. We used 1000 shots for all experiments, as the referenced paper did not specify the shot count. Number of shots refers to 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. Our input data consisted of binaries transformed

into grayscale images of size 64×64, which were preprocessed into feature vectors of dimensions corresponding to the number of used qubits. The same preprocessing method was applied to both quantum and classical experiments, with the kernel being the primary differentiating factor.

The results, presented in Table 1 and Table 2, demonstrate that QSVM consistently matches or outperforms the accuracy and F1 scores of SVM using classical kernels.

The results presented in Table 1 and Table 2 demonstrate that QSVM consistently matches or outperforms the accuracy and F1 scores of SVM using classical kernels. In Figure 1 and Figure 2, we highlight the performance of the kernels based on ZZ and ZZphi feature maps compared to the RBF kernel. Notably, the ZZ and ZZphi kernels exhibit the best performance among the quantum kernels, while the RBF kernel stands out among the classical ones.

Figure 1 displays the F1 score comparison for the ZZ, ZZphi, and RBF kernels with three qubits, illustrating how quantum approaches can remain competitive even with limited qubit resources. In contrast, Figure 2 presents the F1 score comparison for the same kernels with seven qubits, where the quantum kernels (particularly ZZ and ZZphi) achieve their highest F1 scores. Figure 2 provides a more comprehensive understanding of how these kernels scale with increased qubit count and data size, demonstrating the potential of quantum kernels against classical benchmarks like the RBF kernel.

Table 1: Accuracy Comparison

Data (Train/Test)	Qubits	Quantum Kernels				Classical Kernels			
		ZZ	Pauli	ZZphi	Z	Linear	Polynomial	RBF	Sigmoid
500/100	3	0.740	0.790	0.800	0.780	0.750	0.690	0.760	0.510
	4	0.730	0.780	0.800	0.790	0.740	0.720	0.790	0.540
	6	0.660	0.720	0.810	0.810	0.740	0.740	0.810	0.560
	7	0.700	0.800	0.820	0.830	0.740	0.750	0.850	0.620
1000/200	3	0.725	0.675	0.735	0.720	0.705	0.730	0.745	0.575
	4	0.730	0.660	0.735	0.735	0.705	0.720	0.740	0.580
	6	0.745	0.760	0.780	0.775	0.735	0.745	0.790	0.640
	7	0.790	0.735	0.780	0.780	0.730	0.775	0.780	0.640
2000/400	3	0.710	0.730	0.748	0.757	0.718	0.672	0.770	0.603
	4	0.748	0.743	0.775	0.767	0.718	0.685	0.765	0.585
	6	0.777	0.728	0.770	0.780	0.740	0.735	0.782	0.595
	7	0.782	0.767	0.802	0.780	0.743	0.743	0.795	0.583
4000/800	3	0.799	0.784	0.771	0.777	0.766	0.639	0.787	0.671
	4	0.806	0.821	0.771	0.775	0.771	0.637	0.791	0.637
	6	0.830	0.812	0.816	0.800	0.772	0.804	0.830	0.608
	7	0.838	0.805	0.824	0.821	0.771	0.791	0.840	0.616
8000/1600	3	0.783	0.779	0.797	0.796	0.779	0.619	0.804	0.633
	4	0.812	0.792	0.804	0.806	0.781	0.662	0.822	0.630
	6	0.835	0.806	0.819	0.818	0.779	0.734	0.840	0.616
	7	0.851	0.812	0.831	0.821	0.776	0.746	0.845	0.608

Table 2: F1 Score Comparison

Data (Train/Test)	Qubits	Quantum Kernels				Classical Kernels			
		ZZ	Pauli	ZZphi	Z	Linear	Polynomial	RBF	Sigmoid
500/100	3	0.736	0.790	0.797	0.777	0.746	0.662	0.754	0.510
	4	0.729	0.779	0.797	0.788	0.736	0.700	0.787	0.540
	6	0.649	0.716	0.808	0.810	0.736	0.729	0.808	0.560
	7	0.690	0.795	0.819	0.829	0.736	0.738	0.849	0.620
1000/200	3	0.723	0.675	0.732	0.717	0.700	0.728	0.742	0.574
	4	0.729	0.658	0.732	0.731	0.700	0.719	0.737	0.579
	6	0.744	0.756	0.779	0.775	0.730	0.738	0.789	0.640
	7	0.787	0.731	0.779	0.780	0.725	0.771	0.779	0.640
2000/400	3	0.707	0.728	0.747	0.756	0.716	0.651	0.769	0.601
	4	0.746	0.741	0.774	0.767	0.716	0.670	0.764	0.584
	6	0.776	0.726	0.769	0.780	0.739	0.729	0.782	0.595
	7	0.781	0.764	0.802	0.780	0.742	0.737	0.795	0.582
4000/800	3	0.797	0.783	0.769	0.775	0.764	0.612	0.786	0.671
	4	0.805	0.821	0.770	0.773	0.769	0.621	0.790	0.637
	6	0.830	0.811	0.816	0.799	0.771	0.803	0.830	0.607
	7	0.837	0.803	0.823	0.821	0.769	0.790	0.840	0.616
8000/1600	3	0.783	0.779	0.796	0.794	0.778	0.589	0.804	0.633
	4	0.812	0.792	0.803	0.805	0.779	0.646	0.822	0.630
	6	0.835	0.806	0.818	0.818	0.778	0.731	0.840	0.616
	7	0.851	0.812	0.830	0.821	0.775	0.743	0.845	0.607

