University of the Philippines Manila
College of Arts and Sciences
Department of Physical Sciences and Mathematics

Privacy-Preserving Viral Strain Classification Through a Client-Server Application Using An Open-Source Fully Homomorphic Encryption Library

A special problem in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science

Submitted by:

Johann Benjamin P. Vivas

June 2023

Permission is given for the following people to have access to this SP:

Available to the general public	Yes
Available only after consultation with author/SP adviser	No
Available only to those bound by confidentiality agreement	No

ACCEPTANCE SHEET

The Special Problem entitled "Privacy-Preserving Viral Strain Classification Through a Client-Server Application Using An Open-Source Fully Homomorphic Encryption Library" prepared and submitted by Johann Benjamin P. Vivas in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science has been examined and is recommended for acceptance.

Richard Bryann L. Chua, M.Sc.
Adviser

EXAMINERS:

	Approved	**Disapproved**
1. Avegail D. Carpio, M.Sc.	_____	_____
2. Perlita E. Gasmen, M.Sc. (*cand.*)	_____	_____
3. Ma. Sheila A. Magboo, Ph.D. (*cand.*)	_____	_____
4. Vincent Peter C. Magboo, M.D.	_____	_____
5. Marbert John C. Marasigan, M.Sc. (*cand.*)	_____	_____
6. Geoffrey A. Solano, Ph.D.	_____	_____

Accepted and approved as partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science.

Vio Jianu C. Mojica, M.Sc.
Unit Head
Mathematical and Computing Sciences Unit
Department of Physical Sciences and Mathematics

Marie Josephine M. De Luna, Ph.D.
Chair
Department of Physical Sciences and Mathematics

Maria Constancia O. Carrillo, Ph.D.
Dean
College of Arts and Sciences

Abstract

ML techniques and outsourcing are being increasingly used by researchers in their efforts to look into and better understand SARS-CoV-2 and combat the spread of the virus. However, this brings about privacy issues that surround the sharing of, and training of ML models on, SARS-CoV-2 genomic sequences and contextual data, potentially leading to the reidentification of the owners of such genomic data. Thus, there is a need to develop methods of protecting patients' privacy, all while allowing researchers and medical professionals to continue the use of ML techniques and outsourcing to make better informed medical decisions and take more effective actions against the spread of the virus. To that end, this paper proposes a fully homomorphic encryption-based viral classification framework and logistic regression model based on Concrete-ML, a fully open-source FHE ML library.

Keywords: Fully Homomorphic Encryption, Viral Strain Classification, Machine Learning, Security, Privacy, Genomic Data

Contents

List of Figures

List of Tables

I. Introduction

A. Background of the Study

In recent years, the issue of data privacy has become one of the most talked-about topics in multiple domains. Among these are the legal, technological, medical, and financial domains, where it is often mentioned how a dangerous situation may arise from leaked financial, medical, legal, or personal data, and how practicing data privacy is paramount to avoid such situations. [2, 3] Even with efforts including acts such as the General Data Protection Regulation in the European Union and European Economic Area [4], and the Data Privacy Act of 2012 in the Philippines [5], there are many issues and concerns that remain unresolved.

These pressing issues include the privacy concerns with sharing viral genomic sequencing data, as discussed by Song et al [6]. In their work, they brought attention to the nuances in this matter, acknowledging that the sharing of viral genomic sequencing data, in light of recent developments in the COVID-19 pandemic, is tantamount to properly informing regional, national, and international public health responses. It was also important to the development and improvement of clinical therapies and vaccines and allowing researchers, scientists, and experts to better understand the SARS-CoV-2 virus. They also maintain that while the sharing of SARS-CoV-2 viral genomic sequences alone is extremely unlikely to lead to the re-identification of the data' owners due to the SARS-CoV-2 virus's short mutation rate and even shorter serial interval, some minimal contextual data is required to properly assess and process the data itself. These contextual data fields can take the form of the gender, age, province/territory, or a combination of these factors, or can also take other forms, depending on what would bring the most value to the data while minimizing privacy risks.

Further, Song et al. mention that, in most situations, the inclusion of such con-

textual data fields in association with the sequencing data is not contradictory to privacy laws. The authors do, however, note that extra care should be taken to evaluate whether the inclusion of these contextual data fields with viral sequence data would disproportionately increase the risk of re-identification, and thus the risk to privacy on the whole. They explained that this can happen due to the sensitivity of the data in relation to factors such as the population pool and confirmed cases in the specific province. Following this, some particular examples in which certain contextual data fields disproportionately increased the risk of re-identification were brought up by the authors, beginning by mentioning the Gordon v. Canada 2008 federal court case, in which the data field of "province" or "territory" can create a disproportionate risk of re-identification in provinces and territories with a smaller population [7, 8]. Song et al. next discussed the potential re-identification risks of providing data fields similar to "collection agency", which they noted is not uncommonly the name of a local hospital that could provide more detailed geographical information, increasing re-identification risk.

Next, the gender and age data fields were mentioned and described as potentially disproportionately increasing the risk of re-identification when paired with other contextual data, as also discussed by [9, 10, 11]. The amount that the re-identification risk increases varies widely depending on the age bracket in question, as the very elderly or very young individuals may in some cases make up a significantly smaller fraction of a particular population, and should be considered more risk-prone as a result. In such cases, special mitigating measures should be taken to reduce the risks of re-identification, and the authors also recommend the periodic monitoring of re-identification risk to account for the increased efficiency of diagnostic methods and other relevant developments that could potentially increase privacy risks.

Adding to the points already raised are the recent events concerning SARS-CoV-2 sequences and metagenomics data from China that was uploaded to the GISAID

database, which were associated with samples dated January 2020 from the Huanan Seafood Wholesale Market in Wuhan, China [12]. Various sources including the World Health Organization and Ars Technica report that analyses of these data suggest that apart from SARS-CoV-2 sequences, some samples also contained human DNA, as well as mitochondrial DNA of several animal species, including some that are known to be susceptible to SARS-CoV-2 [13, 14, 15]. This development raises the concern that SARS-CoV-2 sequences hosted on public databases such as GISAID may contain trace amounts of human DNA, which is corroborated by the work of Lehrer & Rheinstein [16] and Li et al. [17]. These studies discuss that many SARS-CoV-2 human samples do contain trace amounts of human DNA and RNA, which has the potential to increase the risk of reidentification, especially when considered in tandem with the privacy risks that even minimal contextual data fields associated with the sequences pose and further warrants research into ways to reduce the likelihood of reidentification associated with sharing SARS-CoV-2 sequences and contextual data.

Further, the events previously discussed have also affected GISAID's credibility, and have caused the repository to come under scrutiny in light of other shortcomings with regard to data sharing limitations and the manner in which GISAID handles disputes of credit and sanctions scientists who have allegedly violated the terms and conditions of the repository [18]. This has led to a growth in sentiment for fully open-access data sharing, as evidenced by an open letter published by a group of scientists urging other researchers to publish their datasets on open-access databases [19]. However, publishing datasets, particularly those as timely as SARS-COV-2 sequence datasets, on open-access databases poses a risk of the data being 'scooped', a practice in other scientists publish studies using the dataset first instead of the original authors and creators of the dataset. This prevents the timely sharing of potentially crucial data.

From the works of the aforementioned authors and the developments that have

transpired in recent years, it can be clearly seen that there is a need for low-risk, privacy-preserving viral genomic data that will allow researchers and experts to improve their understanding of COVID-19, as well as other viral diseases, as well as share their findings and genomic data with other researchers to allow for faster and more successful strides towards better medical responses and healthcare. One potential solution to this is the use of Homomorphic Encryption, which, as defined by various sources, is a relatively new form of encryption technology that allows operations such as addition and multiplication to be performed on encrypted data without the need for decryption beforehand. Virtually any algorithm can be computed and performed on encrypted numbers with the two available operations, and this can even include the analysis and classification of encrypted biomedical data, such as DNA sequences [20, 21, 22]

B. Statement of the Problem

The sharing of viral sequence data along with some of its contextual data is vital to the analysis and surveillance of the SARS-CoV-2 pandemic through various processes such as viral sequence classification. That being said, while Song et al [6] explain that viral sequence data alone does not substantially increase the risk of reidentification of an individual due SARS-CoV-2's mutation rate and serial interval, these characteristics can lead to the identification of groups of individuals. To elaborate, SARS-CoV-2's serial interval, the interval between the onset of symptoms in an infector (individual that transmits the virus) and the infectee (individual infected by the virus from the infector), is shorter than its rate of mutation. This means that multiple infector-infectee pairs can share the same viral sequence and that single individuals cannot be identified by any particular strain. However, entire groups of individuals can be identified by these same characteristics. Given this, the identification of groups of individuals can still raise ethical concerns as it could possibly lead to ostracization

or discrimination due to the social stigma that being diagnosed with SARS-CoV-2 carries.

What this means is that viral strain classification, along with other methods to study, classify and protect against SARS-CoV-2 strains based on patients' viral sequence data is potentially at odds with legal and ethical boundaries regarding the patients' rights to data privacy and protection, and thus poses a risk of endangering the data's owner. In the context of the privacy concerns regarding the sharing of viral sequencing data and the use of viral strain classification, the use of Homomorphic Encryption instead addresses the problem from the angle of improving the privacy-preserving properties of our methods as opposed to adjusting the data to meet our needs. This was explored in one of the challenges put forward in the 2021 IDASH Privacy & Security Workshop [23]. The workshop had brought attention to the issue and contributed to the development of various models that were able to successfully preserve the privacy of the patients who owned the viral sequence data by allowing viral strain classification to be performed on encrypted data without prior decryption, thereby significantly reducing the risk of re-identification.

While the results of the iDASH competition were reassuring in that viral strain classification was shown to be a practical task, as the performance of most of the models in performing privacy-preserving viral strain classification was impressive, there were still some optimization issues that led to large variability in the time cost of the classification of the sequences. It also brings attention to the potential of FHE in viral strain classification, while also encouraging many others to explore the use of other publicly available methods to implement solutions to the privacy concerns that plague the sharing and classification of viral sequencing data.

Thus, at the core of this dilemma lies the following questions: "How well does an open-source FHE library lend itself towards the implementation of practical privacy-preserving viral strain classification?", and "What is the most appropriate machine

learning method for viral strain classification that can be implemented in FHE?"

This study aims to answer these questions through the discussion and use of existing FHE libraries with features that lend themselves to machine learning implementations.

C. Objectives of the Study

The objective of this study was to develop an application that allows researchers and service providers to perform the homomorphic computations required to conduct viral strain classification on data that is provided by clients on the client version of the application, effectively ensuring data is encrypted during the entire servicing process. The study resulted in the creation of an application with a client-server model. Here, the client represents a hospital, clinic, or other medical institution that would require the use of ML services to provide better diagnoses and treatment for viral diseases and would need to outsource such tasks to ML service providers. Likewise, the server represents an ML service provider.

The model developed sported the following functionalities:

Allow the hospital (client) to:

1. Encrypt and upload their preprocessed viral strain data
2. Send their encrypted data to the ML service provider for classification
3. Receive and decrypt the results of the classification

Provide the ML service provider (server) with the following capabilities:

1. Receive the encrypted viral strain data from the hospital
2. Perform homomorphic viral strain classification via Logistic Regression
3. Send the results of the homomorphic classification to the hospital

D. Significance of the Project

The exploration and implementation of a privacy-preserving viral strain classification application are of great benefit to various medical fields, as through it, clients, such as medical professionals and researchers, are better equipped to study viral strain sequences without the risk of violating privacy standards, potentially allowing them to develop and/or roll out appropriate treatment and medical initiatives more effectively.

Patients would also benefit greatly from the implementation of this application as well. The use of FHE to encrypt their data significantly reduces the chances of their viral sequence data being used to re-identify them.

E. Scope and Limitations

For this study, the author worked primarily with the existing FHE library Concrete-ML. Concrete-ML was chosen for the implementation of the application in this study due to its propensity for being used in Machine Learning or Deep Learning models and applications, by virtue of its various built-in ML models that can be easily compiled from its plaintext version into a corresponding FHE version of the model. These models are also compatible with Scikit-learn modules, processes, and workflows. Concrete-ML is also implemented in Python, allowing for easier programming due to the high-level nature of the language. One major advantage that Concrete-ML boasts is that it also supports tree-based classification models in FHE, which meant that the implementation of such was significantly easier than first envisioned as there are no known general conditional statements in FHE at the time of writing, which is important in the creating of tree-based classifiers from scratch.

As Concrete-ML compiles their plaintext models into equivalent FHE circuits through Concrete's FHE compiler to produce an equivalent FHE circuit, which is performed under the hood. Additionally, Concrete-ML's FHE compiler makes use of the TFHE scheme only, and adjustments, such as the quantization of non-integer

inputs, are required. Concrete-ML's FHE engine also does not support data batching, so loops are required for the implementation of ML through Concrete-ML.

The application was built primarily for the Windows operating system due to the ease of use and large user base of the system. However, due to operating system limitations for tools considered essential to the study such as Dashing, the Windows Subsystem for Linux was used for functions and tools that did not support Windows, such as Dashing.

This setup allows future studies to reproduce and improve upon this study with more ease, allowing future researchers to focus their efforts on more important aspects.

F. Assumptions

1. The client in the model will provide valid viral strain sequences in FASTA format, which is ideally preprocessed using techniques such as Dashing [24], as performed by the A*FHE team, who were behind the CoVnita framework [25].

II. Review of Related Literature

A. Machine Learning and its use in Viral Strain Classification

In recent years, Machine Learning has often been used in the diagnosis of viral diseases with the long-term goal of preventing their spread. One such instance of the use of ML can be seen in Remita et al's work [26], where in the course of the study, the group developed a virus classification platform, dubbed CASTOR, which simulates, in silico, the restriction digestion of genomic material by different enzymes into fragments. Remita et al. explored and incorporated various machine learning algorithms, each of which fell under one of three types: symbolic methods, characterized by decision trees and random forests, statistical methods, such as the naive Bayes classifier and support vector machine, and k-nearest neighbors, and ensemble methods, under which Adaboost and Bagging fell. The performance of the platform was benchmarked for the classification of distinct datasets of human papillomaviruses (HPV), hepatitis B viruses (HBV), and human immunodeficiency viruses type 1 (HIV-1). The classification results ended with the authors noting that, in general, SVM had performed better in four of the five datasets utilized with the mean of weighted $F - measures$ higher than 0.906, ranking first in HPV Alpha species, HBV genotypes and HIV-1 subtypes classifications, and fourth in HPV genera classification. SVM is followed by Random Forest, Naive Bayes Classifier, and K-Nearest Neighbors in terms of performance, though Remita et al. note that the Random Forest and Naive Bayes classifiers are affected by large variances.

Kim et al. [27] also achieved the successful classification of porcine reproductive and respiratory syndrome virus (PRRSV) sublineages, while also detecting key amino acid positions. The authors achieved this by implementing four ML approaches based on the amino acid scores of the ORF5 gene. The approaches implemented were ran-

dom forest, support vector machine, k-nearest neighbors, and multilayer perceptron. Kim et al. also performed experiments to see how the number of amino acid sequences affected the performance and time consumption of the ML algorithms. The results of their study revealed that all four ML approaches tested for the classification of PRRSV sublineages had been able to perform accurate classification of the sublineages (all algorithms having an AUC of 0.98 and above) when using at least 2 amino acid positions: the two amino acid positions with the highest RF scores. When only one amino acid sequence (with the highest RF importance score) was used, the accuracy had dropped down to around 80% for all algorithms. The results also showed that the accuracy for all four ML algorithms in classifying the PPRSV sublineages were identical.

Other studies, such as those by Lopez et al. [28] and Câmara et al. [29], focused on the classification of Sars-CoV-2 viral strains, with both studies choosing to utilize deep learning in the form of convolutional neural networks, or CNNs, to facilitate their operations. To elaborate on this, in their study, Lopez et al. trained a convolutional neural network on 553 sequences from the National Genomics Data Center (NGDC) repository, resulting in a classifier capable of separating the genome of different virus strains from the Coronavirus family with 98.73% accuracy. Câmara et al. worked with 1557 instances of SARS-CoV-2 from the National Center for Biotechnology Information (NCBI), and 14,684 different viruses from the Virus-Host DB. Further, as their CNN was highly customizable with several changeable parameters, the tests were performed on forty-eight different architectures with the best of these having an accuracy of $91.94 \pm 2.62\%$ in classifying viruses into their realms correctly.

B. Privacy concerns with Machine Learning in the Medical Domain

It cannot be understated how much of an impact these studies, along with many others, have collectively had on the medical and computer science domains. These studies have allowed medical professionals and health informatics experts to identify and either prevent the spread or treat those who have been diagnosed with viral diseases. However, these methods have also had concerns of various natures raised against them, which Char et al. [30, 31] illustrate in their work, discussing concerns such as those that ML algorithms may mirror human biases in decision-making, as well as concerns about the intent behind the design of machine-learning systems, as algorithms can be designed to perform in unethical ways. Despite this, they acknowledge that the incorporation of machine learning into clinical medicine holds much promise as ML is capable of improving the overall quality and speed of healthcare delivery and can serve as a foundation on which several initiatives and projects headed by public and private companies can be built. Lastly, Char et al. reiterate that the consideration of ethical issues and problems inherent in the implementation and use of machine learning in healthcare systems is also warranted [30, 31].

The main issue being addressed in this study, however, is the issue of data privacy, integrity, and overall protection from potential bad actors. In recent years, as machine learning methods and techniques have become more complex and resource-intensive, researchers and companies have turned to the use of cloud computing to achieve the results they need in a cheaper and more cost-effective manner. With that, when a machine learning model is trained or classified in a cloud environment, the server itself obtains data from the user side. Given this, the privacy of the data then depends on the service provider, and thus it could be very easy for malicious acquisition and utilization of data to occur [32, 33]. As can be seen from the issues previously discussed by [6, 31, 30, 32, 33], there is a need for methods to address the concerns

of privacy and data security.

C. Fully Homomorphic Encryption

While there have been several methods of achieving data privacy and anonymization in the past, as evidenced by [34], one such approach has been touted by several as the "holy grail" of cryptography: Homomorphic Encryption, and in particular, Fully Homomorphic Encryption. Homomorphic encryption (HE), as described by numerous works [33, 35, 36, 32], can be described as a relatively new form of technology that allows data to be encrypted and for complex operations to be performed on it without having to decrypt that data in the process. Several different types of HE have been developed over time, including, but not limited to, partially homomorphic encryption (PHE) which originally only had support for either additional homomorphism or multiplicative homomorphism, meaning that only one of the two operations between encrypted data could be achieved to the intended effect, and fully homomorphic encryption (FHE), which fully realized the concept by allowing both addition and multiplication operations to be performed on encrypted datasets, as any arbitrary function could theoretically be implemented with these two operations.

D. FHE libraries and the Concrete-ML FHE ML library

There have been several advances in HE since its original conception by Rivest et al. [37]. All of these advances have culminated in today's FHE libraries, all implemented in various languages, with some implemented in relatively low-level languages like C++ and others in higher-level languages like Python. Some examples of these libraries are Microsoft SEAL, OpenFHE, TenSEAL, Concrete-ML, HElayers, and Pyfhel [38, 39, 35, 40, 41, 36].

While some of these libraries have seen use in different applications and studies which involve machine learning with data that is considered sensitive, such as Mi-

crosoft SEAL and HElayers, which were utilized by two of the top three contenders of the iDASH 2021 competition [42, 33, 25], Concrete-ML stands out from other FHE libraries for its specialization in incorporating FHE into its built-in ML models that are based on, and are compatible with, Scikit-learn processes and workflows. It also has the edge over other libraries in that, aside from already having pre-built ML algorithms, there are built-in algorithms for tree-based models, which are difficult to implement in FHE due to lacking general homomorphic conditional statements for comparison purposes. Concrete-ML was also implemented in Python, which lends to the user-friendliness of the library due to high-level abstractions and interfaces. Additionally, much of the parameter setting in Concrete-ML is handled by its native FHE compiler. All these traits make Concrete-ML a very accessible and relatively easy-to-implement-and-improve FHE ML library and are the reason for the author's choice to utilize this particular library for this study.

E. Applications of FHE

Applications that implement FHE in some of their features have also been implemented and even rolled out. These include works that involve privacy-preserving genome-wide association studies [22, 43], genotype imputation [44, 27, 44], and federated analytics [45]. One other noteworthy implementation of FHE [46] had introduced the Password Monitor feature to Microsoft Edge, which allowed Microsoft to monitor and alert users if their password had been found in a third-party breach. The password monitor features homomorphic encryption, allowing it to avoid learning the passwords of the user while monitoring its status.

It eventually came to be implemented in different domains that deal with very sensitive data. Machine Learning applications that implement FHE methods have been created and shared with the community, setting a precedent for the use of privacy-preserving encryption methods that allow for the processing of sensitive data

without putting the owners of the data at risk [26, 32]. In direct relation is the creation and publication of various privacy-preserving viral strain classification applications as part of the iDASH 2021 competition [42, 33]. The results of the competition serve as a great example of the use of FHE to improve the state of Machine Learning and its use in the delivery of healthcare services, all while preserving the privacy of patient data. At the time of writing, there are two solutions from the competition that have received publications in websites and journals, or are currently being peer-reviewed: IBM's HElayers-based Privacy-Preserving Viral Strain Classification based on k-mer signatures and FHE [47, 33, 48, 49], and I2R's A*FHE-2 Team's End-to-end Privacy-preserving SARS-CoV-2 Classification Framework entitled CoVnita [25].