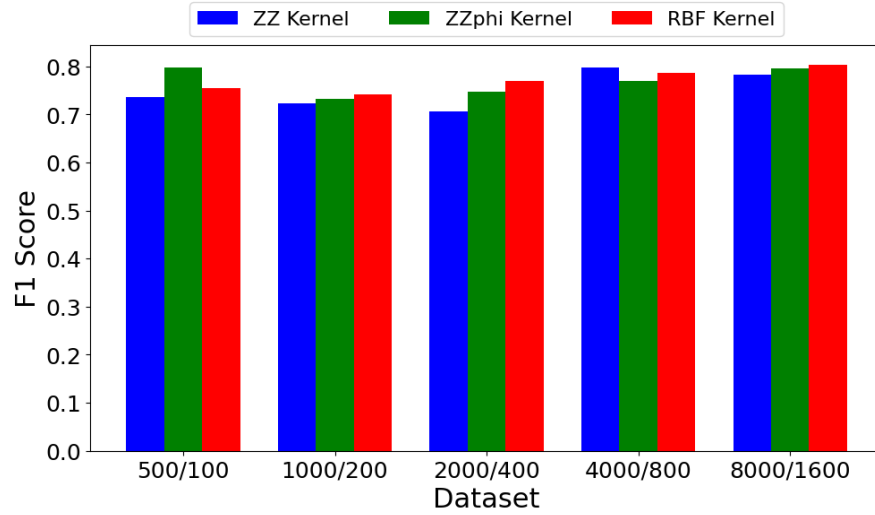


Fig. 1: F1 Score Comparison With 3 Qubits

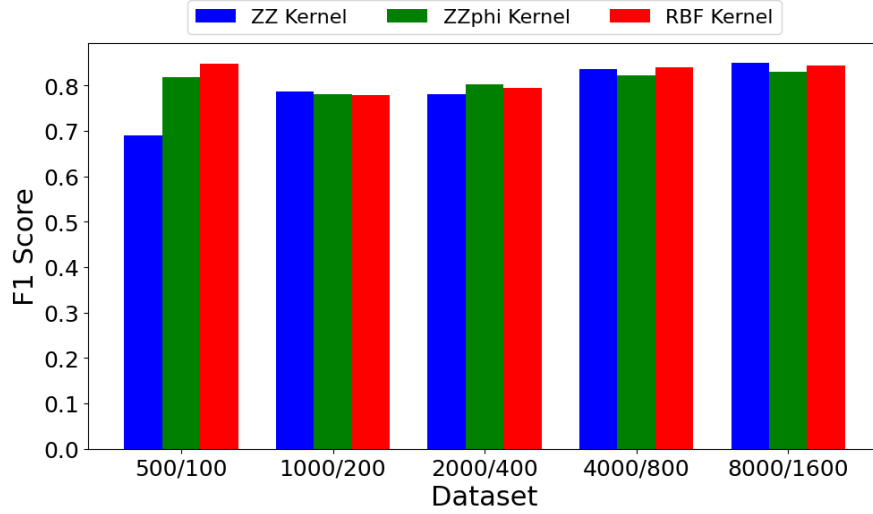


Fig. 2: F1 Score Comparison With 7 Qubits

4.3.2 IBM Quantum Systems

The second phase of our experiments involves QSVM classification using quantum kernels computed on IBM Quantum computers, to which we have access thanks to a license from the Czech Technical University in Prague (CTU).

Inspired by the potential of NISQ computers, our initial goal was to implement and evaluate QSVM primarily on IBM Quantum computers. However, during implementation, we encountered several challenges that significantly altered the course of our experiments, as detailed in Section 3. These challenges stem mainly from limitations within the Qiskit Machine Learning module, particularly regarding transpilation requirements and constraints on job sizes when using IBM Quantum systems.

As a result, we faced limitations when running experiments on real quantum hardware. To mitigate these issues, we implemented a fix involving the addition of transpilation and adopting a one-job-per-kernel-entry approach, as described in Section 3. Transpilation, a critical requirement for executing quantum circuits on IBM Quantum systems, involves adapting circuits to conform to the target quantum system’s ISA. While our fix addressed the critical obstacles, more efficient and optimal solutions likely exist. Unfortunately, due to time constraints during the project, we were unable to fully explore these alternatives. As a result, we were limited to testing small datasets, with a maximum of 20 training samples and 10 test samples.

QSVM classification requires two quantum kernel matrices: one for training and one for testing. The training matrix is symmetric and has a size of $n \times n$, where n is the number of training samples. The test matrix is $m \times n$, where m is the number of

testing samples. For the dataset of 20 training and 10 testing samples, our one-job-per-kernel-entry approach results in 390 jobs on the quantum computer.

During the debugging phase, we conducted experiments to evaluate the time required to execute a single job. Although the individual jobs are relatively small regarding data volume and processing time, the nature of machine learning tasks requires a substantial number of jobs, particularly with our current implementation, where one job is required per kernel entry. Each job involves running a parameterized quantum circuit (based on the chosen feature map) with a specific sample (feature vector) as the parameter. We tested different numbers of shots and various quantum processors, finding that executing one job takes approximately 15 seconds of *quantum time*. Quantum time refers to the total duration a quantum system is committed to fulfilling a user’s request [12]. Therefore, the total time required to evaluate the small dataset with 20 training and 10 testing samples is approximately 97.5 minutes on the quantum computer. These limitations are further compounded by the constraints of the CTU license, which grants us access to only 400 minutes per month.

We experimented with the number of jobs submitted in a single session. Sessions allow all jobs to be executed consecutively, minimizing queue wait times. However, as the number of jobs and the quantum minutes used approach the limits imposed by our license, queue wait times can increase 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` system, leading us to explore alternative systems.

In our experimentation, we tested various systems and opted to submit all jobs within a single session to manage larger workloads. When selecting the least busy system available, we typically encountered queue times of only a few minutes. However, with the busiest systems (in our case, `ibm_torino`), wait times could extend to several hours, even for a relatively small number of jobs. While the quantum processing time required to execute the jobs was consistent across various systems, with differences of only a few seconds, these variations had a notable impact given our limited resources and the larger volume of jobs we needed to process.

Table 3 presents the results of our experiments. We used different IBM Quantum systems for each dataset size, including `ibm_torino`, `ibm_algiers`, `ibm_cairo`, and `ibm_kyoto`. The column labeled *job time* specifies the average execution time of each job on the respective system, measured in quantum seconds. Although we provide accuracy and F1 score metrics for completeness, it is important to note that due to the small dataset sizes, these metrics may not fully represent the performance of the QSVM algorithm. However, they offer insights into relative performance across different systems and dataset sizes. The table highlights the iterative nature of our experiments, beginning with smaller datasets and progressively scaling up. We started with 4 training and 2 test samples, gradually increasing to 8 training and 4 test samples, and eventually evaluating a larger dataset with 20 training and 10 test samples.

Table 3: Experiment Results: QSVM Classification on IBM Quantum Systems

Data (Train/Test)	IBM Quantum System	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

5 Conclusion and Future Work

We extended the previous work by focusing on the implementation and evaluation on real quantum computers, which brings its own challenges. We addressed and fixed the issues in the original implementation of classes for quantum kernel in Qiskit Machine Learning 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 has not yet been resolved at the time of finishing this work. Our fixes address critical flaws in the original implementation and pave the way for more efficient usage of quantum computing resources in malware detection.

Besides the local simulator, we also used IBM Quantum computers to compute the quantum kernel for QSVM classification. We tested how the IBM Quantum computers behave under the workload of many computation jobs.

In future work, we aim to optimize the transpilation process and the one-job-per-kernel entry approach to enable large-scale experiments on IBM Quantum computers. Further investigation into their topology would also be beneficial, as each system features a unique layout of qubits. We may reduce computation time by specifying the exact qubits used for computation.

From an algorithmic perspective, we plan to experiment with feature map design and combine different data mapping functions to enhance our approach. Furthermore, we would like to investigate various preprocessing techniques and their impact on the classification results.

Acknowledgements This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/211/OHK3/3T/18 funded by the MEYS of the Czech Republic.