Akavia et al.'s work focused on the implementation of a client-server protocol through the Microsoft SEAL and HElayers libraries, wherein the model owner would provide viral strain classification services on encrypted data, which is uploaded by the DNA-Sequence owner. To elaborate, the process would begin with the client computing their DNA-Sequence's k-mer set. Next, the client would encrypt this set through FHE using any FHE scheme that supports the encryption of complex numbers, and send it to the server. Lastly, the actual classification is performed based on the k-mer sets derived from the DNA sequences, and the Jaccard similarity between a particular strain in the data and various sequences representative of each of the different Sars-CoV-2 viral strains. Once Jaccard similarity scores were successfully computed for the k-mer sets and normalized, the strain with the highest similarity score is chosen to be the classification of that particular strain, and classification would continue until the process was completed. On the other hand, A*FHE-2's solution was CoVnita, an end-to-end privacy-preserving framework for SARS-COV-2 classification. They implemented this framework, resulting in a viral strain classification model that used a logistic regression algorithm and was built on a federated learning approach, wherein various data providers train their own local models using their own data, after

which a joint global model that does not require data providers to share their genomic samples, is trained. The model's FHE capabilities were powered by Microsoft SEAL. A*FHE also noted that they had chosen logistic regression as they felt that it is an appropriate machine learning model for the purpose of minimizing data exposure, as, unlike machine learning models like k nearest neighbors, which will expose all the individual data values, logistic regression restricts the sharing of information and also prevents the exposure of individual data values. It should also be noted that both publicly documented solutions had placed in the top three for the iDASH competition, and both used open-source libraries which were implemented in an ML context. Thus, this study aims to investigate whether practical privacy-preserving classification can be achieved with other open-source libraries, particularly those more specialized for applications in ML such as Concrete-ML.

III. Theoretical Framework

As Creeger and Cornami Inc. [50] put it, Homomorphic Encryption, in particular, Fully Homomorphic Encryption, aka FHE, are new and burgeoning technologies that have the potential to provide quantum-secure computing over encrypted data. FHE guarantees that the plaintext data and the results derived from its analysis, decomposition, computation, or other forms of processing, are kept secure from breach even with compromised infrastructures or models. Most of the FHE schemes today are based on lattice mathematics [51], and are considered safe from breaches via quantum computing, and to that end, are considered post-quantum cryptography.

A. Developments of FHE

The original concept of Homomorphic Encryption began to form in 1978 when Rivest et al. [52] recognized it and addressed homomorphic behavior in their study. They showed that when two RSA (Rivest-Shamir-Adleman) encrypted numbers are added together, their result, when decrypted, is equivalent to the result of the same operation if done on the unencrypted numbers [37]. This is called privacy homomorphism and is present in certain encryption schemes. By leveraging these properties, the separation of data processing and data access can be achieved, encrypted queries could be made possible, and data involved in such operations were never decrypted, at rest or during its life cycle. More than 30 years later in 2009, Craig Gentry [53] proposed the very first FHE scheme. Gentry defined the algorithms of the scheme as a series of logic gates, and unrestricted computation was possible with the scheme, and the results of computation with encrypted numbers were encrypted in much the same way. While Gentry's FHE scheme was extremely slow, taking half an hour to finish a single logic gate on available standard x86 hardware at the time as evidenced by the results of its implementation [54], it helped kickstart the development of other FHE schemes,

which can be divided into four distinct generations:

B. FHE Across the Generations

The first generation of FHE can be characterized by the release of Gentry's FHE scheme. Gentry's publication paved the way for the further development of fundamental concepts in FHE, such as the introduction of noise to a public key by DGHV [55]. DGHV developed an encryption scheme that allowed homomorphic operations over integers, which they used to replace the SHE portion of Gentry's originally conceived scheme. The second generation of developments in FHE followed, defined by the development of the BFV (Brakerski/Fan-Vercauteren) and BGV (Brakerski-Gentry-Vaikuntanathan) schemes [50]. These schemes introduced the LWE (Learning With Error) and RLWE (Ring Learning With Error) security models, as well as leveling schemes that allowed for the execution of a logic-gate circuit of a certain set depth before requiring a bootstrap to manage the noise level. Meanwhile, the most noteworthy events that formed the third generation of FHE developments included the introduction of the GSW (Gentry-Sahai-Waters) homomorphic encryption scheme, which avoided relinearization, which was considered computationally expensive, when performing homomorphic multiplication. Due to this, the scheme exhibited slower noise growth. The development of more efficient ring variants with FHEW (Ducas-Micciancio — "Fastest Homomorphic Encryption in the West"), as well as the simplification and increased optimization of bootstrapping, were also observed in this generation. Lastly, the fourth generation brought the CKKS (Cheon-Kim-Kim-Song) scheme, which introduced efficient rounding operations for encrypted values, controlling noise rate increases in HE multiplication and reducing the number of bootstraps required in a logic circuit. The generation also brought the concept of PBS (programmable bootstrapping) to TFHE (torus fully homomorphic encryption) [56], reducing the number of bootstraps required in a logic circuit.

C. The Concrete-ML FHE ML Library

We utilized the open-source FHE ML library Concrete-ML for its specialization in machine learning and deep learning, and execution of ML/DL algorithms in FHE, as well as its implementation in Python, allowing easier programming thanks to its high-level nature and abstractions. It is also compatible with Scikit-learn modules and workflows, and has built-in support for implementation in client-server architectures. Concrete-ML also features supports a variety of regression and classification models that can be compiled to FHE equivalents. These supported models are categorized into linear models, tree-based models, and neural networks for deep learning.

For linear models, Concrete-ML supports linear regression, logistic regression, linear SVC, linear SVR, Poisson regressor, Tweedie regressor, gamma regressor, lasso, ridge, and elastic net [57]. For tree-based models, it supports decision tree classifier, decision tree regressor, random forest classifier, and random forest regressor, which are scikit-learn compatible, as well as the XG classifier and XG regressor, from XG Boost [58]. Lastly, Concrete-ML supports neural net classifiers and neural net regressors for its neural network repertoire [59]. All in all, these models provide a better interface for exploring FHE ML methods and finding the best fit for viral strain classification [40].

D. Concrete-ML Workflow

A Concrete-ML model's lifecycle can be divided into the model's development and its deployment [60]. Under the model development phase, the following actions are taken:

1. The **training** of the model, where a model is trained on plaintext/unencrypted training data. Since ConcreteML only supports FHE inference at the time of writing, this particular step of model development can only be performed using

plaintext data.

2. The **quantization** of the model. The process of quantization converts inputs, model weights, and all intermediate values of the inference computation to integers. The point at which quantization is performed depends on the model type. It can be performed either during training (also dubbed Quantization-Aware Training) for Neural Networks or after training (Post-Training Quantization) for Linear Models. For tree-based models, the data instead of the model is quantized into its integer equivalent. The main parameter that affects the quantization process is the number of bits, n_bits.

   In the case of linear models, n_bits can use a single integer value dependent on the number of attributes. For a higher number of attributes, it is recommended to use a lower value. On the other hand, a dictionary of integer values can also be passed to n_bits to allow it to use different parameters. For tree-based models, the maximum accumulator bit-width is determined to be $n\_bits + 1$ bits. As the Concrete FHE framework is only limited to 8-bit integers, n_bits must be less than 8. Further information on the process can be accessed via the Zama AI documentation on Quantization [61].

3. The **compilation** of the model. This process is handled in its entirety by Concrete-Numpy [62]. Concrete-ML hides away most of the complexity of this step by allowing users to compile the model by simply using the `compile()` function [63].

4. Performing **FHE inference** using the compiled FHE Concrete model [60, 61, 62].

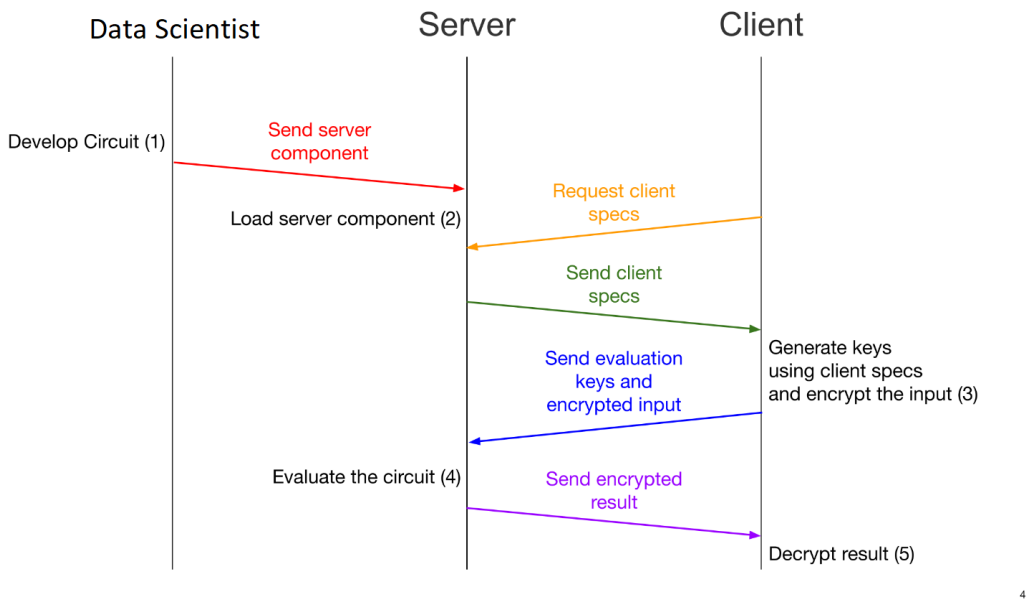


Figure 1: Summary of the overall communications protocol for deploying machine learning services [1].

Next, under the model deployment phase, Concrete-ML provides built-in functionalities for deploying the compiled FHE models in a client/server setting. The deployment workflow and model serving pattern are described as follows: For deployment, when the training and compilation of the model to its FHE equivalent is performed, three files are created when saving the model. These are:

1. **client.zip**, which contains the secure cryptographic parameters (`client.specs.json`) needed for the client to generate private and evaluation keys. It also contains **serialized_processing.json**, which contains required metadata about pre and post-processing, such as quantization parameters to quantize the input and dequantize the output, and a copy of `versions.json`.

2. **server.zip**, which contains the compiled model. This file is used to run the model on a server.

3. **versions.json**, which contains information about the version of Concrete-ML

used to compile the model. This file is important since compiled Concrete-ML models do not work with older versions of Concrete-ML.

Next, ConcreteML provides users with the concrete.ml.deployment.fhe_client_server module, which contains several APIs for FHE deployment, to make this process possible. Users make use of various classes included in this module for the deployment process. The FHEModelDev class is used for saving and exporting the files needed for the client-server system by first instantiating the class and using the `save()` method. The process of loading and running the compiled FHE circuit is performed by using the `run()` methods of the FHEModelServer class. Lastly, the FHEModelClient class is used for the encryption and decryption of the client's data. Once the client is created and the model is loaded, the private and evaluation keys are generated via the `generate_private_and_evaluation_keys()` method. After that, the serialization evaluation keys are retrieved and stored in a variable using the `get_serialized_evaluation_keys()` method. The evaluation key is then sent to the server, therefore ensuring that all requirements for client-server interaction are met. With this, the user/client can then begin preparing their input data for inference by quantizing, encrypting, and serializing the data using the `quantize_encrypt_serialize()` method. After FHE inference has been completed and the results have been sent back to the client, they can view the results of the inference by deserializing, decrypting, and dequantizing the data with the `deserialize_decrypt_dequantize()` method. [62, 64, 65, 66].

E. Viral Strain Classification Workflow

Similar to the workflow outlined by Sim et al. and Akavia et al. [25, 49], the general workflow for performing viral strain classification can be divided into the following steps, adjusted to integrate with the Concrete-ML workflow (Section D.):

1. The **training and compilation** of the classification model. As discussed in subsection D., this step of the workflow is to be performed on plaintext data. After this, the compilation and saving of the model follow, resulting in the FHE equivalent of the model capable of performing Homomorphic Inference.

2. The **dissemination of prerequisite files to the client for key generation, encryption, and decryption**. The client requests for the prerequisite files which it uses to generate the private and evaluation keys. Additional required files will also be requested by the client from the server. These include the Dashing binary releases and accompanying shell scripts for CSV file formatting (discussed in the next subsection), `features_and_classes.txt`, which hosts the results of the feature selection during training that allows the client to drop non-selected columns, and the original class labels for translating the classifier's outputs back into its text labels.

3. **Preprocessing, quantization, encoding, and encryption** of the input data. The client preprocesses the data using Dashing, which is discussed in subsection F.. After preprocessing, the data is then quantized, encrypted, and serialized using Concrete-ML's built-in API.

4. Lastly, **Homomorphic Inference** on the prepared data using the compiled model is performed. The results of the inference are then sent back to the client, who deserializes, decrypts, and dequantizes the data using the Concrete-ML API to view the results.

F. The Dashing Preprocessing Tool

Dashing is a software tool for estimating similarities of genomes or sequencing datasets [24]. The similarity estimation is performed by splitting a provided genome sequence into k-mers and converting each k-mer into a 64-bit hash. Dashing also uses the

HyperLogLog sketch, which is an approximate counting method in $O(\log_2 \log_2(n))$ space that estimates a count by incrementing counters with exponentially decaying probability [24, 67], alongside other cardinality estimation methods (which are specialized for set unions and intersections) to estimate the cardinality of the hashed genomes. The tool can summarize genomes more rapidly than previous MinHash-based methods while providing greater accuracy across a wide range of input sizes and sketch sizes [24]. Sim et al. [25] utilized this tool to allow for a layer of abstraction from the raw sequence data, thereby reducing the dimensionality of the data and allowing easier processing of it. The tool is accessible via the following GitHub link: https://github.com/dnbaker/dashing and can be downloaded in the form of a binary release. These binary releases work only on specific instruction sets, and are the following:

- `dashing_s128`, which works on systems supporting the 128-bit SSE2 SIMD instruction set
- `dashing_s256`, which works on systems that support the AVX2 256-bit SIMD instruction set
- `dashing_s512`, which runs on systems that feature the AVX512BW 512-bit SIMD instruction set

If a binary release does not work on a given system, then a release with a lower number should be used.

G. Logistic Regression

Logistic Regression is a type of statistical approach that is used quite often in predictive analysis. It is also used extensively when performing binary or multi-class classification and is also relatively simple to implement. [68, 69] Chang [70] noted

that logistic regression is relatively fast when compared to other supervised classification techniques such as SVM or ensemble methods. While logistic regression suffers to some degree in terms of accuracy as a result of its speed, Pradhan et al. and Zeller et al. note that logistic regression performs well when used with linearly separable classes and classification, potentially mitigating the model's lower accuracy [69, 71]. Zeller et al. used logistic regression in viral strain classification, since the genetic patterns in sequences, which come about as a result of genetic divergence over time, are linearly separable across aligned amino acid positions. This quality lends itself well to the use of logistic regression as well as other supervised machine learning methods, such as random forest classification. Lastly, Sim et al. [25] also used logistic regression from a privacy perspective in their proposed framework for privacy-preserving viral strain classification, noting that the use of logistic regression in the classification of viral strains is more appropriate as the classification approach does not expose the individual data values used to train the model in contrast to models such as k-Nearest Neighbors that do. The model's ease of implementation, speed of classification, and excellent performance when used with linearly separable classes make it the ideal algorithm for use in viral strain classification.

IV. Design and Implementation

A. Threat Model

Our system follows a semi-honest (honest-but-curious) threat model. That is, we assume that our server (an ML service provider) will stay honest and will adhere to the system's workflow, but is curious about the client's private information. We also assume that all communication channels between the two parties (client and server) are secure. Given this, if the security of the encryption scheme is guaranteed, then there will be no leakage of the client's private data to any potentially malicious parties, including the server. We also assume that the client (the owner of the secret key) does not collude with the server. Our security goals are as follows:

1. The server should not obtain or receive any information on the client's encrypted inputs.

2. The server should not find out any information regarding the encrypted prediction results.

B. Dataset

We used data from the publicly available repository Global Initiative on Sharing Avian Influenza Data (GISAID) [72, 73, 74]. We initially used the data of Sim et al. [25] (Episet ID `EPI_SET_220924cw`, accessible at https://doi.org/10.55876/gis8.220924cw). The dataset of nearly 9000 sequences was comprised primarily of four strains of interest. The strains are the B.1.617.2 (Delta), C.37 (Lambda), B.1.621 (Mu), and B.1.1.529 (Omicron) lineages, and were made the main focus of the study. This resulted in a smaller, unbalanced dataset of 6805 samples. To address this imbalance, we performed an upsampling of the minority (the B.1.1.529 and the B.1.617.2 classes) through the acquisition of a total of 2100 samples from GISAID via the

GISAIDR library [75]. The samples were compiled to a GISAID Episet with the ID `EPI_SET_230520sm` (accessible via https://doi.org/10.55876/gis8.230520sm). This resulted in a final dataset with the following sample distribution:

Lineage	**No. of samples**
B.1.1.529	2066
B.1.617.2	2044
B.1.621	2305
C.37	2478
Total	**8893**

Table 1: Number of samples per lineage used in the main dataset.

C. Input File Structure

Clients will interact with our viral strain classification through the client-side GUI application, which takes one FASTA file as input for classification. The format for FASTA files is as follows:

1. The file begins with the FASTA definition line, which is included before the nucleotide sequence. It must begin with a carat (">") and should be followed by a unique sequence identifier (SeqID) [76].

    - The FASTA definition line in this sequence should follow the following format: `>Reference/Database|AccessionID|DateCollected`
    - The SeqID should be unique for each nucleotide sequence.
    - The SeqID should not contain any spaces.
    - The SeqID should only contain the following characters: letters, digits, underscores (_), periods (.), asterisks (*), colons (:), hyphens (-), and number signs (#).

- All the information should be on a single line of text. The FASTA definition line should not include any hard returns.

2. The sequence begins after the FASTA definition line, and can contain returns. NCBI recommends that each line should be no more than 80 characters, and should only contain IUPAC symbols only, with "N" being used to symbolize ambiguous characters [76].

D. Preprocessing Techniques and Tools

To improve preprocessing and model training speeds, we followed Sim et al's framework and truncated the first 20kB of each of the genome sequences in the dataset. The truncated regions correspond to the regions of the sequences that preceded the S gene, which is the key driver of biological mutations between the SARS-CoV-2 strains [25]. Additionally, due to viral strains being typically defined based on their phenotypic characteristics instead of simple sequence similarity, alignment-free preprocessing methods were explored and utilized to prepare the viral sequences for strain classification. These methods transform raw genomic sequences into feature vectors, which can then be used as inputs for training classification models. The alignment-free tool that we used is Dashing [24]. Dashing provides a layer of abstraction from the raw sequence data by splitting an individual sequence into k-mers, converting each k-mer into a 64-bit hash, and then computing the HyperLogLog sketch of the sequence to estimate the cardinality of the hash sets. The output of this process, which was performed for each sequence in the dataset, was a vector containing the cardinality of 512 hash sets, represented by buckets. These buckets were then used as the features in training the machine learning model [25].

For the commands used during the Dashing process, we used shell scripts for transforming the resulting HLL sketches into a CSV file (from https://github.com/bjorgkav/concreteml-covid-classifier):

```
./dashing\_s512 sketch -k31 -p13 -S9 -F path.txt
```

Listing 1: Code listing of Dashing commands used to generate the HLL sketches (using 512-bit release)

```
#!/bin/bash
#clear output file (repeated testing)
rm output.txt

#assume no is 512 (based on Sim et al.'s paper)
echo -n "Accession ID," >> output.txt

no_of_features=512
for ((i=1; i<$no_of_features; i++))
do
    printf %s "feature_$i," >> output.txt
done

printf "%s\n" "feature_$i" >> output.txt

for f in FASTAFiles/*.hll
do
    content=$(./dashing_s512 view $f)

    #sed 's/[//g;]//g'
    echo "$f, $content" >> output.txt
done

```

```
24      #remove all occurrences of [ and ] in output
25      echo "Removing [ and ] from output and placing in csv
    ..."
26      echo "$(sed -i 's/[][]//g' ./output.txt)"
27      echo "$(sed -i 's/\<FASTAFiles\>//g' ./output.txt)"
28      echo "$(sed -i 's/\///g' ./output.txt)"
29      #| sed -e 's/\<FASTAFiles\>//g' | sed -e 's/\///g'
30      echo "$(sed -i 's/\.//g' ./output.txt)"
31      echo "$(sed -i 's/\<fastaw31spacing9hll\>//g' ./output
    .txt)"
32
33      mv ./output.txt ./output.csv
```

Listing 2: Code listing for reading HLL sketch values and generating a CSV file from the Dashing output (using 512-bit release)

The Dashing commands used in the study, shown in Listing 1, k was set to 31 using "-k31", the number of threads was set to 13 using "-p13", and the sketch size set to 9 using "-S9", for 2^9 bytes, with setting the sketch size affecting the number of features generated (a sketch size of 10 results in 1024 features being generated). After using Dashing to convert the individual SARS-COV-2 sequences into feature buckets, the output is joined with its respective `Lineage` and `Accession ID` values to form a dataset containing the GISAID accession ID, the lineage (also called variant or class) of the sample, and the features generated from Dashing, as shown below:

	A	B	C	D	E	F	G	H	I	J	K	L
1	Accession ID	Lineage	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7	feature_8	feature_9	feature_10
2	10005540	B.1.617.2	5	4	5	4	9	9	4	4	7	5
3	10019223	B.1.617.2	6	4	5	4	9	9	4	4	7	5
4	10019225	B.1.617.2	6	4	5	4	9	9	4	4	7	5
5	10020460	B.1.617.2	6	4	5	4	9	9	5	4	7	5
6	10025251	B.1.617.2	6	4	5	4	9	9	4	4	7	5
7	10065987	B.1.617.2	6	4	5	4	9	9	4	4	7	5
8	10070360	B.1.617.2	6	4	5	4	9	9	4	4	7	5
9	10070477	B.1.617.2	6	4	5	4	9	9	4	4	7	5
10	10070479	B.1.617.2	6	4	5	4	9	9	4	4	7	5
11	10070500	B.1.617.2	6	4	5	4	9	9	4	4	7	5
12	10070508	B.1.617.2	6	4	5	4	9	9	4	3	7	5
13	10070522	B.1.617.2	6	4	5	4	9	9	4	4	7	5
14	10070529	B.1.617.2	6	4	5	4	9	9	4	4	7	5
15	10070530	B.1.617.2	6	4	5	4	9	9	4	4	7	5
16	10070532	B.1.617.2	6	4	5	4	9	9	4	4	7	5
17	10070543	B.1.617.2	6	4	5	4	9	9	4	4	7	5
18	10072008	C.37	7	4	5	4	9	9	4	4	7	5
19	10072695	B.1.617.2	6	3	5	4	9	6	4	3	7	4
20	10080386	B.1.621	6	7	5	4	9	9	4	4	7	5
21	10080389	B.1.621	6	4	5	4	9	9	4	4	7	5
22	10080395	B.1.621	6	5	5	4	9	9	4	4	7	5
23	10081966	B.1.617.2	5	4	5	4	9	9	5	4	7	5
24	10116461	B.1.617.2	6	4	5	4	9	9	4	4	7	5
25	10123129	B.1.617.2	6	4	5	4	9	9	4	3	7	5

Figure 2: The preprocessed dataset as displayed via Microsoft Excel

E. Feature Selection Algorithm

To improve the performance of the logistic regression model, a feature selection algorithm is used to address the high number of features generated from Dashing the training sequences. Scikit-learn's k-best features algorithm (`sklearn.feature_selection.SelectKBest`), a univariate feature selection algorithm, was used to select the k highest scoring features based on univariate statistical tests [77, 78, 79]. This algorithm was chosen for its simplicity and fast running time, as well as its ability to ensure that feature selection results in a set with lower dimensionality. The k value was set to 20, effectively selecting the 20 highest-scoring features based on statistical tests. The feature selection algorithm was used to reduce the dimensionality of the dataset from the initial count of 512 features to 20 features.

F. Implementation of Classification

Following Sim et al's Covnita framework, a logistic regression classifier was chosen as the main classification model for the system due to its speed, simplicity in implementation, good performance with linearly separable data, and privacy advantages as discussed in section G. of Chapter III..

In terms of training, the `LogisticRegression` implementations of both scikit-learn and Concrete-ML were developed and trained on the same dataset in order to gauge the performance of the Concrete-ML model in comparison to the scikit-learn model, given that Concrete-ML is said to be compatible with scikit-learn workflows. Doing this also provided some information regarding the effect of Concrete-ML's quantization process on its accuracy and overall performance, with the scikit-learn model being used as a comparison. A standard train-test split of 80% training data and 20% testing data was utilized via scikit-learn's `train_test_split` function, and the univariate feature selection algorithm was applied to reduce the dimensionality of the dataset to 20 features. The training, development, compilation, and saving of the models were performed in the Concrete ML Docker container and Google Colab.

G. System Architecture

The following technologies and tools were utilized to implement the client and server-side applications:

- **Pygubu and Pygubu-designer** - A "what you see is what you get" (WYSIWYG) GUI designer for the Python's `tkinter` module, as well as `CustomTkinter`
- **Tkinter** - Standard Python interface to the Tk GUI toolkit
- **CustomTkinter** - A Python UI library based on tkinter that provides more modern UI widgets to allow for more up-to-date UI designs

- **Dashing** - Software tool for sketching similarities of genomes or sequencing datasets.
- **pandas** - Python software library for data manipulation and analysis
- **Concrete-ML** - Privacy-preserving FHE machine learning library built on Concrete
- **WSL2** - A Windows compatibility layer for Linux that enables running a Linux terminal environment on Windows without virtualization
- **scikit-learn** - Software machine learning library for the Python programming language
- **Django Framework** - Open source Python-based Model-View-Controller (MVC) web framework
- **Bootstrap** - Open-source CSS front-end development framework for web applications
- **Google Colab** - Cloud-based Jupyter notebook environment

The logistic regression classification model that is stored on the server-side application was developed using the `pandas`, `scikit-learn` and `Concrete-ML` libraries on the Google Colab notebook environment. The `pandas` library was used to parse the dataset used for training into a dataframe for pre-processing and training, while the `scikit-learn` library provided the modules for feature selection using the univariate feature selection algorithm, splitting the dataset into training and testing sets, and for the scikit-learn logistic regression model, which was used as a standard for comparison of model performance. Lastly, the `Concrete-M` library was used to create the system's final logistic regression classification model.

The client-side application serves as the primary method that clients of the system use to interact with the system, allowing the client to preprocess, encrypt, and send their SARS-CoV-2 sequences in one action. The app UI was built using CustomTkinter and Tkinter GUI libraries to develop the user interface of the application, with Pygubu and Pygubu-designer acting as rapid application development tools to allow more efficient UI development. Its sequence hashing and column selection functionalities were provided by the Dashing tool and `pandas` library, respectively, and the Concrete-ML library provided the client-side application with encryption and decryption functionality. The app communicates with the server-side application's API to send the preprocessed and encrypted sequences for prediction and uses its decryption functionalities to view the results of the prediction.

The server-side application was developed using the Django framework and Bootstrap and provides an API endpoint through which the client application passes its encrypted inputs and generated keys to the server. The server-side application then returns its encrypted prediction results to the client for decryption.

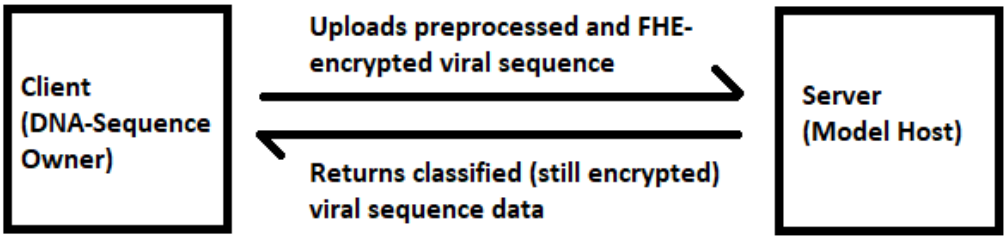


Figure 3: The client-server architecture.

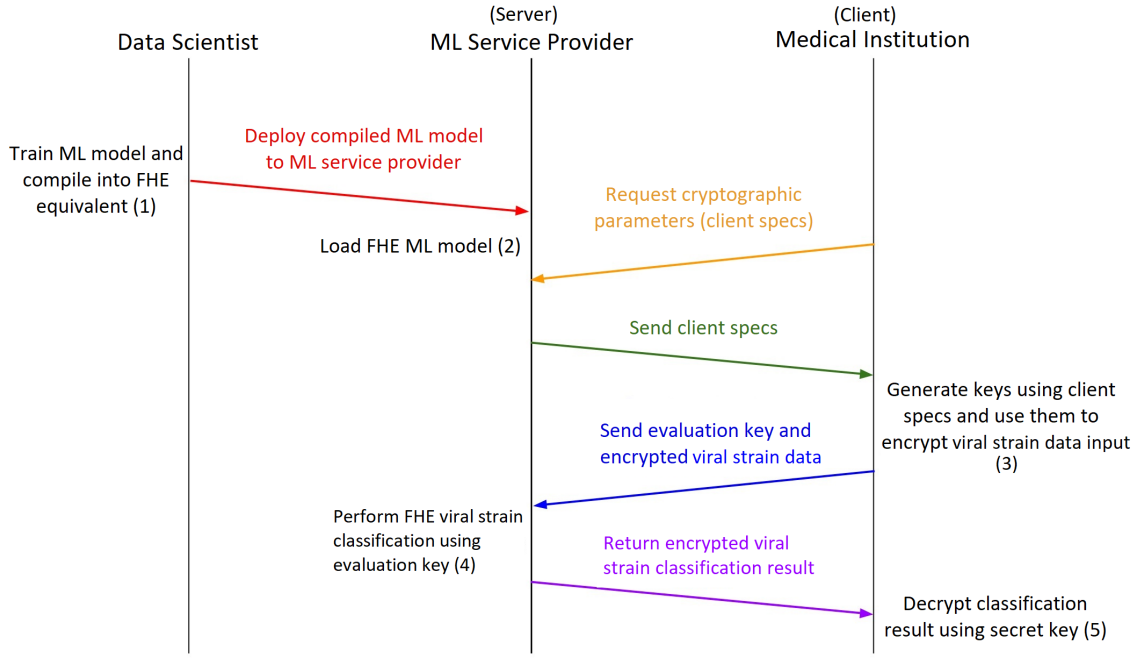


Figure 4: Detailed workflow of the client-server system.

As shown in Figure 4, the detailed workflow begins with the data scientists training, compiling, and saving the model into its FHE equivalent, with the last step being performed using the `FHEModelDev.save()` command, as outlined in subsection D. of Chapter III.. The data scientists then store the saved model (`server.zip`) on the server machine, along with the cryptographic parameters (`client.zip`) to allow the server to send them to client machines at their request. In development, these files (along with `versions.json`, also discussed in subsection D. of Chapter III.) are stored in the `Compiled Model` folder of the Django project directory (`BASE_DIR`), which the server then accesses using relative file paths as needed via Python's `os.path.join()` function. The training, saving, and storing process is similar to the steps taken to store a saved scikit-learn model on a server for a similar use case, with the main difference being that instead of storing only a single file on the server like in traditional scikit-learn workflows, the process involves storing two files on the server:

`client.zip` for dissemination to client machines, and `server.zip` for classification of clients' encrypted sequences. This allows data scientists that are already familiar with scikit-learn workflows to acclimate to Concrete-ML's workflow more easily.

Once the previous step is completed, the client (which represents users such as medical professionals, medical institutions, or researchers) can open the client-side app, which automatically requests the cryptographic parameters (`client.zip`, also discussed in subsection D. of Chapter III.) from the server using the `requests` library. It also requests `features_and_classes.txt` (discussed in subsection E. of Chapter III.) for column filtering. After the client-side application has ensured that `client.zip` and other required files have been downloaded, the client begins preprocessing and encrypting their data using the client-side application, which abstracts the Dashing, evaluation key generation, encryption of the clients' inputs behind a single file input, and filtering of non-selected features and class translation post-prediction (via `features_and_classes.txt`). It also accounts for any instruction set incompatibilities for the Dashing tool as discussed in section F. of Chapter III., ensuring that the correct version of the Dashing tool is utilized in the sequence hashing set. The key generation and encryption steps were performed using Concrete-ML's API, in particular, through the following commands from the `FHEModelClient` module:

- `generate_private_and_evaluation_keys()` for key generation
- `get_serialized_evaluation_keys()` to access the keys within the program for saving and sending to the server
- `quantize_encrypt_serialize()` to facilitate the encryption of the inputs

The client-side application also automatically sends the encrypted file input to the server for classification along with the generated evaluation keys for classification. This is done by using Python's `requests` library to interact with the classification endpoint server's API. In development, the classification API endpoint

is `http://localhost:8000/start_classification`, with `localhost:8000` being a locally-run Django-development server. Once the server receives the clients' evaluation keys and encrypted viral strain sequences, it then performs FHE inference on the sequences using the `FHEModelServer.run()` command. It then saves the encrypted predictions to separate files, compresses the encrypted predictions to a ZIP file, and sends the ZIP file back to the client via the client-side application, which automatically unpacks the ZIP file and uses the `FHEModelClient.deserialize_decrypt_dequantize()` method to facilitate the decryption of the prediction results using the keys generated earlier in the workflow and the display of the results for the client to view.

H. Technical Architecture

The requirements for running the system at full capacity include the following specifications:

- **Operating System**: Linux or Windows Subsystem for Linux 2
- **Memory**: 4GB minimum
- **Storage space**: 7GB free disk space for Concrete-ML package and dependencies

An estimated 7 gigabytes of free disk space is required to run the full system due to `Concrete-ML`'s dependencies, including `torch` and `concrete`. Additionally, as the client application requires the Dashing tool, the appropriate shell scripts, a text file listing the selected features for classification and original class labels (`features_and_classes.txt`), and the `client.zip` file, which holds the required cryptographic parameters for key generation, encryption, and decryption, the application requires an internet connection to download the files needed from the system's Github repository.

V. Results

A. ConcreteML Performance

A..1 Test Machine Specifications and Details

All test results obtained and displayed as tables in this section and the next were averaged over ten runs to account for the stochastic nature of the data and algorithms. The performance of ConcreteML was evaluated on a machine with the following specifications:

Test Machine Specifications	
Operating System	Windows 10 Home Single Language version 22H2
Linux Compatibility Layer	Windows Subsystem for Linux 2 (using Ubuntu 22.04 LTS)
Processor	11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz
Installed RAM	16.0 GB
System Type	64-bit

Table 2: System specifications for the test machine used to gather model performance data

A..2 Training Workflow

The training workflow is as follows:

1. A total of 8994 SARS-CoV-2 genomic data sequences, all obtained from GISAID, had their first 20kB truncated following Sim et al's initial CoVnita workflow [25]. They were then converted into feature values using the Dashing tool, specifically the `sketch` function of the Dashing tool, which is further discussed in section F. of Chapter III.. For the parameters, k (used to split the data into k-mers) was set to 31. The number of threads ("-p") used for the operation was

set to 13, while the sketch size ("-S") was set to 9 (for 2^9 bytes). This results in a sketch for each sequence that contains a total of 512 features. The exact EPISETs for the dataset can be found in subsection B. of Chapter IV..

2. The dataset was loaded into a `pandas` DataFrame in the training environment and the classes were label encoded using the scikit-learn library `sklearn.preprocessing.LabelEncoder`.

3. The feature data used in prediction was then cast as the float data type to adhere to the strict formatting standards that Concrete-ML's model fitting function enforces.

4. The `sklearn.feature_selection.SelectKBest` univariate feature selection algorithm was used to reduce the dimensionality of the dataset. The number of features ($n\_features$) was set to 20, leading to the algorithm selecting only the 20 highest-scoring features on the dataset according to univariate tests.

5. Scikit-learn's (`sklearn.model_selection.train_test_split`) function was used to split the data into a training set and a testing set, with a split of 80 percent training data and 20 percent testing data.

6. The models are then initialized and fit to the training data set. For the FHE model, an extra step is performed:

    - Once the quantized plaintext model has been fit to the training data, it can then be compiled to its FHE equivalent to produce the trained FHE model.

A..3 Models Trained

A total of three different logistic regression models were trained:

- The **plaintext model**, which uses scikit-learn's logistic regression model. This serves as the baseline against which the other two models (that use Concrete-ML's implementation of the logistic regression model) will be compared.

- The **quantized plaintext model**, which uses Concrete-ML's logistic regression model with FHE disabled. This differs from the plaintext model in that there is still a quantization step included in the fitting process despite not using FHE. As discussed in subsection D. of Chapter III., quantization converts inputs, model weights, and all intermediate values of the inference computation to integers, which can lead to a level of degradation in performance. It is also a mandatory step to perform before compiling a model to its FHE equivalent, as TFHE, the encryption scheme utilized by Concrete-ML, is currently limited to 16-bit integers. This model is included in the setup to determine the extent of the effect that quantization has on the models' accuracy.

- The **FHE model**, which is the quantized plaintext model with FHE enabled. This will be compared with both the plaintext and quantized plaintext models to see if the use of FHE in the model adversely affects model performance to a tolerable extent.

A..4 Comparison of Logistic Regression with other ML Algorithms

A comparison of various machine learning algorithms available to Concrete-ML was performed to determine the most optimal model for viral strain classification and to verify the performance of Logistic Regression compared to other algorithms. A total of four machine learning algorithms were compared: Logistic Regression, Linear Regression, Support Vector Classifier, and Random Forest. Both the scikit-learn and Concrete-ML implementations were compared as well, resulting in the performance of 8 algorithms being compared. The results are as follows:

Model Type	Linear Regression	Random Forest	SVC	Logistic Regression
Plaintext	95.309591%	99.505340%	99.066892%	99.280495%
Quantized Plaintext	95.315662%	99.072513%	99.145587%	99.252389%
FHE	95.315662%	99.072513%	99.145587%	99.252389%

Table 3: Accuracy comparison between different algorithms for both scikit-learn and Concrete-ML, averaged over 10 runs

Table 3 shows that the logistic regression algorithm performed the best in the quantized plaintext and FHE model runs, while achieving the second-highest results in the plaintext model runs. As discussed in subsection G. of Chapter III., this performance can also be attributed to logistic regression's ability to perform well when used with linearly separable classes and classification [69, 71]. This also extends to the use case of viral strain classification, as genetic patterns in sequences, which come about as a result of genetic divergence over time, are linearly separable across aligned amino acid positions [71].

A..5 Model Accuracy

The performance of the Concrete-ML models (the Quantized Plaintext, and FHE models) was compared against the performance of the scikit-learn (Plaintext) model. Three performance metrics were explored to gain an understanding of their performance:

- The standard classification accuracy of the model's predictions on the test set. It is defined as the number of correctly predicted samples divided by the total

number of samples the model was tested on. The metric is considered one of the most common and simplest performance metrics to implement and works well when used with a more balanced dataset. This makes it suitable for use with our own fairly balanced data, as it provides a simple and easy-to-interpret performance metric that allows us to evaluate the performance of the model at a glance [80]. However, this metric also has shortcomings in that it can be influenced by class imbalances in datasets and also cannot make use of probabilities of predictions, thereby limiting the information that we can gain regarding the model's performance, such as false positives and negatives [81, 82]. This metric was implemented using scikit-learn's `sklearn.metrics.accuracy_score` module.

- The models' Area Under the Receiver Operating Characteristic Curve (ROC AUC) score. The ROC AUC score is calculated by computing the area under the Receiver Operating Characteristic (ROC) curve, which features the true positive rate (TPR) on the Y axis, and the false positive rate (FPR) on the X axis across several classification thresholds [77, 83, 80]. With scikit-learn's `sklearn.metrics.roc_auc_score` module, this area is computed using the models' prediction scores (obtained using the model's `predict_proba()` method). For a multiclass problem such as ours, a One-vs-Rest strategy was utilized to split the multiclass problem into one binary classification problem per class. Next, macro averaging was performed, where the ROC AUC scores for each class were calculated, and their unweighted mean was computed to produce the final result, ensuring that all classes would be treated equally. The OvR strategy and macro-averaging settings were specified using the `multi_class='ovr'` and `average='macro'` settings for the `roc_auc_score()` function, respectively [77]. In comparison to the accuracy metric, the ROC AUC score makes use of the model's TPR and FPR, providing us with a more com-

prehensive idea of the model's performance. It is also more capable of handling imbalances in datasets compared to the accuracy metric due to using the prediction scores of the classes instead of accuracy's simple ratio of correctly predicted samples to the total number of samples [84, 82].

- The models' confusion matrix, which is a table that summarizes the number of correctly predicted samples and incorrectly predicted samples while providing a more in-depth visualization of how these samples were classified. This helps us understand the behavior of a particular classifier and see how or where it makes misclassifications [85, 86]. Scikit-learn's `sklearn.metrics.confusion_matrix` is used to implement our classifier's confusion matrix, while `sklearn.metrics.ConfusionMatrixDisplay` is used to visualize the matrix in a more informative manner.

The results of our tests were averaged over ten runs to account for the stochastic nature of the data and the train-test split algorithm, and are discussed in subsection A. chapter VI..

B. Client-Server Classification System

An FHE-enabled client-server classification system was developed, with the client primarily interacting with the client side of the application and only interacting with the server during the classification of the client's encrypted sequences.