References

1. Avast: AI and machine learning. <https://www.avast.com/technology/ai-and-machine-learning>, 2024.

2. Sheldon Axler. *Linear Algebra Done Right*. Springer International Publishing, Cham, 4th edition, 2024.
3. Grégoire Barrué and Tony Quertier. Quantum machine learning for malware classification. <https://doi.org/10.48550/arXiv.2305.09674>, 2023.
4. Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1), 2022.
5. Yaswitha Gujju, Atsushi Matsuo, and Rudy Raymond. Quantum machine learning on near-term quantum devices: Current state of supervised and unsupervised techniques for real-world applications. <https://doi.org/10.48550/arXiv.2307.00908>, 2023.
6. Vojtěch Havlíček, Antonio D. Córcoles, Kristan Temme, Aram W. Harrow, Abhinav Kandala, Jerry M. Chow, and Jay M. Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, 2019.
7. IBM: What is principal component analysis (PCA)? <https://www.ibm.com/topics/principal-component-analysis>, 2024.
8. IBM Quantum documentation: PauliFeatureMap. <https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.PauliFeatureMap>, 2023.
9. IBM Quantum documentation: ZFeatureMap. <https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.ZFeatureMap>, 2023.
10. IBM Quantum documentation: ZZFeatureMap. <https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.ZZFeatureMap>, 2023.
11. IBM Quantum documentation: Configure runtime compilation for Qiskit runtime. <https://docs.quantum.ibm.com/run/configure-runtime-compilation>, 2024.
12. IBM Quantum documentation: Estimate job run time. <https://docs.quantum.ibm.com/run/estimate-job-run-time>, 2024.
13. IBM Quantum documentation: Maximum execution time for Qiskit runtime workloads. <https://docs.quantum.ibm.com/run/max-execution-time>, 2024.
14. IBM Quantum platform. <https://quantum.ibm.com/>, 2024.
15. IBM Quantum platform: Update to Qiskit runtime primitives. <https://docs.quantum.ibm.com/announcements/product-updates/2024-02-14-qiskit-runtime-primitives-update#update-to-qiskit-runtime-primitives>, 2024.
16. Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.
17. Kaspersky: Machine learning in cybersecurity. <https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity>, 2024.
18. Eliška Krátká. Quantum computing methods for malware detection. <https://gitlab.fit.cvut.cz/kratkeli/quantum-malware-detection>, 2024.
19. Michael Lester. PE malware machine learning dataset. <https://practicalsecurityanalytics.com/pe-malware-machine-learning-dataset/>, 2021.
20. MalShare. <https://malshare.com/>, 2024.
21. L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: Visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, pages 1–7, New York, NY, USA, 2011. ACM.
22. Yuval Nativ, Lahad Ludar, and 5fingers. theZoo. <https://github.com/ytisf/theZoo>, 2021.
23. Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.
24. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

25. Anna Phan. Qiskit global summer school 2021: Introduction to quantum kernels and SVMs. <https://github.com/Qiskit/platypus/blob/main/notebooks/summer-school/2021/resources/lab-notebooks/lab-3.ipynb>, 2021.
26. John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.
27. Qiskit Algorithms: BaseStateFidelity. https://qiskit-community.github.io/qiskit-algorithms/stubs/qiskit_algorithms.state_fidelities.BaseStateFidelity.html#qiskit_algorithms.state_fidelities.BaseStateFidelity, 2024.
28. Qiskit Algorithms: ComputeUncompute. https://qiskit-community.github.io/qiskit-algorithms/stubs/qiskit_algorithms.state_fidelities.ComputeUncompute.html#qiskit_algorithms.state_fidelities.ComputeUncompute, 2024.
29. Qiskit Algorithms: ISA circuit support for latest runtime. <https://github.com/qiskit-community/qiskit-algorithms/issues/164>, 2024.
30. Qiskit IBM runtime: Sampler fails to run FidelityKernel even if circuits are transpiled. <https://github.com/Qiskit/qiskit-ibm-runtime/issues/1519>, 2024.
31. Qiskit Machine Learning. <https://qiskit-community.github.io/qiskit-machine-learning/>, 2024.
32. Qiskit Machine Learning: BaseKernel. https://qiskit-community.github.io/qiskit-machine-learning/stubs/qiskit_machine_learning.kernels.BaseKernel.html#qiskit_machine_learning.kernels.BaseKernel, 2024.
33. Qiskit Machine Learning: FidelityQuantumKernel. https://qiskit-community.github.io/qiskit-machine-learning/stubs/qiskit_machine_learning.kernels.FidelityQuantumKernel.html, 2024.
34. B. Schölkopf, S. Mika, Burges. C.J.C., P. Knirsch, K.R. Muller, G. Ratsch, and A.J. Smola. Input space versus feature space in kernel-based methods. *IEEE Transactions on Neural Networks*, 10(5):1000–1017, 1999.
35. scikit-learn: accuracy_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html, 2024.
36. scikit-learn: f1_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html, 2024.
37. scikit-learn: precision_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html, 2024.
38. scikit-learn: recall_score. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html, 2024.
39. R.K. Shahzad. *Automated Malware Detection and Classification Using Supervised Learning*. Blekinge Institute of Technology Doctoral Dissertation Series. Blekinge Tekniska Högskola, 2024.
40. Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.
41. VirusShare. <https://virusshare.com/>, 2024.