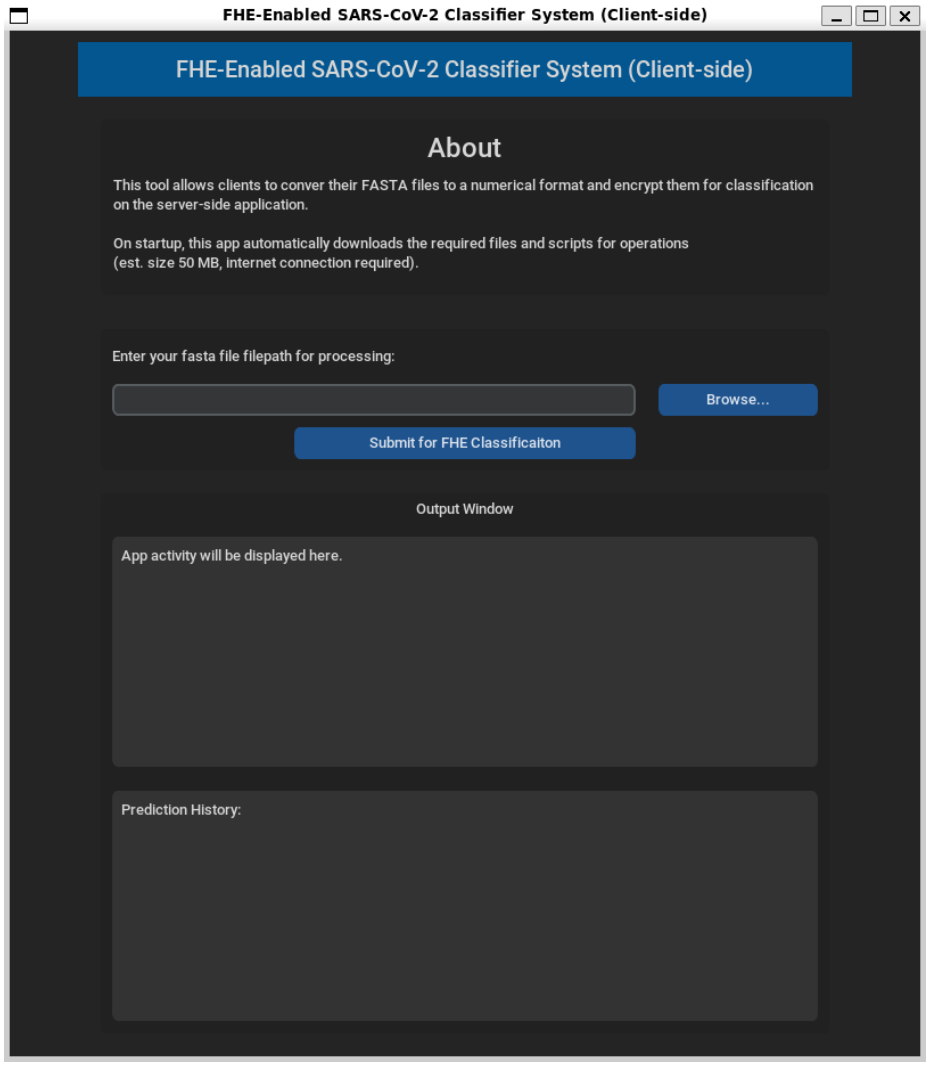


Figure 5: The Client GUI application after starting up.

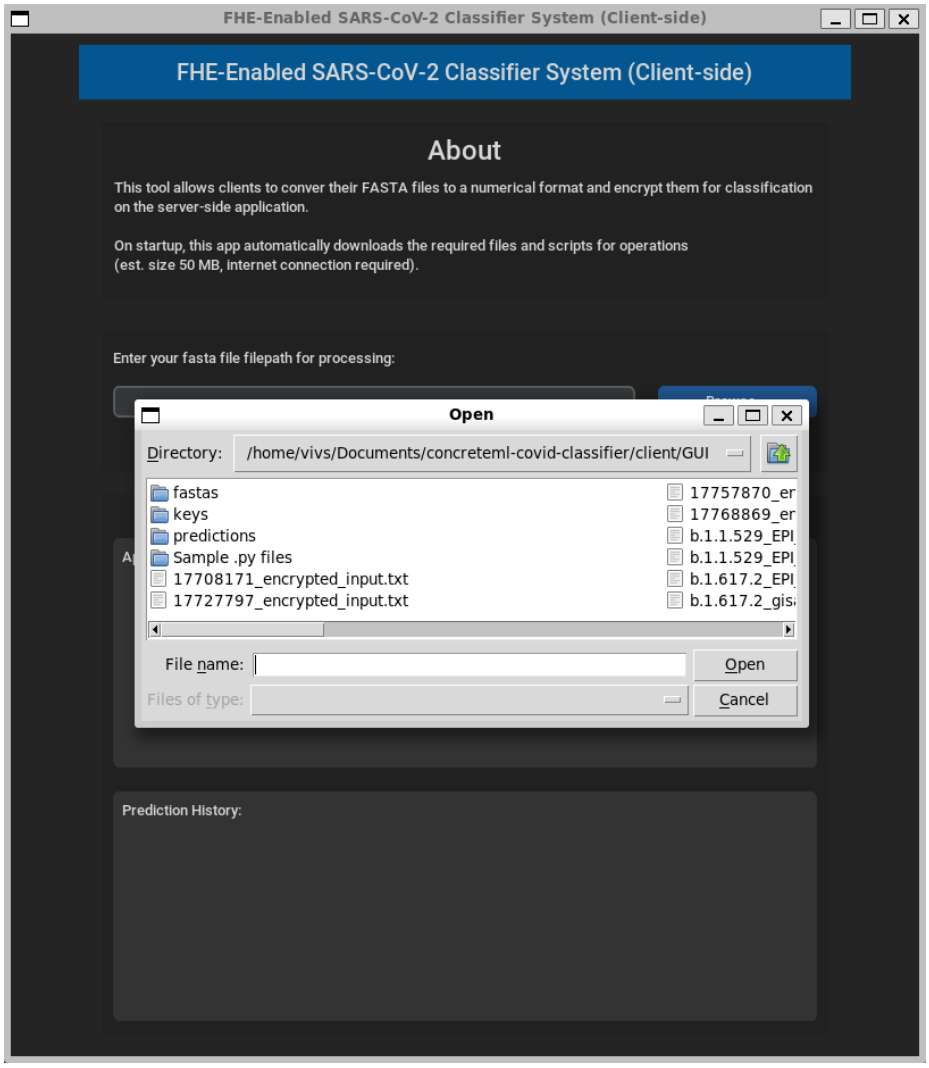


Figure 6: The Client GUI application's file selection interface.

The client and server sides of the system were developed as two separate applications, with the client application being developed as a desktop GUI application that makes use of the `CustomTkinter` and `Tkinter` packages to facilitate the rendering and windowing of the application's user interface. The server application was developed as a web application using the Django framework. The files `client.zip` and `server.zip`, which were generated when the Concrete-ML model was saved, were stored or otherwise made available for download for their respective applications. The package requirements for the main system were also saved to `requirements.txt`, which can be found on the system's GitHub repository at

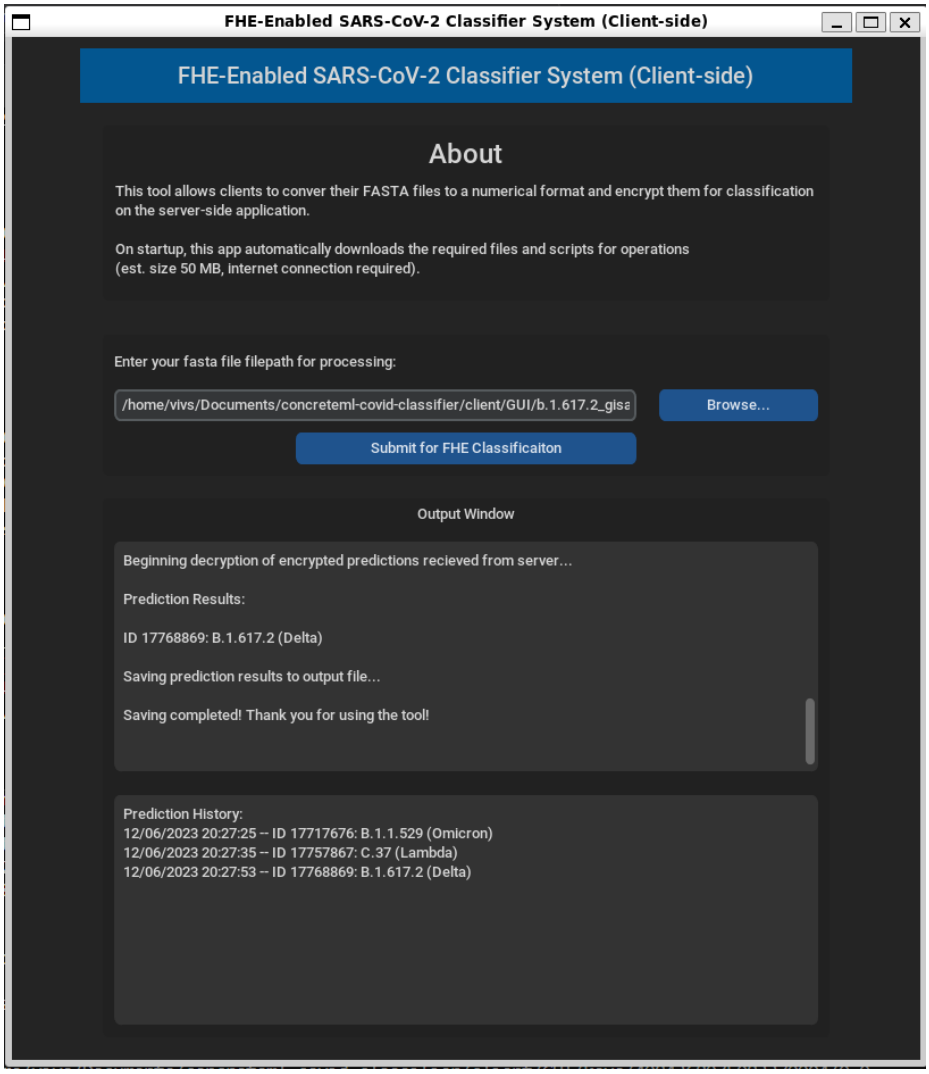


Figure 7: The Client GUI application's output window showing prediction results.

https://github.com/bjorgkav/concreteml-covid-classifier.

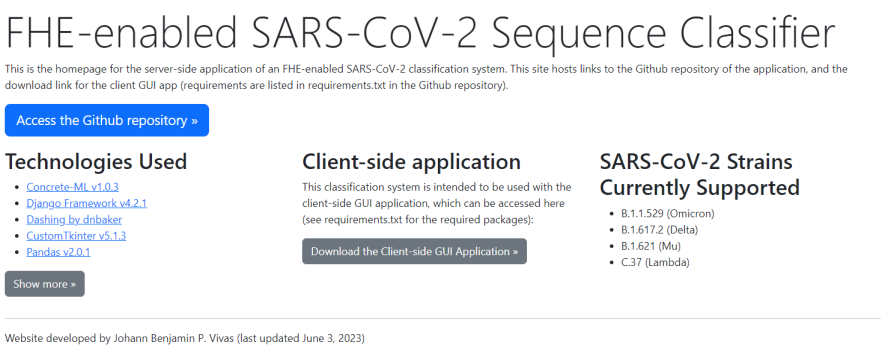


Figure 8: The server-side web application developed in Django.

While the server side of the system is not intended to be interacted with, a UI interface was designed to allow the web application to provide visitors with information regarding the system, its technologies, currently supported strains, and links to its GitHub repository.

VI. Discussions

A. Accuracy

The results in terms of training and testing dataset accuracy and performance are as follows:

Logistic Regression Model	Accuracy	ROC AUC Score
Plaintext	99.280495%	0.999879
Quantized Plaintext	99.252389%	0.999877
FHE	99.252389%	0.999877

Table 4: Model performance in terms of accuracy and AUROC score (One vs Rest)

Logistic Regression Model	Accuracy Loss vs Plaintext	ROC AUC Score Loss vs Plaintext
Plaintext	0.000000%	0.000000%
Quantized Plaintext	0.028106%	0.000175%
FHE	0.028106%	0.000175%

Table 5: Average loss of performance of FHE classification compared to scikit-learn for both standard accuracy and AUROC score

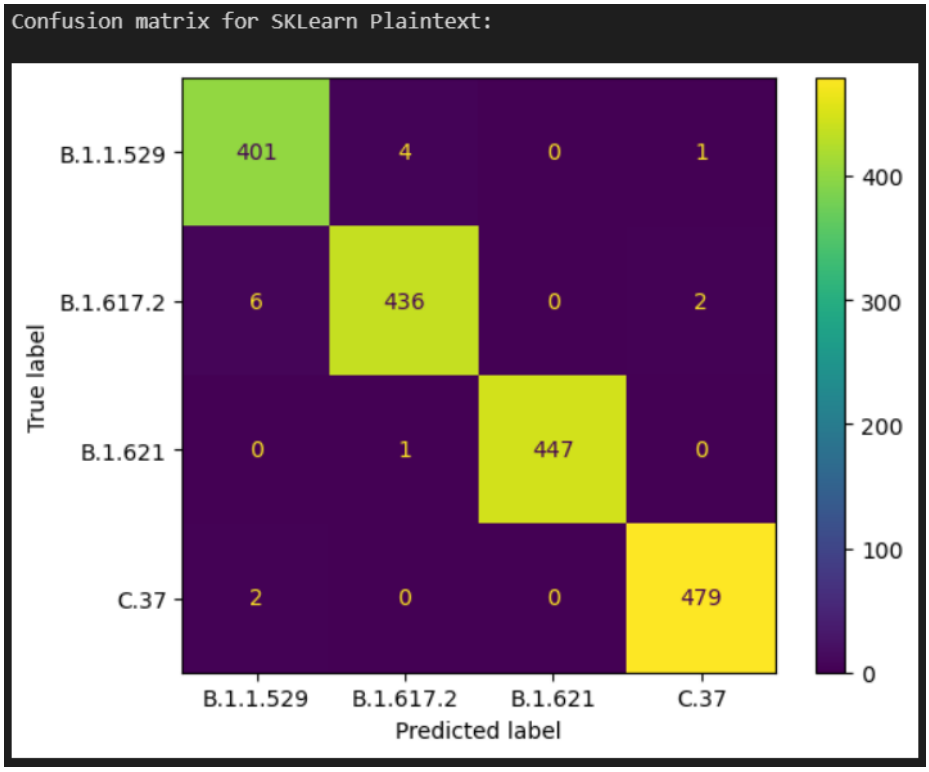


Figure 9: The confusion matrix for the scikit-learn (Plaintext) model

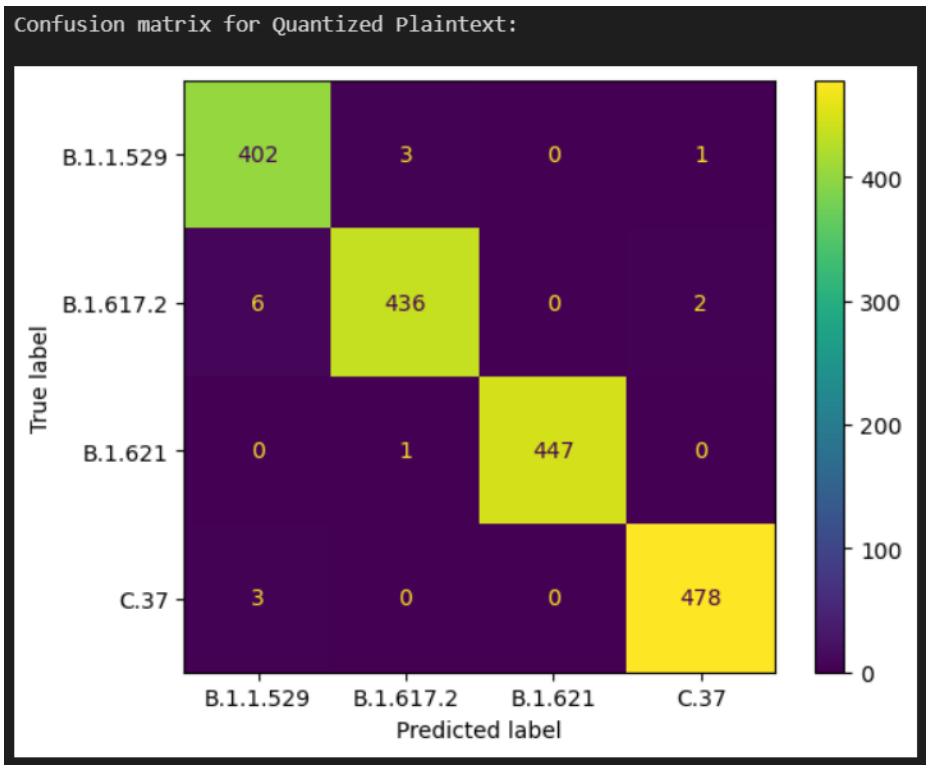


Figure 10: The confusion matrix for the Concrete-ML (Quantized Plaintext) model

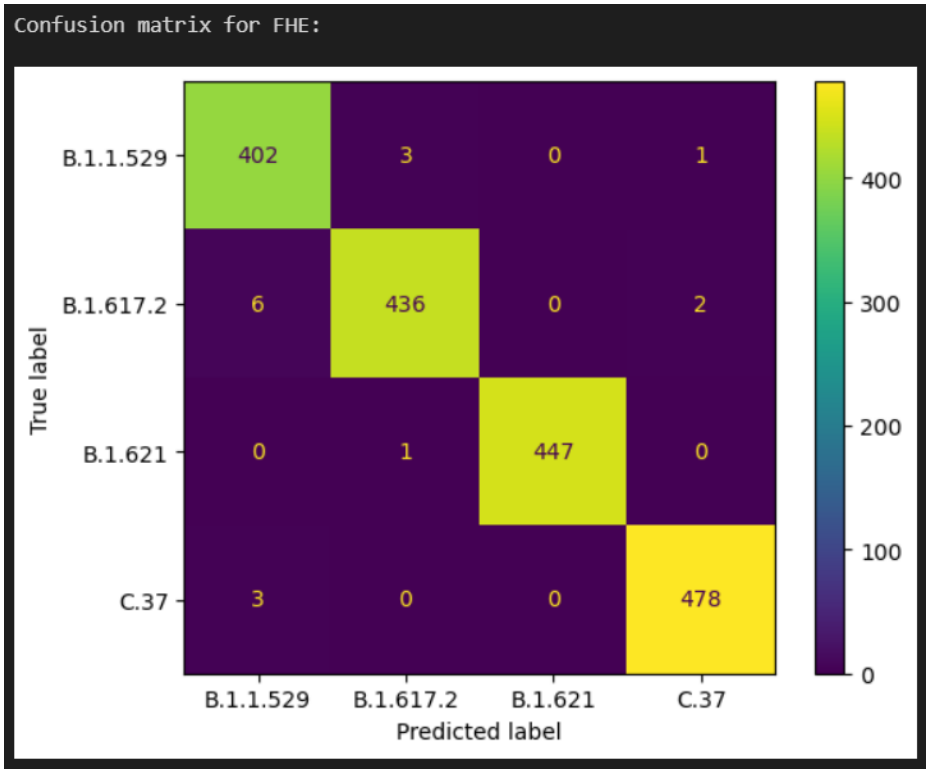


Figure 11: The confusion matrix for the Concrete-ML (FHE) model

As can be seen from Tables 4 to 5 and Figures 9, 10, and 11, all three models (Plaintext, Quantized Plaintext, and FHE) were able to achieve relatively high scores in all three performance metrics when tested against the test set consisting of 20% of the total samples (1779 samples). It can also be seen that the performance of the quantized plaintext and FHE models are nearly identical to the performance of the plaintext model for all three metrics. Additionally, at some points, the performance of the Concrete-ML model achieves a near-equal level of accuracy compared to the baseline scikit-learn model's performance when scored on the accuracy metric. The loss of performance from both the quantization of the inputs in order to fit the format of the Concrete-ML model, along with the conversion of plaintext to its FHE equivalent, is below 0.1 percent in all cases, indicating a relatively insignificant, and thus tolerable, loss in accuracy.

B. Error Analysis

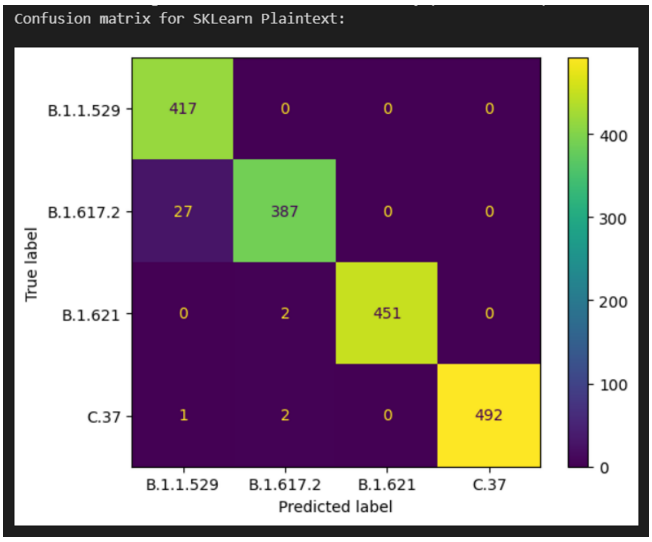
An error analysis was also conducted by performing runs and comparing the results of the confusion matrices per model to determine the extent of the differences in their classification ability as a result of the quantization and encryption steps undertaken for the Concrete-ML models (the Quantized Plaintext and the FHE models). The results of all runs performed can be categorized into three cases:

- Case 1, Where the Quantized Plaintext and FHE model performed the best in the run, with scikit-learn misclassifying one more sample in total than was misclassified by the Concrete-ML models. This case is the least common among the cases, occurring only once in the runs performed, with the FHE vs Plaintext similarity score (shown in the results below) indicating a 99 percent similarity.

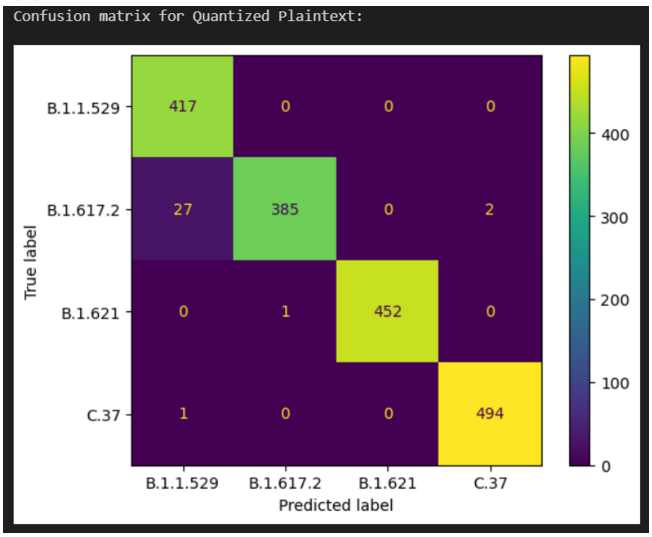
- Case 2, where all models achieved the same degree of performance. This means that any misclassifications made by the scikit-learn Plaintext model were also misclassified by the Concrete-ML models. This was the most common case among the three cases, with the FHE vs Plaintext similarity score at 100 percent.

- Case 3, where the scikit-learn Plaintext model performed the best among the three models. In this case, the Concrete-ML models misclassified one more sample in total than the Plaintext model, resulting in an FHE vs Plaintext similarity of 99 percent. This was the second most common case among the runs.

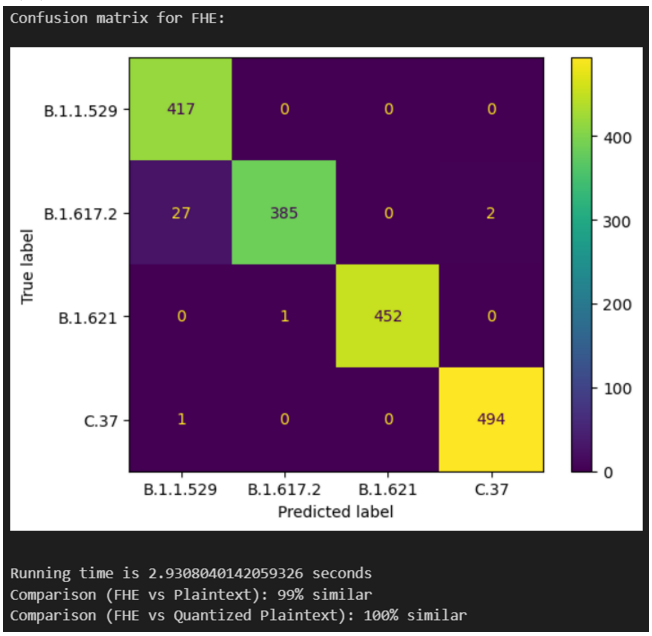
The following three runs depict these three cases:



(a) scikit-learn Plaintext Model Confusion Matrix

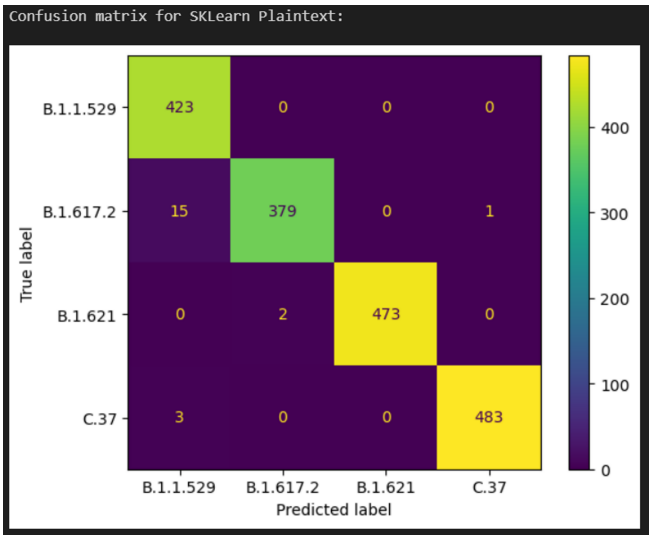


(b) Quantized Plaintext Model Confusion Matrix

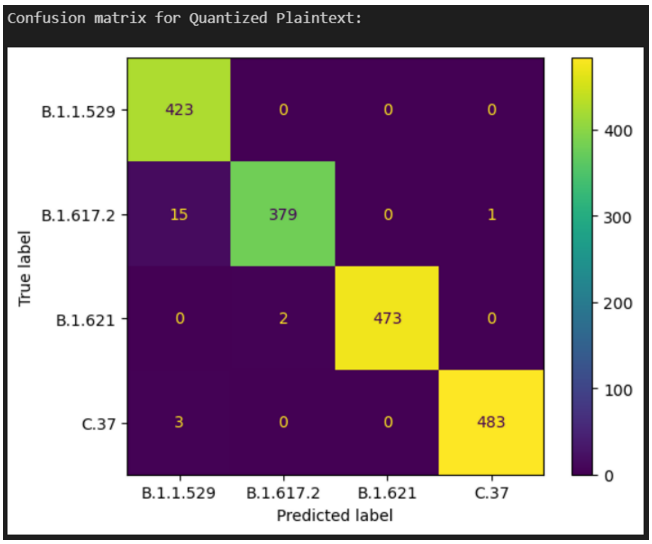


(c) FHE Model Confusion Matrix

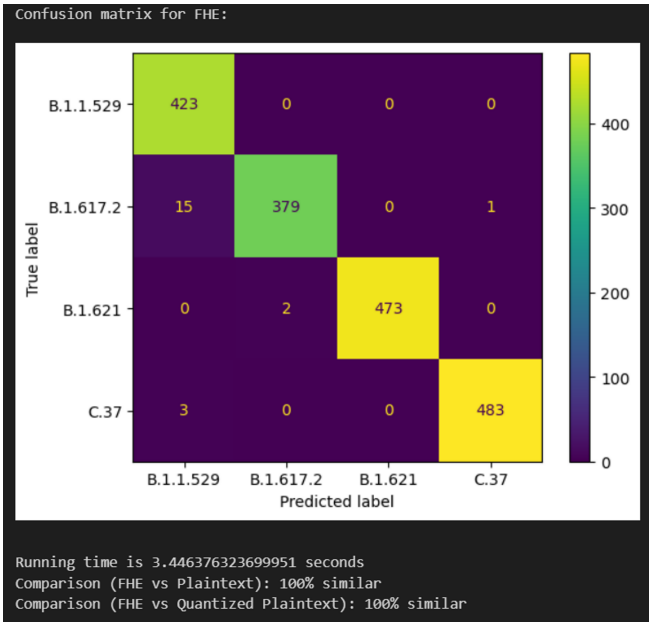
Figure 12: Confusion matrices of the three models for error analysis run 1



(a) scikit-learn Plaintext Model Confusion Matrix

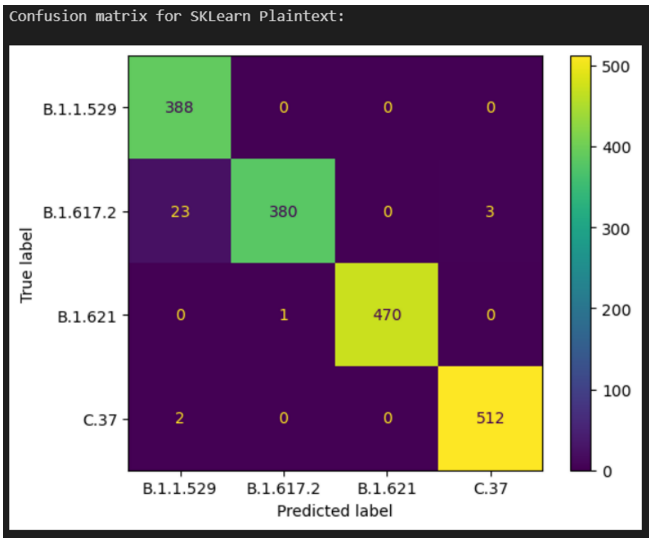


(b) Quantized Plaintext Model Confusion Matrix

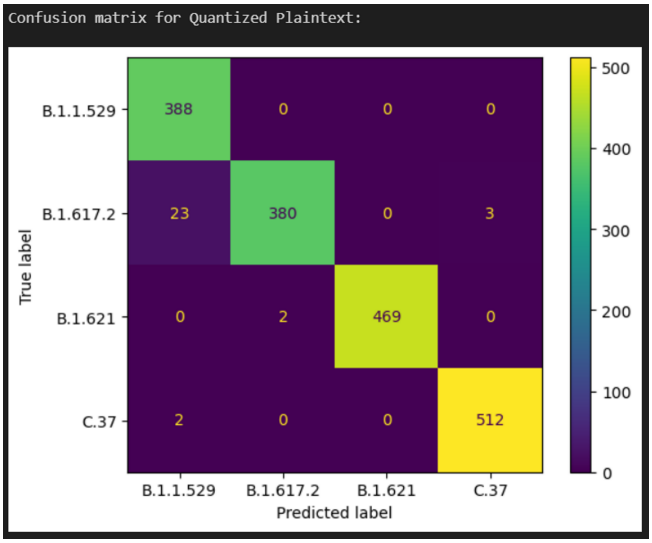


(c) FHE Model Confusion Matrix

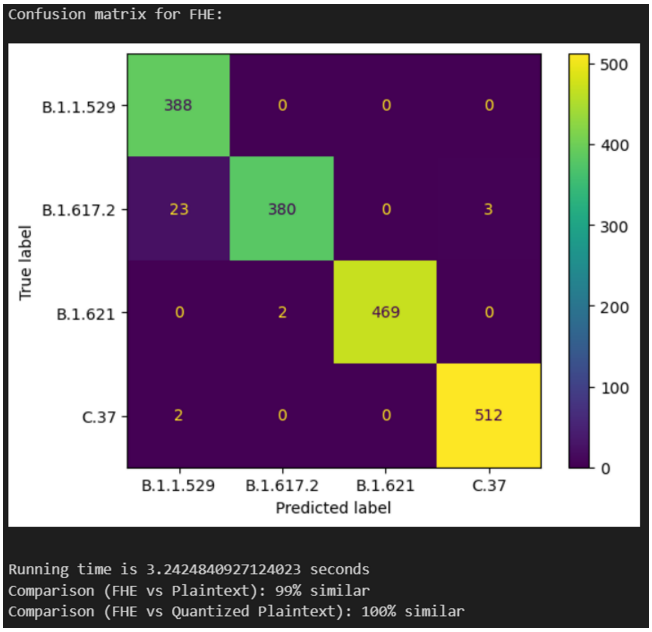
Figure 13: Confusion matrices of the three models for error analysis run 2



(a) scikit-learn Plaintext Model Confusion Matrix



(b) Quantized Plaintext Model Confusion Matrix



(c) FHE Model Confusion Matrix

Figure 14: Confusion matrices of the three models for error analysis run 3

As can be seen from figures 12, 13, and 14, the extent of the differences between the models is at most 1, which can still be considered a tolerable level of error in most use cases.

C. Model Training and Classification Speed

Model Type	Training time (nanoseconds)	Training time increase vs plaintext	Quantization time (nanoseconds)
Plaint.	95986247.062683	0.000000%	-
Quant.	115684342.384338	20.521789%	19698095.321655

Table 6: Average training time for each model, with respective time increase

Table 6's results show that the training time for the Plaintext, Quantized Plaintext, and FHE models are relatively close, with at most a 20 percent increase from the averaged Plaintext training time. We can attribute the increase compared to the Plaintext results in training time to the quantization step that is performed by the Concrete-ML models to respect FHE constraints (which are that FHE is, at the time of writing, limited to 16-bit integers) [60]. Specifically, linear models such as `LogisticRegression` perform quantization after training is performed [61]. The quantization time can be computed by subtracting the plaintext training time from the Quantized Plaintext model training times. The results of that subtraction show that an average of around 19698095 nanoseconds (or around 0.019698095 seconds) is added to the training time of the Quantized Plaintext due to the additional post-training quantization process.

We also evaluated model performance in terms of running time for the classification of new samples using the models. The running time in seconds was obtained using Python's `time.time()` function to obtain the starting and ending time of the task and then compute the difference. These results were then averaged over ten runs

to account for randomness. The results are displayed as follows:

Model Type	Test Set Pred. Time (nanosec.)	Avg. per Sample Time (nanosec.)	Increase in Pred. Time
Plaint.	267052.650452	150.113913	0.000000%
Quant.	819325.447083	460.553933	206.802964%
FHE	955438.613892	537.064988	257.771628%

Table 7: Average model prediction time on the entire test set and per sample

The results of the prediction time comparison are shown in Table 7. There is an expected increase in prediction time from the plaintext model to the Concrete-ML models, as the additional quantization step is required for both models, regardless of the use of FHE [61]. This additional step leads to more complex equations and a longer running time because performing matrix multiplications or convolutions using quantized values requires the computation of important quantization parameters, which are used in equations over those quantized values to ensure that the final result is also quantized. These parameters include the scale factor (the value used to map a float number to its integer representation and vice versa) and the zero point (the value that zero takes in the target representation) [61, 87, 88, 89]. In our case, this results in a nearly 210 percent increase in prediction running time compared to the plaintext model for the quantized plaintext model, and an almost 260 percent increase from the plaintext model's prediction running time for the FHE model.

Comparisons	Entire test set	Per sample time	Training time

Continued on next page

Table 8: Summarized model slowdowns in terms of running time for prediction and training

Plaintext vs Quantized Plaintext	2.07x slower	2.07x slower	0.21x slower
Plaintext vs FHE	2.58x slower	2.58x slower	0.21x slower
FHE vs Quantized Plaintext	0.17x slower	0.17x slower	0x slower

Table 8: Summarized model slowdowns in terms of running time for prediction and training

We summarize the training and prediction time results in table 8, with results indicating that the Concrete-ML models (Quantized Plaintext and FHE) are more than two times slower than the scikit-learn Plaintext model in terms of prediction time, and only being slightly slower than the Plaintext model during training. Additionally, the FHE model exhibited only a slight increase in the prediction time vs the Quantized Plaintext model. Though the results indicate that the use of quantization and encryption significantly increases the running times for training and prediction on the models, the actual training and prediction times themselves can still be considered tolerable for most use cases, as the training and prediction times stay below 0.2 seconds on average.

Run no.	**FHE Compilation time in nanoseconds**
1	2047241449
2	2026603460
3	1970088005
4	2024210453
5	1939769745

Continued on next page

Table 9: Average of FHE model compilation time

6	2018125772
7	2008958340
8	2048562288
9	2002124786
10	1998833179
Average	**2008451748**

Table 9: Average of FHE model compilation time

The compilation time for the FHE model was also recorded and averaged over ten runs. The results of the runs, as seen in table 9, show an average FHE model compilation time of roughly 2.01 seconds. This indicates that the model compilation time is still also relatively tolerable in most use cases given that the model compilation step only takes place during the training phase illustrated in Figure 4 where the classification model is trained, compiled and saved.

D. Key and Ciphertext Size Comparison

Eval Key Size (kB)	**Private Key Size (kB)**	**128-bit security RSA Key Size in kB**	**Increase in size vs RSA Standard**
0.0230	4.0000	2.4140	65.7001%

Table 10: FHE eval key size and private key size comparison vs RSA private key of similar security level

As the key length of an encryption scheme is an important security parameter, a comparison of Concrete-ML's key size with the standard RSA key size of a similar security level was done to assess whether the encryption scheme used by the library

can be considered efficient in terms of key size. Concrete-ML's security level is currently fixed to 128-bits, so an RSA key that provided 128-bit security was used for comparison, the results of which can be seen in Table 10 [90, 91]. The key size comparison shows that Concrete-ML's key size is significantly larger than the standard RSA key size, indicating a loss in memory efficiency, as the key generated for encryption requires a much larger key size to achieve a similar security level as that of the RSA encryption scheme.

Encryption	**File Size (kB)**
Plaintext Input	0.0436
Concrete-ML FHE	200.1970
128-bit security RSA Encryption	0.3750
Percentage Increase in Size from RSA	53285.8667%

Table 11: Ciphertext size comparison between Concrete-ML's FHE encryption and standard RSA encryption using a 3072-bit key

In addition to the key size comparison, a ciphertext size comparison was performed to further assess the memory efficiency of the Concrete-ML's encryption function and scheme. Across fives runs, each using different strains except for the fifth run, the ciphertext size of the encrypted data stayed consistent at 200.197 kB. This is expected, as all samples that were encrypted had undergone the same preprocessing steps (truncation, Dashing, column dropping according to selected features) before being encrypted and were encrypted using the same encryption function, that being the encryption function described in section G. of Chapter IV.. We also see a significant increase in Concrete-ML's ciphertext size compared to the RSA encryption size, also indicating a loss in memory efficiency, requiring significantly more storage space to achieve a 128-bit security level than the RSA encryption scheme.

E. Issues Encountered in Development

We faced certain issues during the development of the system itself, which mainly involved errors raised by the Concrete-ML library due to its rather specific and strict formatting. There is an issue whenever a `pandas` dataframe with an integer `dtype` was passed to the Concrete-ML `LogisticRegression` model without being cast to the float data type. This might be due to the inclusion of the quantization step in the Concrete-ML process, which converts float values into an integer format that can be used with the Torus Fully Homomorphic Encryption (TFHE) scheme, which only supports integer calculations. We addressed this by casting the preprocessed dataset to the float datatype during training. A similar issue is encountered when using the compiled Concrete-ML model to make predictions on data allocated to the test set during the train-test split step of the workflow, or completely new SARS-CoV-2 sequences that have been preprocessed as described in the training workflow in Chapter V.. The issue was resolved by ensuring that inputs for testing and prediction on new samples were cast to a `numpy` array with the data type `uint16` before passing it into Concrete-ML's prediction function. It is important to note that these formatting issues do not apply to the Plaintext (scikit-learn) model, as their model does not require a specific data type due to the lack of any quantization functionality for the model.

F. System Assessment

Both the client-server classification system and classification model that was developed as a result of this study were able to meet all the required objectives and planned functionalities illustrated in Chapter I.. Concerning the issues that were brought up in this study's earlier chapters, it seems very likely that the development of this application and its excellent performance in the privacy-preserving classification of encrypted SARS-CoV-2 sequences could stand as a testament to the viability of im-

plementing privacy-preserving technologies such as Fully Homomorphic Encryption into the tools and methods used in the medical field and its practices. More importantly, the development of the system and its corresponding model exhibits the substantial potential benefits in terms of privacy and protection of intellectual property that medical professionals, patients, and researchers can gain from incorporating FHE into their practices.

The system stands apart from its reference works as an application developed using Concrete-ML, an open-source FHE library that was developed specifically to allow for the use of Machine Learning computations on encrypted data, thus allowing for more specialized implementations of FHE that can also work well in tandem with currently existing scikit-learn workflows. In particular, Sim et al's work on CoVnita presented a viral strain classification framework that implemented FHE using the Microsoft SEAL library, while Akavia et al. utilized the IBM HElayers FHE library for the same purpose [25, 38, 33, 47]. Both of these libraries may be considered more general-purpose FHE libraries, and thus lack the built-in machine learning models and ease of use that Concrete-ML offers due to its implementation in the high-level Python programming language and its ability to integrate smoothly into scikit-learn workflows, while also providing a simple interface that allowed users to convert their plaintext model into an FHE equivalent through its `compile()` function. Concrete-ML also boasts a plethora of built-in functions that facilitate and simplify the process of key generation, encryption, and decryption significantly, making it much more beginner and user-friendly in comparison to other FHE libraries. This study also provides a comparison between the standard plaintext scikit-learn ML model and its corresponding Quantized Plaintext and FHE ML models, which has not been done in previous works, and provides a great deal of insight into the current state and potential direction of FHE ML moving forward.

The FHE-enabled classification system is also easier to learn than some of its coun-

terparts due to Concrete-ML's comprehensive documentation, and is also quick and simple to update or adapt for other diseases, as such processes mainly only require the replacement of `client.zip`, `server.zip`, along with the `selected_features.txt` file, and the installation of updated packages as necessary. The implementation of the client-side as a desktop GUI app which then consumes the server's API instead of a separate web application further differentiates the system from its contemporaries while also highlighting the privacy advantage of using a desktop application as it prevents any servers from seeing any form of the client's data before it is encrypted.

With all the points discussed above, it is clear that the system serves as a useful tool that allows clients such as medical professionals and researchers to study and perform viral strain classification on viral strain sequence data without putting patients' sensitive data at risk of being leaked to ML service providers.

VII. Conclusions

We address the widespread privacy concerns related to the use of ML outsourcing to allow for better-informed medical decisions and higher-quality healthcare by developing an FHE-enabled viral strain classification system for SARS-CoV-2 genomic sequences that implements a client-server architecture to allow for maximized privacy of data and ease of use through a GUI application. The system was developed using the open-source FHE ML-specialized library Concrete-ML, providing the system a relatively high degree of customization, adaptability, intelligibility (as a result of its similarity to, and compatibility with, the scikit-learn library and workflows). The client side of the system uses a combination of feature extraction from submitted genomic sequences through the Dashing tool and univariate feature selection through the `sklearn.feature_selection.SelectKBest` module to transform the client's submitted genomic sequences into a set of features with comparatively low dimensionality. The client GUI application then generates a set of keys for encryption, encrypts the aforementioned set of features using the keys, and saves them into files for uploading to the server by consuming its API, where the encrypted inputs are then classified using the Logistic Regression model stored on the server and sent back to the client for decryption.

The system serves as a solution to the privacy issue of medical professionals, medical institutions, scientists, and other researchers using patients' genomic data in machine learning applications to assist them in providing better healthcare. By introducing FHE into clients' workflows, the affected patients' privacy is significantly less likely to be endangered when their genomic data are used as inputs in machine learning services and applications, as the data can remain encrypted even while being operated on. The implementation of this technology also allows clients to continue the use of ML services and ML outsourcing without risking patients' privacy and potential legal action.

VIII. Recommendations

In terms of improvements to the current version of the classification system, should future studies take inspiration from this study, we recommend that the studies' authors explore the implementation of batch processing to allow for a more efficient and intuitive experience when working with larger datasets and collections of sequences, as well as the addition of a cache such that sequences that were already previously classified would no longer be re-classified unless specified, potentially cutting down on memory usage and running time when performing predictions.

In addition, exploring the use of similar libraries in the same use case, viral strain classification, is also recommended. Of particular interest is the possible implementation of a similar client-server classification system using the `Pyfhel` library developed by Ibarrond and Viand [92, 36], which is a Python wrapper for the SEAL and OpenFHE (currently in progress) FHE libraries, essentially exposing the underlying functionality of these FHE libraries in the high-level Python language, which allows for the more accessible implementation of FHE. Additionally, Pyfhel supports different FHE schemes (such as BFV and CKKS) as opposed to the singular TFHE scheme utilized by Concrete-ML. This could serve as a point of comparison in terms of running time, data precision and quality, ease of use, and overall accuracy and model performance.

IX. Bibliography

[1] Zama-AI, *Summary of the overall communications protocol to enable cloud deployment of machine learning services.* Zama-AI, Dec 2022.

[2] D. Tobin, "What is data privacy-and why is it important?," *Integrate.io*, May 2021.

[3] "What is data privacy? — privacy definition — Cloudflare," *Cloudflare.*

[4] "Data protection in the EU," *European Commission - European Commission*, Oct 2022.

[5] "Data privacy act primer," *National Privacy Commission*, Oct 2022.

[6] L. Song, H. Liu, F. S. Brinkman, E. Gill, E. J. Griffiths, W. W. Hsiao, S. Savić-Kallesøe, S. Moreira, G. Van Domselaar, M. H. Zawati, and et al., "Addressing privacy concerns in sharing viral sequences and minimum contextual data in a public repository during the covid-19 pandemic," *Frontiers in Genetics*, vol. 12, 2022.

[7] "Gordon v. Canada (Health), 2008 FC 258," *Office of the Privacy Commissioner of Canada*, Jun 2014.

[8] "Gordon v. Canada (Minister of Health), 2008 FC 258," *vLex.*

[9] L. Sweeney, "Simple demographics often identify people uniquely," *figshare*, Jun 2018.

[10] P. Golle, "Revisiting the uniqueness of simple demographics in the us population," *Proceedings of the 5th ACM workshop on Privacy in electronic society - WPES '06*, 2006.

[11] L. Rocher, J. M. Hendrickx, and Y.-A. de Montjoye, "Estimating the success of re-identifications in incomplete datasets using generative models," *Nature Communications*, vol. 10, no. 1, 2019.

[12] K. J. Wu, "A major clue to covid's origins is just out of reach," Mar 2023.

[13] "Sago statement on newly released sars-cov-2 metagenomics data from china cdc on gisaid," Mar 2023.

[14] B. Mole, "Genetic data links sars-cov-2 to raccoon dogs in china market, scientists say," Mar 2023.

[15] K. Cullinan, "High drama as scientists who may have found covid 'animal x' are kicked off data-sharing platformnbsp;," Mar 2023.

[16] S. LEHRER and P. H. RHEINSTEIN, "Human gene sequences in sars-cov-2 and other viruses," *In Vivo*, vol. 34, p. 1633–1636, Jun 2020.

[17] H. Li, X. Hong, L. Ding, S. Meng, R. Liao, Z. Jiang, and D. Liu, "Sequence similarity of sars-cov-2 and humans: Implications for sars-cov-2 detection," *Frontiers in Genetics*, vol. 13, Jul 2022.

[18] M. Lenharo, "Gisaid in crisis: Can the controversial covid genome database survive?," May 2023.

[19] R. Van Noorden, "Scientists call for fully open sharing of coronavirus genome data," Feb 2021.

[20] A. Aloufi, P. Hu, Y. Song, and K. Lauter, "Computing blindfolded on data homomorphically encrypted under multiple keys: A survey," *ACM Computing Surveys*, vol. 54, p. 1–37, Dec 2022.

[21] M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, and et al., "Security of homomorphic encryption - microsoft.com," Jan 2018.

[22] J. A. Cruz and R. Chua, "Secure remote genome-wide association studies using fully homomorphic encryption," *Theory and Practice of Computation Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2019)*, 2020.

[23] *Competition tasks - IDASH privacy security workshop 2021 - secure genome analysis competition*, 2021.

[24] D. N. Baker and B. Langmead, "Dashing: Fast and accurate genomic distances with hyperloglog," *Genome Biology*, vol. 20, no. 1, 2019.

[25] J. J. Sim, W. Zhou, F. M. Chan, M. S. Annamalai, X. Deng, B. H. Tan, and K. M. Aung, "CoVnita: An End-to-end Privacy-preserving Framework for SARS-CoV-2 Classification," *CoVnita: An End-to-end Privacy-preserving Framework for SARS-CoV-2 Classification PREPRINT (Version 1)*, Nov 2022.

[26] M. A. Remita, A. Halioui, A. A. Malick Diouara, B. Daigle, G. Kiani, and A. B. Diallo, "A machine learning approach for viral genome classification," *BMC Bioinformatics*, vol. 18, no. 1, 2017.

[27] J. Kim, K. Lee, R. Rupasinghe, S. Rezaei, B. Martínez-López, and X. Liu, "Applications of machine learning for the classification of porcine reproductive and respiratory syndrome virus sublineages using amino acid scores of ORF5 gene," *Frontiers in Veterinary Science*, vol. 8, 2021.

[28] A. Lopez-Rincon, A. Tonda, L. Mendoza-Maldonado, D. G. Mulders, R. Molenkamp, C. A. Perez-Romero, E. Claassen, J. Garssen, and A. D. Kran-

eveld, "Classification and specific primer design for accurate detection of SARS-COV-2 using Deep Learning," *Scientific Reports*, vol. 11, no. 1, 2021.

[29] G. B. Câmara, M. G. Coutinho, L. M. Silva, W. V. Gadelha, M. F. Torquato, R. d. Barbosa, and M. A. Fernandes, "Convolutional neural network applied to SARS-COV-2 sequence classification," *Sensors*, vol. 22, no. 15, p. 5730, 2022.

[30] D. S. Char, N. H. Shah, and D. Magnus, "Implementing machine learning in health care — addressing ethical challenges," *New England Journal of Medicine*, vol. 378, no. 11, p. 981–983, 2018.

[31] D. S. Char, M. D. Abràmoff, and C. Feudtner, "Identifying ethical considerations for Machine Learning Healthcare Applications," *The American Journal of Bioethics*, vol. 20, no. 11, p. 7–17, 2020.

[32] J. Li, X. Kuang, S. Lin, X. Ma, and Y. Tang, "Privacy preservation for machine learning training and classification based on homomorphic encryption schemes," *Information Sciences*, vol. 526, p. 166–179, Jul 2020.

[33] A. Akavia, B. Galili, H. Shaul, M. Weiss, and Z. Yakhini, "Efficient privacy-preserving viral strain classification via K-Mer signatures and FHE," *IBM Research Publications*, May 2022.

[34] M. Templ and M. Sariyar, "A systematic overview on methods to protect sensitive data provided for various analyses," *SpringerLink*, Aug 2022.

[35] A. Benaissa, B. Retiat, B. Cebere, and A. E. Belfedhal, "TenSEAL: A library for encrypted tensor operations using homomorphic encryption," *arXiv.org*, Apr 2021.

[36] A. Ibarrondo and A. Viand, "Pyfhel: PYthon For Homomorphic Encryption Libraries," *Proceedings of the 9th on Workshop on Encrypted Computing Applied Homomorphic Cryptography*, p. 11–16, Nov 2021.

[37] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, p. 120–126, 1978.

[38] Microsoft, "Microsoft/SEAL," *GitHub*, Mar 2022. Available at https://github.com/microsoft/SEAL.

[39] Openfheorg, "Openfheorg/openfhe-development," *GitHub*, Nov 2022. Available at https://github.com/openfheorg/openfhe-development.

[40] Zama-AI, "Concrete-ML," *GitHub*, Nov 2022. Available at https://github.com/zama-ai/concrete-ml.

[41] D. Murik, A. Bitar, and S. Martinelli, "IBM/HElayers: IBM HElayers Homomorphic Encryption SDK for C++ and Python," *GitHub*, Nov 2022. Available at https://github.com/IBM/helayers.

[42] T.-T. Kuo, T. Bath, S. Ma, N. Pattengale, M. Yang, Y. Cao, C. M. Hudson, J. Kim, K. Post, L. Xiong, and et al., "Benchmarking blockchain-based gene-drug interaction data sharing methods: A case study from the iDASH 2019 secure genome analysis competition blockchain track," *International Journal of Medical Informatics*, vol. 154, p. 104559, 2021.

[43] A. K. Ladisla and R. B. Chua, "Theory and practice of computation," *Theory and Practice of Computation Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2018)*, 2019.

[44] E. Sarkar, E. Chielle, G. Gursoy, O. Mazonka, M. Gerstein, and M. Maniatakos, "Fast and scalable private genotype imputation using machine learning and partially homomorphic encryption," *IEEE Access*, vol. 9, p. 93097–93110, 2021.

[45] D. Froelicher, J. R. Troncoso-Pastoriza, J. L. Raisaro, M. A. Cuendet, J. S. Sousa, H. Cho, B. Berger, J. Fellay, and J.-P. Hubaux, "Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption," *Nature Communications*, vol. 12, no. 1, 2021.

[46] K. Lauter, S. Kannepalli, K. Laine, and R. C. Moreno, "Password Monitor: Safeguarding Passwords in Microsoft Edge," *Microsoft Research*, Jan 2021.

[47] IBM *Efficient Privacy-Preserving Viral Strain Classification via k-mer Signatures and FHE (Poster)*, 2022. Available at https://drive.google.com/file/d/1hAcn_DZPQ5KA837JVoMkMwg__W4FtESA/view.

[48] H. Shaul, B. Galili, A. Akavia, M. Weiss, and Z. Yakhini, "IBM homomorphic encryption: A DASHing solution for healthcare data privacy," *IBM Developer*, Feb 2022.

[49] A. Akavia, B. Galili, H. Shaul, M. Weiss, and Z. Yakhini, "Efficient privacy-preserving viral strain classification via K-Mer signatures and FHE," *Cryptology ePrint Archive*, Jan 2023.

[50] M. Creeger and C. Inc., "The rise of fully homomorphic encryption," *The Rise of Fully Homomorphic Encryption - ACM Queue*, Sep 2022.

[51] O. Regev, "Lattice-based cryptography," *Lecture Notes in Computer Science*, p. 131–141, 2006.

[52] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, p. 169–180, 1978.

[53] C. Gentry, "Fully homomorphic encryption using ideal lattices," *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*, 2009.

[54] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *Advances in Cryptology – EUROCRYPT 2011*, p. 129–148, 2011.

[55] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," *https://eprint.iacr.org/*, 2010.

[56] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," *whitepaper.zama.ai*, 2020.

[57] Zama-AI, "Linear models," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/built-in-models/linear.

[58] Zama-AI, "Tree-based models," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/built-in-models/tree.

[59] Zama-AI, "Neural networks," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/built-in-models/neural-networks.

[60] Zama-AI, "Key concepts," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/getting-started/concepts.

[61] Zama-AI, "Quantization," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/advanced-topics/quantization.

[62] Zama-AI, "Compilation," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/advanced-topics/compilation.

[63] Zama-AI, "Quick start," *Concrete ML*. Available at https://docs.zama.ai/concrete-numpy/getting-started/quick_start.

[64] Zama-AI, "Production deployment," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/advanced-topics/client_server.

[65] Zama-AI, "Concrete.ml.deployment.fhe_client_server," *Concrete ML*. Available at https://docs.zama.ai/concrete-ml/developer-guide/api/concrete.ml.deployment.fhe_client_server.

[66] Zama-AI, "Concrete-ml/clientserver.ipynb at release/0.5.x · zama-ai/concrete-ml," *GitHub*, Sep 2022. Available at https://github.com/zama-ai/concrete-ml/blob/release/0.5.x/docs/advanced_examples/ClientServer.ipynb.

[67] P. Flajolet, Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," *Discrete Mathematics amp;amp; Theoretical Computer Science*, vol. DMTCS Proceedings vol. AH,..., no. Proceedings, 2007.

[68] IBM, "What is logistic regression?," *IBM*. Available at https://www.ibm.com/topics/logistic-regression.

[69] A. Pradhan, S. Prabhu, K. Chadaga, S. Sengupta, and G. Nath, "Supervised learning models for the preliminary detection of COVID-19 in patients using demographic and epidemiological parameters," *MDPI*, Jul 2022.

[70] A. C. Chang, "Machine and deep learning," *Intelligence-Based Medicine*, Aug 2020.

[71] M. A. Zeller, Z. W. Arendsee, G. J. Smith, and T. K. Anderson, "Classlog: Logistic regression for the classification of genetic sequences," *bioRxiv*, Jan 2022.

[72] S. Khare, C. Gurry, L. Freitas, M. B. Schultz, G. Bach, A. Diallo, N. Akite, J. Ho, R. TC Lee, W. Yeo, and et al., "GISAID's role in pandemic response," *China CDC Weekly*, vol. 3, no. 49, p. 1049–1051, 2021.

[73] S. Elbe and G. Buckland-Merrett, "Data, disease and diplomacy: GISAID's innovative contribution to global health," *Global Challenges*, vol. 1, no. 1, p. 33–46, 2017.

[74] Y. Shu and J. McCauley, "GISAID: Global initiative on sharing all influenza data – from vision to reality," *Eurosurveillance*, vol. 22, no. 13, 2017.

[75] W. Wirth and S. Duchene, "WYTAMMA/GISAIDR: Programmatically interact with the GISAID database.," 2022.

[76] National Center for Biotechnology Information, "FASTA Format for Nucleotide Sequences," Feb 2021. Available at https://www.ncbi.nlm.nih.gov/genbank/fastaformat.

[77] 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*, vol. 12, pp. 2825–2830, 2011.

[78] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[79] J. K. E. Yee, "Brief introduction to four categories of feature selection techniques," Jan 2021.

[80] A. Bajaj, "Performance metrics in machine learning [complete guide]," May 2023.

[81] A. Mishra, "Metrics to evaluate your machine learning algorithm," Feb 2018.

[82] E. Zvornicanin, "Accuracy vs auc in machine learning," May 2023.

[83] J. Czakon, "F1 score vs roc auc vs accuracy vs pr auc: Which evaluation metric should you choose?," May 2023.

[84] J. Brownlee, "Roc curves and precision-recall curves for imbalanced classification," Sep 2020.

[85] N. Tyagi, "What is confusion matrix?," Mar 2021.

[86] R. Vidiyala, "Confusion matrix in a nutshell," May 2020.

[87] D. Corvoysier, "A brief introduction to machine learning models quantization," May 2023. Available at https://www.kaizou.org/2023/05/machine-learning-quantization-introduction.html.

[88] L. Mao, "Quantization for neural networks," Feb 2023. Available at https://leimao.github.io/article/Neural-Networks-Quantization/#Quantization.

[89] "google/gemmlowp: low-precision matrix multiplication," Sep 2022. Available at https://github.com/google/gemmlowp/tree/master.

[90] J. Frery, A. Stoian, R. Bredehoft, L. Montero, C. Kherfallah, B. Chevallier-Mames, and A. Meyre, "Privacy-preserving tree-based inference with fully homomorphic encryption," Mar 2023.

[91] E. B. Barker, W. C. Barker, W. E. Burr, W. T. Polk, and M. E. Smid, "Sp 800-57. recommendation for key management, part 1: General (revised)," tech. rep., Gaithersburg, MD, USA, 2007.

[92] Ibarrond, "Ibarrond/pyfhel: Python for homomorphic encryption libraries, perform encrypted computations such as sum, mult, scalar product or matrix multiplication in python, with numpy compatibility. uses seal/palisade as backends, implemented using cython.," *GitHub*, Nov 2022. Available at https://github.com/ibarrond/Pyfhel.

X. Appendix

A. Source Code

```
# %%
from numpy import mean
from numpy import std
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
from sklearn.metrics import accuracy_score, roc_auc_score, recall_score, confusion_matrix, ConfusionMatrixDisplay
from concrete.ml.sklearn import LogisticRegression, LinearRegression
from concrete.ml.sklearn.svm import LinearSVC
from sklearn.svm import LinearSVC as skSVC
from concrete.ml.sklearn.rf import RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier as skRF
from sklearn.linear_model import LogisticRegression as skLR
from sklearn.linear_model import LinearRegression as skLinear
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import cross_val_score
from sklearn.feature_selection import VarianceThreshold
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import time, numpy
import pandas as pd

start_time = time.time()

dataset = pd.read_csv("AFHE DATASET (05-18-2023).csv")

feature_cols = [c for c in dataset.columns[2:]]

x = dataset.loc[:,feature_cols].values #must be floats
y = dataset.loc[:,'Lineage'].values #must be integers

# Preprocessing with labels for the lineage
le = preprocessing.LabelEncoder()
y = le.fit_transform(y)
print(le.classes_)

x = x.astype(float)

print("Shape of x: ", x.shape)
print("Shape of y:", y.shape)

print(f"Running time is {time.time() - start_time} seconds")

# %%
# Feature Selection

start_time = time.time()

print("\nUsing K best features feature selection...")
print("Shape of x before selection: ", x.shape)
selector = SelectKBest(chi2, k=20)
x = selector.fit_transform(x, y)
col_idxs = selector.get_support(indices=True)
print("Shape of x after selection: ", x.shape)

print(f"Running time is {time.time() - start_time} seconds")

# %%
# Retrieve train and test sets
start_time = time.time()
X_train, X_test, y_train, y_test = train_test_split(
x, y, test_size=.20)
print(f"Test set size: {X_test.shape}")
print(f"Running time is {time.time() - start_time} seconds")

# %%
#NOTE WE HAVE A MULTICLASS BUT NOT MULTILABEL PROBLEM. only one label selected from multiple classes is assigned
print("Getting metrics for scikit-learn model (Plaintext)...")
skmodel = skLR(C=1)

start_time = time.time()
skmodel.fit(X_train,y_train)
print(f"Training time is {time.time() - start_time} seconds")

start_time = time.time()
y_pred_sklearn = skmodel.predict(X_test)
print(f"Prediction time is {time.time() - start_time} seconds")
print(f"Accuracy: {skmodel.score(X_test,y_test)*100}%")
print(f"Macro-averaged ROC AUC Score: {roc_auc_score(y, skmodel.predict_proba(x), multi_class='ovr')}")
```

```
print(f"Recall: {recall_score(y_test, y_pred_sklearn, average='weighted')*100}%")
sklearn_cm_display = ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred_sklearn), display_labels=le.classes_)
sklearn_cm_display.plot()
plt.show()

# %%
print("Getting metrics for Concrete-ML model (Quantized Plaintext)...")
model = LogisticRegression(C=1)

# Fit the model
start_time = time.time()
model.fit(X_train, y_train)
print(f"Training time is {time.time() - start_time} seconds")

# Run the predictions on non-encrypted data as a reference
start_time = time.time()
y_pred_clear = model.predict(X_test)
print(f"Prediction time is {time.time() - start_time} seconds")
print(f"Accuracy: {model.score(X_test,y_test) * 100}%")
print(f"Macro-averaged ROC AUC Score: {roc_auc_score(y, model.predict_proba(x), multi_class='ovr')}")
print(f"Recall: {recall_score(y_test, y_pred_clear, average='weighted')*100}%")
concrete_plain_display = ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred_clear), display_labels=le.
    classes_)
concrete_plain_display.plot()
plt.show()

#%%
print("Getting metrics for Concrete-ML model (FHE)...")

start_time = time.time()
print("Compiling the quantized model...")
model.compile(x)
print("Model compiled!")
print(f"Compilation time is {time.time() - start_time} seconds")

start_time = time.time()
y_pred_fhe = model.predict(X_test, fhe="execute")
print(f"Prediction time is {time.time() - start_time} seconds")
print(f"Accuracy: {accuracy_score(y_test, y_pred_fhe) * 100}%")
print(f"Macro-averaged ROC AUC Score: {roc_auc_score(y, model.predict_proba(x), multi_class='ovr')}")
print(f"Recall: {recall_score(y_test, y_pred_fhe, average='weighted')*100}%")
print(f"Comparison (FHE vs Plaintext): {int((y_pred_fhe == y_pred_sklearn).sum()/len(y_pred_fhe)*100)}% similar")
print(f"Comparison (FHE vs Quantized Plaintext): {int((y_pred_fhe == y_pred_clear).sum()/len(y_pred_fhe)*100)}%
    similar")
concrete_fhe_display = ConfusionMatrixDisplay(confusion_matrix(y_test, y_pred_fhe), display_labels=le.classes_)
concrete_fhe_display.plot()
plt.show()

# %%
print(f"Sklearn Accuracy (Linear Reg, RF, SVC):")

#SKLEARN LINEAR REGRESSION
skmodel2 = skLinear()
skmodel2.fit(X_train,y_train)
skmodel2.predict(X_test)
print("Sklearn Linear Regression Accuracy: ", skmodel2.score(X_test,y_test)*100,"%")

#SKLEARN RANDOM FOREST
skmodel3 = skRF()
skmodel3.fit(X_train, y_train)
skmodel3.predict(X_test)
print("Sklearn Random Forest Accuracy: ",skmodel3.score(X_test,y_test)*100,"%")

#SKLEARN SVC
skmodel4 = skSVC()
skmodel4.fit(X_train, y_train)
skmodel4.predict(X_test)
print("Sklearn SVC Accuracy: ",skmodel4.score(X_test,y_test)*100,"%")

print(f"\nConcrete-ML Accuracy (Linear Reg, RF, SVC):")

model2 = LinearRegression()
model2.fit(X_train,y_train)
model2.predict(X_test)
print("Concrete-ML Linear Regression Accuracy: ", model2.score(X_test,y_test)*100,"%")

model3 = RandomForestClassifier()
model3.fit(X_train, y_train)
model3.predict(X_test)
print("Concrete-ML Random Forest Accuracy: ",model3.score(X_test,y_test)*100,"%")

model4 = LinearSVC()
model4.fit(X_train, y_train)
model4.predict(X_test)
print("Concrete-ML SVC Accuracy: ",model4.score(X_test,y_test)*100,"%")

model2.compile(x)
model3.compile(x)
model4.compile(x)
model2.predict(X_test)
model3.predict(X_test)
model4.predict(X_test)
```

```
172
173 print("\nFHE Concrete-ML Linear Regression Accuracy: ", model2.score(X_test,y_test)*100,"%")
174 print("FHE Concrete-ML Random Forest Accuracy: ",model3.score(X_test,y_test)*100,"%")
175 print("FHE Concrete-ML SVC Accuracy: ",model4.score(X_test,y_test)*100,"%")
176
177 # %%
178 import json
179
180 #Attempting to save the model
181 from concrete.ml.deployment import FHEModelClient, FHEModelDev, FHEModelServer
182
183 start_time = time.time()
184 fhemodel_dev = FHEModelDev("./concrete-covid-classifier", model)
185 fhemodel_dev.save()
186 print(f"Running time for saving the FHE model is {time.time() - start_time} seconds")
187
188 for col in col_idxs:
189     print(feature_cols[col])
190
191 for c in le.classes_:
192     print(c)
193
194 start_time = time.time()
195 with open("features_and_classes.txt", "w") as f:
196     classes_list = list(le.classes_)
197     temp_dict = {"features":[feature_cols[col] for col in col_idxs], "classes":{classes_list.index(x):x for x in
       classes_list}}
198
199     f.write(json.dumps(temp_dict))
200 print(f"Running time for saving the features and classes is {time.time() - start_time} seconds")
```

Listing 3: Source code for the logistic regression model training script

```
1  import shutil
2  from django.http import FileResponse
3  from django.shortcuts import render, HttpResponse, HttpResponseRedirect
4  from concreteClassifierApp.settings import BASE_DIR
5  import os
6  import io, zipfile, requests
7  import subprocess
8  from pandas import DataFrame as pd
9  from pandas import read_csv
10 from concrete.ml.deployment import FHEModelServer
11
12 # Create your views here.
13 def index(request):
14
15     return render(request, 'index.html', context={'classes_list':{0: 'B.1.1.529 (Omicron)', 1: 'B.1.617.2 (Delta)
       ', 2: 'B.1.621 (Mu)', 3: 'C.37 (Lambda)'}})
16
17 def start_classification(request):
18
19     clean_predictions_folder()
20
21     count = 0
22     model_path = os.path.join(BASE_DIR, "Compiled Model")
23     keys_path = os.path.join(BASE_DIR, "classifier/keys")
24     keys_file = request.FILES['keys_file']
25     pred_dir = os.path.join(BASE_DIR, "classifier/predictions")
26
27     data = request.FILES['inputs'].read().strip()
28
29     enc_file_list = []
30
31     print(f"Data received from client is {data[:200]}")
32     count += 1
33     serialized_evaluation_keys = keys_file.read()
34     encrypted_prediction = FHEModelServer(model_path).run(data, serialized_evaluation_keys)
35     pred_file_name = f"encrypted_prediction_{count}.enc"
36     pred_file_path = os.path.join(pred_dir, pred_file_name)
37     with open(pred_file_path, "wb") as f:
38         f.write(encrypted_prediction)
39
40     #send all predictions as a zip file to client
41     enc_file_list.append(pred_file_path)
42
43     zipfile = create_zip(enc_file_list)
44
45     return zipfile
46
47 def create_zip(file_list):
48     count = 0
49     zip_filename = os.path.join(BASE_DIR, "classifier/predictions/enc_predictions.zip")
50     zip_download_name = "enc_predictions.zip"
51     buffer = io.BytesIO()
52     zip_file = zipfile.ZipFile(buffer, 'w')
53     #zip_file = zipfile.ZipFile(zip_filename, 'w')
54
55     for filename in file_list:
56         count += 1
```

```
        with open(filename, "rb") as file_read:
            zip_file.write(filename, f"encrypted_prediction_{count}.enc")
    zip_file.close()

    #craft download response
    resp = HttpResponse(buffer.getvalue(), content_type = "application/force-download")
    resp['Content-Disposition'] = f'attachment; filename={zip_download_name}'

    return resp

def clean_predictions_folder():
    pred_dir = os.path.join(BASE_DIR, f"classifier/predictions")

    if(os.listdir(pred_dir)):
        for f in os.listdir(pred_dir):
            os.remove(os.path.join(pred_dir, f))
```

Listing 4: Source code for the server-side classification function

```
from django.contrib import admin
from django.urls import path
from . import views

app_name = "classifier"

urlpatterns = [
    path('', views.index, name='index'),
    path('start_classification', views.start_classification, name='start_classification'),
]
```

Listing 5: Source code for the server-side URL settings

```
{% load static %}

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <meta name="description" content="">
    <meta name="author" content="">
    <link rel="icon" href="/docs/4.0/assets/img/favicons/favicon.ico">

    <title>FHE-enabled SARS-CoV-2 Sequence Classifier (Server-Side)</title>

    <link rel="canonical" href="https://getbootstrap.com/docs/4.0/examples/jumbotron/">

    <!-- Bootstrap core CSS -->
    <link href="{% static 'css/bootstrap.min.css' %}" rel="stylesheet">

    <!-- Custom styles for this template -->
    <link href="{% static 'css/jumbotron.css' %}" rel="stylesheet">
  </head>

  <body>

    <form action="{% url 'classifier:start_classification' %}" method="post">
      {% csrf_token %}
      <div class="form-group mt-3" hidden>
          <label class="mr-2">Upload your encrypted inputs:</label>
          <input type="file" name="file">
      </div>
      <div class="form-group mt-3" hidden>
          <label class="mr-2">Upload your evaluation keys (.ekl):</label>
          <input type="file" name="keys_file">
      </div>
    </form>

    <main role="main">

      <!-- Main jumbotron for a primary marketing message or call to action -->
      <div class="jumbotron">
        <div class="container">
          <h1 class="display-3">FHE-enabled SARS-CoV-2 Sequence Classifier</h1>
          <p>This is the homepage for the server-side application of an FHE-enabled SARS-CoV-2 classification
      system. This site hosts links to the Github repository of the application, and the download link for the
      client GUI app (requirements are listed in requirements.txt in the Github repository).</p>
          <p><a class="btn btn-primary btn-lg" href="https://github.com/bjorgkav/concreteml-covid-classifier"
      role="button" target="_blank">Access the Github repository &raquo;</a></p>
        </div>
      </div>

      <div class="container">
        <!-- Example row of columns -->
        <div class="row">
          <div class="col-md-4">
            <h2>Technologies Used</h2>
            <ul>
```

```
            <li><a href="https://github.com/zama-ai/concrete-ml" target="_blank" rel="noopener noreferrer">Concrete-ML v1.0.3</a></li>
            <li><a href="https://www.djangoproject.com/" target="_blank" rel="noopener noreferrer">Django Framework v4.2.1</a></li>
            <li><a href="https://github.com/dnbaker/dashing" target="_blank" rel="noopener noreferrer">Dashing by dnbaker</a></li>
            <li><a href="https://github.com/TomSchimansky/CustomTkinter" target="_blank" rel="noopener noreferrer">CustomTkinter v5.1.3</a></li>
            <li><a href="https://pandas.pydata.org/" target="_blank" rel="noopener noreferrer">Pandas v2.0.1</a></li>
          </ul>
          <p><a class="btn btn-secondary" href="https://github.com/bjorgkav/concreteml-covid-classifier/blob/main/requirements.txt" target="_blank" role="button">Show more &raquo;</a></p>
        </div>
        <div class="col-md-4">
          <h2>Client-side application</h2>
          <p>This classification system is intended to be used with the client-side GUI application, which can be accessed here (see requirements.txt for the required packages):</p>
          <p><a class="btn btn-secondary" href="https://github.com/bjorgkav/concreteml-covid-classifier/blob/main/client/GUI/Client_GUI_App.py" role="button">Download the Client-side GUI Application &raquo;</a></p>
        </div>
        <div class="col-md-4">
          <h2>SARS-CoV-2 Strains Currently Supported</h2>
          <ul>
            {% for value in classes_list.values %}
            <li>{{ value }}</li>
            {% endfor %}
          </ul>
        </div>
      </div>

      <hr>

    </div> <!-- /container -->

  </main>

  <footer class="container">
    <p>Website developed by Johann Benjamin P. Vivas (last updated June 3, 2023)</p>
  </footer>

  <!-- Bootstrap core JavaScript
  ================================================== -->
  <!-- Placed at the end of the document so the pages load faster -->
  <script src="js/vendor/popper.min.js"></script>
  <script src="js/bootstrap.min.js"></script>
 </body>
</html>
```

Listing 6: Source code for the server-side web application homepage

```
#!/usr/bin/python3
import shutil, subprocess, zipfile, requests
from customtkinter import (
    CTk,
    CTkButton,
    CTkEntry,
    CTkFont,
    CTkFrame,
    CTkLabel,
    IntVar,
    StringVar,
    CTkTextbox,
    set_appearance_mode,
    set_default_color_theme)

from tkinter import filedialog as fd
from tkinter import END, INSERT
from datetime import date, datetime
from concrete.ml.deployment import FHEModelClient
import os, requests, stat, numpy, json, traceback
from pandas import DataFrame as pd
from pandas import read_csv
from sklearn.preprocessing import LabelEncoder

#region class
class ClientTkinterUiDesignApp:
    def __init__(self, master=None):
        # initialize FHEModelClient and output dictionary
        self.fhe_model_client = FHEModelClient(os.path.dirname(__file__), os.path.join(os.path.dirname(__file__), "keys"))
        self.data_dictionary = {}

        # create required folders if not exists
        this_folder = os.path.dirname(__file__)

        required_folder_names = ["fastas", "keys", "predictions"]

        for name in required_folder_names:
            if not os.path.exists(os.path.join(this_folder, f"{name}")):
```

```
                os.mkdir(os.path.join(this_folder, f"{name}"))

        # build ui
        self.root = CTk(None)
        self.root.configure(padx=60, pady=10)
        set_appearance_mode("dark")
        set_default_color_theme("dark-blue")
        self.root.geometry("800x900")
        self.root.resizable(True, True)
        self.root.title(
            "FHE-Enabled SARS-CoV-2 Classifier System (Client-side)")
        self.encrypt_name_var = StringVar()
        self.decrypt_name_var = StringVar()
        self.title = CTkLabel(self.root)
        self.title.configure(
            bg_color="#035690",
            font=CTkFont(
                "roboto",
                20,
                None,
                "roman",
                False,
                False),
            justify="center",
            text='FHE-Enabled SARS-CoV-2 Classifier System (Client-side)')
        self.title.pack(anchor="n", fill="x", ipady=10, side="top")
        self.description_frame = CTkFrame(self.root)
        self.about_label = CTkLabel(self.description_frame)
        self.about_label.configure(
            font=CTkFont(
                "roboto",
                24,
                None,
                "roman",
                False,
                False),
            text='About')
        self.about_label.pack(expand=False, fill="both", pady=10, side="top")
        self.description_label = CTkLabel(self.description_frame)
        self.description_label.configure(
            justify="left",
            text='This tool allows clients to convert their FASTA files to a numerical format and encrypt them
    for classification \non the server-side application. \n\n0n startup, this app automatically downloads the
    required files and scripts for operations \n(est. size 50 MB, internet connection required).')
        self.description_label.pack(expand=False, fill="x", side="top")
        self.description_frame.pack(
            fill="both", ipady=10, padx=20, pady=20, side="top")
        self.dashing_frame = CTkFrame(self.root)
        self.dashing_label = CTkLabel(self.dashing_frame)
        self.dashing_label.configure(
            anchor="w",
            justify="left",
            text='Enter your fasta file filepath for processing:')
        self.dashing_label.grid(column=0, padx=10, pady=10, row=0, sticky="nw")
        self.dashing_filename = CTkEntry(self.dashing_frame)
        self.dashing_name_var = StringVar()
        self.dashing_filename.configure(
            exportselection=False,
            justify="left",
            state="disabled",
            takefocus=False,
            textvariable=self.dashing_name_var,
            width=460)
        self.dashing_filename.grid(column=0, padx=10, row=1)
        self.dashing_browse = CTkButton(self.dashing_frame, hover=True)
        self.dashing_browse.configure(hover_color="#299cd9", text='Browse...')
        self.dashing_browse.grid(column=2, padx=10, row=1)
        self.dashing_browse.configure(command=self.getDashingInput)
        self.dashing_begin = CTkButton(self.dashing_frame)
        self.dashing_begin.configure(
            hover_color="#299cd9",
            text='Submit for FHE Classification',
            width=300)
        self.dashing_begin.grid(column=0, columnspan=3, pady=10, row=2)
        self.dashing_begin.configure(command=self.processData)
        self.dashing_frame.pack(
            anchor="w",
            fill="x",
            padx=20,
            pady=10,
            side="top")
        ctkframe2 = CTkFrame(self.root)
        self.app_output_label = CTkLabel(ctkframe2)
        self.app_output_label.configure(text='Output Window')
        self.app_output_label.pack(side="top")
        self.app_output = CTkTextbox(ctkframe2)
        self.app_output.configure(height=75, state="disabled")
        _text_ = 'App activity will be displayed here.'
        self.app_output.configure(state="normal")
        self.app_output.insert("0.0", _text_)
        self.app_output.configure(state="disabled")
        self.app_output.pack(expand=True, fill="both", padx=10, pady=10)
```

```
129         self.app_pred_history = CTkTextbox(ctkframe2)
130         self.app_pred_history.configure(height=75, state="disabled")
131         _text_ = 'Prediction History:\n'
132         self.app_pred_history.configure(state="normal")
133         self.app_pred_history.insert("0.0", _text_)
134         self.app_pred_history.configure(state="disabled")
135         self.app_pred_history.pack(expand=True, fill="both", padx=10, pady=10)
136         ctkframe2.pack(expand=True, fill="both", padx=20, pady=10, side="top")
137 
138         # Main widget
139         self.mainwindow = self.root
140 
141     def run(self):
142         self.mainwindow.mainloop()
143 
144     def writeOutput(self, string, delete_switch = False):
145         """Function for writing argument 'string' to the app's output window. Set argument 'delete_switch' to
        True to clear the window before printing."""
146         self.app_output.configure(state="normal")
147         if(delete_switch):
148             self.app_output.delete("1.0", END) #tk.END
149             self.app_output.insert("0.0", f"{string}\n\n")
150         else:
151             self.app_output.insert(INSERT, f"{string}\n\n")
152         self.app_output.see(END)
153         self.app_output.configure(state="disabled")
154 
155     def writePredOutput(self, string, delete_switch = False):
156         """Function for writing argument 'string' to the app's prediction output window. Set argument '
        delete_switch' to True to clear the window before printing."""
157         self.app_pred_history.configure(state="normal")
158         if(delete_switch):
159             self.app_pred_history.delete("1.0", END) #tk.END
160             self.app_pred_history.insert("0.0", f"{string}\n")
161         else:
162             self.app_pred_history.insert(INSERT, f"{string}\n")
163         self.app_pred_history.see(END)
164         self.app_pred_history.configure(state="disabled")
165 
166     def get_size(self, file_path, unit='bytes'):
167         file_size = os.path.getsize(file_path)
168         exponents_map = {'bytes': 0, 'kb': 1, 'mb': 2, 'gb': 3}
169         if unit not in exponents_map:
170             raise ValueError("Must select from \
171             ['bytes', 'kb', 'mb', 'gb']")
172         else:
173             size = file_size / 1024 ** exponents_map[unit]
174             return round(size, 3)
175 
176     def processData(self):
177         self.getFeaturesAndClasses()
178         self.writeOutput("", True)
179         self.beginDashing()
180         self.beginEncryption()
181         self.beginDecryption()
182 
183     def getDashingInput(self):
184         dashing_filename = fd.askopenfilename()
185         self.dashing_name_var.set(dashing_filename)
186 
187     def getEncryptInput(self):
188         encrypt_filename = fd.askopenfilename()
189         self.encrypt_name_var.set(encrypt_filename)
190 
191     def getDecryptInput(self):
192         decrypt_filename = fd.askopenfilename()
193         self.decrypt_name_var.set(decrypt_filename)
194 
195     def beginDashing(self):
196         """Function to begin dashing the user's input. Expects the 'self.dashing_name_var' to point to a .fasta
        file or zip file. Outputs a CSV file for encryption."""
197         try:
198             if(os.listdir(os.path.join(os.path.dirname(__file__), "fastas"))):
199                 for f in os.listdir(os.path.join(os.path.dirname(__file__), "fastas")):
200                     os.remove(os.path.join(os.path.join(os.path.dirname(__file__), "fastas"), f))
201 
202             self.writeOutput("Beginning Dashing...", False)
203 
204             filename = self.dashing_name_var.get()
205             if filename.endswith(".fasta"):
206                 first_line, sequence, id = self.readTruncateSequence(filename)
207                 self.writeFasta(id, first_line, sequence)
208                 self.useDashing()
209 
210                 self.writeOutput("Writing dashed sequences to output.csv in the current directory...")
211 
212                 dashing_output = os.path.join(os.path.dirname(__file__), f"output.csv")
213                 self.dropColumns(dashing_output)
214                 self.encrypt_name_var.set(dashing_output)
215 
216                 self.writeOutput("Dashing Completed!")
217             else:
```

```
                raise Exception("Invalid file type: supported file types include .fasta, .zip")
        except Exception as e:
            self.writeOutput(f"Error: {traceback.format_exc()}")

    def beginEncryption(self, check_size = False):
        """Function to begin the encryption of the user's dashed SARS-CoV-2 sequences. Expects 'self.
        encrypt_name_var' to point to the CSV file containing dashed sequences. Outputs a text file and .ekl file
        for the encrypted inputs and serialized evaluation keys respectively in this app's directory."""
        try:

            for f in os.listdir(os.path.dirname(__file__)):
                if f.split("/")[-1] in ["encrypted_input.txt", "serialized_evaluation_keys.ekl"]:
                    os.remove(f)

            if(not self.encrypt_name_var.get().endswith(".csv")):
                raise Exception("Invalid file type. Only .csv files are supported.")

            self.writeOutput("Generating Keys...", False)

            self.generateKeys()

            self.writeOutput("Key generation complete! Key files written to folder inside 'keys' directory.")

            self.writeOutput("Beginning encryption...")

            dashing_output = self.encrypt_name_var.get()
            df = read_csv(dashing_output)
            arr_no_id = df.drop(columns=['Accession ID']).to_numpy(dtype="uint16")

            #encrypted rows for input to server
            encrypted_rows = []

            #encrypted dictionary for outputs
            count = 0
            for id in df['Accession ID']:
                self.data_dictionary[count] = {'id':id, 'result':''}

            for row in range(0, arr_no_id.shape[0]):
                self.encrypted_id = self.data_dictionary[row]['id']
                clear_input = arr_no_id[[row],:]

                encrypted_input = self.fhe_model_client.quantize_encrypt_serialize(clear_input)
                self.writeOutput(f"New row encrypted of {type(encrypted_input)}; adding to list of encrypted
        values...")
                encrypted_rows.append(encrypted_input)

            self.encrypted_rows = encrypted_rows

            self.writeOutput(f"Encryption complete! Here are the first 15 character of your encrypted output:\n{
        encrypted_rows[0][0:16]}")

            enc_filename = self.saveEncryptedOutput(self.encrypted_id)

            self.writeOutput("Saved encrypted inputs and key files to 'encrypted_input.txt' and '
        serialized_evaluation_keys.ekl' respectively.\nPlease do not move these files until after prediction.")

            app_url = "http://localhost:8000"

            client = requests.session()

            client.get(app_url)

            predictions_zip_name = self.sendEncryptRequestToServer(enc_filename, client=client)

            self.decrypt_name_var.set(predictions_zip_name)

        except Exception as e:
            self.writeOutput(f"Error: {traceback.format_exc()}")

    def sendEncryptRequestToServer(self, encrypt_filename, client):
        """Sends 'encrypted_input.txt' and 'serialized_evaluation_keys.ekl' (expected to be located in the same
        directory as the app) to the server-side app through the Python requests library. URL is set to localhost
        :8000 in development."""

        app_url = "http://localhost:8000"

        if 'csrftoken' in client.cookies:
            # Django 1.6 and up
            csrftoken = client.cookies['csrftoken']
        else:
            # older versions
            csrftoken = client.cookies['csrf']

        eval_keys_file = open('serialized_evaluation_keys.ekl', "rb")
        inputs_file = open(encrypt_filename, "rb")
        request_data = dict(csrfmiddlewaretoken=csrftoken)
        request_files = dict(inputs=inputs_file, keys_file=eval_keys_file)

        self.writeOutput("Sending encrypted inputs and keys to server for classification...")

        self.writeOutput("Waiting for server's response...")

```

```
        #send the above files to "localhost:8000/{function_name}"
        request_output = client.post(f"{app_url}/start_classification", data = request_data, files=request_files, headers=dict(Referer=app_url), )

        if request_output.ok:
            self.writeOutput(f"Response Code {request_output.status_code}: Classification completed!")

            with open(os.path.join(os.path.dirname(__file__), "predictions/enc_predictions.zip"), "wb") as z:
                z.write(request_output.content)

        return os.path.join(os.path.dirname(__file__), "predictions/enc_predictions.zip")

    def generateKeys(self):
        model_dir = os.path.dirname(__file__)
        key_dir = os.path.join(os.path.dirname(__file__), "keys")

        if(os.listdir(key_dir)):
            for f in os.listdir(key_dir):
                shutil.rmtree(os.path.join(key_dir, f))

        fhemodel_client = FHEModelClient(model_dir, key_dir=key_dir)

        # The client first needs to create the private and evaluation keys.
        fhemodel_client.generate_private_and_evaluation_keys()

        # Get the serialized evaluation keys
        self.serialized_evaluation_keys = fhemodel_client.get_serialized_evaluation_keys()

    def saveEncryptedOutput(self, id, show_key_sizes = True):
        """Saves encrypted rows as a text file to send to the server for classification. Also shows key sizes for comparison by default."""
        filename = f"{id}_encrypted_input.txt"
        with open(os.path.join(os.path.dirname(__file__), filename), "wb") as enc_file:
            for line in self.encrypted_rows:
                enc_file.write(line)

        with open(os.path.join(os.path.dirname(__file__), r'serialized_evaluation_keys.ekl'), "wb") as f:
            f.write(self.serialized_evaluation_keys)

        if show_key_sizes:
            eval_key_size = self.get_size("./serialized_evaluation_keys.ekl", 'kb')
            print(f"Evaluation key size: {eval_key_size} kB")

            # Check the size of the evaluation keys (in MB)
            priv_key_size = self.get_size("./keys", 'kb')
            print(f"Private key size: {priv_key_size} kB")

        return filename

    def getFeaturesAndClasses(self, file = os.path.join(os.path.dirname(__file__), "features_and_classes.txt")):
        """Parses 'features_and_classes.txt' in the current directory and extracts a dictionary containing the selected features and the original class labels for decryption."""
        with open(file, "r") as fc_file:
            dictionary = json.loads(fc_file.readline())
            self.selected_features = dictionary["features"]
            self.classes_labels = dictionary["classes"]
            self.classes_labels = {int(key):value for key, value in self.classes_labels.items()}
            print(self.selected_features)
            print(self.classes_labels)

    def dropColumns(self, dashing_output):
        """Drops columns from the Dashing output based on the parsed selected features from getFeaturesAndClasses()."""
        features = self.selected_features
        feature_list = ["Accession ID"] + features

        drop_df = read_csv(dashing_output)
        drop_df = drop_df[[column.strip() for column in feature_list]]
        drop_df.to_csv("./output.csv", index=False, header=True)

    def readTruncateSequence(self, fasta_fpath, verbose = False):
        """Reads the entire sequence from the input file and truncates it before creating a new FASTA file with the truncated sequence."""
        truncated_seq = ""

        with open(fasta_fpath, "r") as f:
            for line in f.readlines(): #chunks() method is essentially opening the file in binary mode.
                if ">" not in line:

                    to_add = line.strip().replace('\n', '')

                    truncated_seq += to_add
                else:
                    if verbose:
                        print("> found.")
                    if("|" not in line):
                        self.writeOutput("Warning: Please follow the recommended input file structure: >Reference/Database|AccessionID|DateCollected")
                        first_line = line
                        id = line.split(" ")[0].strip().replace('>', '')
                    else:
                        first_line = line
```

```
                        id = line.split("|")[1].strip()

        decoded_truncated_seq = truncated_seq[20000:]

        return first_line, decoded_truncated_seq, id

    def writeFastaBytes(self, id, first_line, sequence):
        """Writes a .fasta BYTES file in the 'fastas' folder named after the fasta's ID and containing the truncated sequence."""
        fasta_folder = os.path.join(os.path.dirname(__file__), f"fastas")
        if not os.path.exists(fasta_folder):
            os.mkdir(fasta_folder)

        with open(os.path.join(fasta_folder, f"{id}.fasta"), "wb") as output_file:
            output_file.write(first_line)
            output_file.write(sequence)

    def writeFasta(self, id, first_line, sequence):
        """Writes a .fasta file in the 'fastas' folder named after the fasta's ID and containing the truncated sequence."""
        fasta_folder = os.path.join(os.path.dirname(__file__), f"fastas")
        if not os.path.exists(fasta_folder):
            os.mkdir(fasta_folder)

        with open(os.path.join(fasta_folder, f"{id}.fasta"), "w") as output_file:
            output_file.write(first_line)
            output_file.write(sequence)

    def verifyDashingIntegrity(self, list):
        """Checks if the Dashing files have been downloaded properly. Forces re-download as needed."""
        for dashing_file in list:
            try:
                mb_size = os.stat(dashing_file).st_size / (1024*1024)
                if(mb_size < 18): #dashing files should not be less than 18 mb
                    raise Exception("Dashing files may not have been downloaded correctly. Redownloading the files...")
            except Exception as e:
                self.writeOutput(str(e))
                getRequiredFiles(force_download=True)

    def useDashing(self):
        """Calls the appropriate shell scripts (dashingShell<bits>.sh) and files after giving them execution permissions.
        Takes into account instruction set incompatibility and attempts to use lower-bit binary releases if possible."""

        files_to_allow = [
            os.path.join(os.path.dirname(__file__),'dashingShell512.sh'),
            os.path.join(os.path.dirname(__file__),'dashing_s512'),
            os.path.join(os.path.dirname(__file__),'readHLLandWrite512.sh'),
            os.path.join(os.path.dirname(__file__),'dashingShell256.sh'),
            os.path.join(os.path.dirname(__file__),'dashing_s256'),
            os.path.join(os.path.dirname(__file__),'readHLLandWrite256.sh'),
            os.path.join(os.path.dirname(__file__),'dashingShell128.sh'),
            os.path.join(os.path.dirname(__file__),'dashing_s128'),
            os.path.join(os.path.dirname(__file__),'readHLLandWrite128.sh'),
        ]

        files_to_verify = [
            os.path.join(os.path.dirname(__file__),'dashing_s512'),
            os.path.join(os.path.dirname(__file__),'dashing_s256'),
            os.path.join(os.path.dirname(__file__),'dashing_s128'),
        ]

        for f in files_to_allow:
            st = os.stat(f)
            os.chmod(f, st.st_mode | stat.S_IEXEC)

        self.verifyDashingIntegrity(files_to_verify)

        #calls the shell script and returns CalledProcessError if an exit code is not zero
        try:
            subprocess.check_output(['sh', 'dashingShell512.sh'])
        except subprocess.CalledProcessError as e:
            self.writeOutput(f"Error running default dashing_s512: {'OS must support AVX512BW instructions'}.")
            self.writeOutput("Trying dashing_s256...")
            try:
                subprocess.check_output(['sh', 'dashingShell256.sh'])
            except subprocess.CalledProcessError as e:
                self.writeOutput(f"Error running default dashing_s256: {'OS must support AVX2 instructions.'}")
                self.writeOutput("Trying dashing_s128...")
                try:
                    subprocess.check_output(['sh', 'dashingShell128.sh'])
                except subprocess.CalledProcessError as e:
                    self.writeOutput(f"Error running all dashing binaries: {'OS must support AVX512BW, AVX2, or SSE2 instructions'}")

    def beginDecryption(self):
        """Begins decryption of the encrypted prediction results received from the server after classification.
        Expects the input filepath (self.decrypt_name_var) to be a .zip file, and raises an error if not.
        Also saves the prediction results to a CSV file for viewing later."""
        try:
```

```
474
475             if not self.decrypt_name_var.get().endswith(".zip"):
476                 raise Exception("Invalid file type: Only .zip files are supported. Was there an error the server?
        ")
477
478             self.writeOutput("Beginning decryption of encrypted predictions recieved from server...")
479
480             decrypted_predictions = []
481
482             #setting classes dictionary
483             try:
484                 classes_dict = self.classes_labels
485             except:
486                 classes_dict = {0: 'B.1.1.529 (Omicron)', 1: 'B.1.617.2 (Delta)', 2: 'B.1.621 (Mu)', 3: 'C.37 (
        Lambda)'}
487
488             pred_folder = os.path.join(os.path.dirname(__file__), "predictions")
489
490             zip_name = self.decrypt_name_var.get()
491
492             with zipfile.ZipFile(zip_name, "r") as zObject:
493                 zObject.extractall(path=pred_folder)
494
495             enc_file_list = [filename for filename in os.listdir(pred_folder) if filename.endswith(".enc")]
496
497             for filename in enc_file_list:
498                 with open(os.path.join(pred_folder, filename), "rb") as f:
499                     decrypted_prediction = self.fhe_model_client.deserialize_decrypt_dequantize(f.read())[0]
500                     decrypted_predictions.append(decrypted_prediction)
501
502             decrypted_predictions_classes = numpy.array(decrypted_predictions).argmax(axis=1)
503             final_output = [classes_dict[output] for output in decrypted_predictions_classes]
504
505             for i in range(len(final_output)):
506                 self.data_dictionary[i]['result'] = final_output[i]
507
508             self.writeOutput("Prediction Results:")
509             for dictionary in self.data_dictionary.values():
510                 self.writeOutput(f"ID {dictionary['id']}: {dictionary['result']}")
511                 self.writePredOutput(f"\n{datetime.now().strftime('%d/%m/%Y %H:%M:%S')} -- ID {dictionary['id']}:
         {dictionary['result']}")
512
513             self.writeOutput("Saving prediction results to output file...")
514
515             #create a file to save the prediction into
516             self.savePredictionResult()
517             self.writeOutput("Saving completed! Thank you for using the tool!")
518
519
520         except Exception as e:
521             self.writeOutput(f"Error: {str(e)}")
522
523     def savePredictionResult(self):
524         for dict in self.data_dictionary.values():
525             print(dict)
526             df = pd.from_dict({key: [str(value).split(" ")[0]] for key, value in dict.items()})
527             output_name = f"prediction_result_{dict['id']}_{date.today()}.csv"
528             df.to_csv(os.path.join(os.path.dirname(__file__), f"predictions/{output_name}"), index=False, header=
        True)
529 #endregion
530
531 #region functions outside the class
532
533 def getRequiredFiles(force_download = False):
534     """Downloads the required files for the application. Set force_download to True to download the files even if
          present in the directory. Will be called if the Dashing binaries were not downloaded correctly.
535
536     By default, targets the project's GitHub repository (see 'files' array) for downloading the files, but can be
          set to localhost:8000 to download from the local deployment server."""
537     files = [
538         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/ClientDownloads/
        dashing_s512",
539         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/Compiled%20Model/client.zip
        ",
540         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/ClientDownloads/
        selected_features.txt",
541         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/ClientDownloads/
        dashingShell512.sh",
542         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/
        AlternativeDashingDownloads/dashingShell128.sh",
543         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/
        AlternativeDashingDownloads/dashingShell256.sh",
544         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/ClientDownloads/
        readHLLandWrite512.sh",
545         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/
        AlternativeDashingDownloads/readHLLandWrite128.sh",
546         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/
        AlternativeDashingDownloads/readHLLandWrite256.sh",
547         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/Compiled%20Model/client.zip
        ",
548         r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/ClientDownloads/
        features_and_classes.txt",
```

```
        r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/AlternativeDashingDownloads/dashing_s128",
        r"https://raw.githubusercontent.com/bjorgkav/concreteml-covid-classifier/main/client/AlternativeDashingDownloads/dashing_s256",
        ]

    for file in files:
        parsed_name = file.split("/")[-1].replace("%20", " ")
        print(f"Checking current directory for file: {parsed_name}")
        if (parsed_name not in os.listdir(os.path.dirname(__file__))) or force_download:
            download(file, os.path.dirname(__file__))

def download(url, dest_folder):
    if not os.path.exists(dest_folder):
        os.makedirs(dest_folder)

    filename = url.split('/')[-1].replace(" ", "_")
    file_path = os.path.join(dest_folder, filename)

    r = requests.get(url, stream=True)

    if r.ok:
        print("saving to", os.path.abspath(file_path))
        with open(file_path, 'wb') as f:
            for chunk in r.iter_content(chunk_size=1024 * 8):
                if chunk:
                    f.write(chunk)
                    f.flush()
                    os.fsync(f.fileno())
    else:  # HTTP status code 4XX/5XX
        print("Download failed: status code {}\n{}".format(r.status_code, r.text))
#endregion

if __name__ == "__main__":
    getRequiredFiles()

    app = ClientTkinterUiDesignApp()
    app.run()
```

Listing 7: Source code for client-Side application

```
#!/usr/bin/bash

#given the data from the paths indicated in path.txt,
#split the each sequence into k-mers, and
#compute the HLL sketch of each sequence, with a sketch size (spacing) of 9

echo "Running Dashing tool..."

./dashing_s128 sketch -k31 -p13 -S9 fastas/*.fasta

if [ $? -ne 0 ]; then
    echo "Error encountered using this dashing binary. Use a different binary."
    exit 1
fi

echo "Reading output and Creating CSV..."

./readHLLandWrite128.sh #(requires chmod +x readHLLandWrite.sh for execution permissions)
```

Listing 8: Source code for Dashing Script (128-bit instruction set)

```
#!/usr/bin/bash

#given the data from the paths indicated in path.txt,
#split the each sequence into k-mers, and
#compute the HLL sketch of each sequence, with a sketch size (spacing) of 9

echo "Running Dashing tool..."

./dashing_s256 sketch -k31 -p13 -S9 fastas/*.fasta

if [ $? -ne 0 ]; then
    echo "Error encountered using this dashing binary. Use a different binary."
    exit 1
fi

echo "Reading output and Creating CSV..."

./readHLLandWrite256.sh #(requires chmod +x readHLLandWrite.sh for execution permissions)
```

Listing 9: Source code for Dashing Script (256-bit instruction set)

```
#!/usr/bin/bash

#given the data from the paths indicated in path.txt,
```

```
#split the each sequence into k-mers, and
#compute the HLL sketch of each sequence, with a sketch size (spacing) of 9

echo "Running Dashing tool..."

./dashing_s512 sketch -k31 -p13 -S9 fastas/*.fasta

if [ $? -ne 0 ]; then
    echo "Error encountered using this dashing binary. Use a different binary."
    exit 1
fi

echo "Reading output and Creating CSV..."

./readHLLandWrite512.sh #(requires chmod +x readHLLandWrite.sh for execution permissions)
```

Listing 10: Source code for Dashing Script (512-bit instruction set)

```
#!/bin/bash
#clear output file
rm ./output.txt

#assume feature count is 512
echo -n "Accession ID," >> ./output.txt

no_of_features=512
for ((i=1; i<$no_of_features; i++))
do
    printf %s "feature_$i," >> ./output.txt
done

printf "%s\n" "feature_$i" >> ./output.txt

for f in ./fastas/*.hll
do
    content=$(./dashing_s128 view $f)
    echo "$f, $content" >> ./output.txt
done

#remove all occurrences of [ and ] in output
echo "Removing [ and ] from output and placing in csv..."
echo -n "$(sed -i 's/[][]//g' ./output.txt)"
echo -n "$(sed -i 's/\<fastas\>//g' ./output.txt)"
echo -n "$(sed -i 's/\///g' ./output.txt)"
echo -n "$(sed -i 's/\.//g' ./output.txt)"
echo "$(sed -i 's/fastas//g' ./output.txt)"
echo "$(sed -i 's/fastaw31spacing9hll//g' ./output.txt)"

mv ./output.txt ./output.csv
```

Listing 11: Shell script for converting Dashing output to CSV (128-bit instruction set)

```
#!/bin/bash
#clear output file
rm ./output.txt

#assume no is 512
echo -n "Accession ID," >> ./output.txt

no_of_features=512
for ((i=1; i<$no_of_features; i++))
do
    printf %s "feature_$i," >> ./output.txt
done

printf "%s\n" "feature_$i" >> ./output.txt

for f in ./fastas/*.hll
do
    content=$(./dashing_s256 view $f)
    echo "$f, $content" >> ./output.txt
done

#remove all occurrences of [ and ] in output
echo "Removing [ and ] from output and placing in csv..."
echo -n "$(sed -i 's/[][]//g' ./output.txt)"
echo -n "$(sed -i 's/\<fastas\>//g' ./output.txt)"
echo -n "$(sed -i 's/\///g' ./output.txt)"
echo -n "$(sed -i 's/\.//g' ./output.txt)"
echo "$(sed -i 's/fastas//g' ./output.txt)"
echo "$(sed -i 's/fastaw31spacing9hll//g' ./output.txt)"

mv ./output.txt ./output.csv
```

Listing 12: Shell script for converting Dashing output to CSV (256-bit instruction set)

```
#!/bin/bash
#clear output file
rm ./output.txt

#assume feature count is 512
echo -n "Accession ID," >> ./output.txt

no_of_features=512
for ((i=1; i<$no_of_features; i++))
do
    printf %s "feature_$i," >> ./output.txt
done

printf "%s\n" "feature_$i" >> ./output.txt

for f in ./fastas/*.hll
do
    content=$(./dashing_s512 view $f)
    echo "$f, $content" >> ./output.txt
done

#remove all occurrences of [ and ] in output
echo "Removing [ and ] from output and placing in csv..."
echo -n "$(sed -i 's/[][]//g' ./output.txt)"
echo -n "$(sed -i 's/\<fastas\>//g' ./output.txt)"
echo -n "$(sed -i 's/\///g' ./output.txt)"
echo -n "$(sed -i 's/\.//g' ./output.txt)"
echo "$(sed -i 's/fastas//g' ./output.txt)"
echo "$(sed -i 's/fastaw31spacing9hll//g' ./output.txt)"

mv ./output.txt ./output.csv
```

Listing 13: Shell script for converting Dashing output to CSV (512-bit instruction set)

XI. Acknowledgment

This project would not have come to fruition if not for the following people:

My adviser, Sir Richard Bryann Chua, to whom I would like to express my deepest appreciation and gratitude for his invaluable guidance, supervision, and patience in the course of this project. The consistency with which he meets me weekly for consultations allowed me to make regular progress in my project, and his willingness to teach and guide me as I learned the various technologies used in this study was invaluable to the success of this endeavor. I could not have finished this project without his help. I also could not have completed this journey without the guidance and feedback of my defense committee.

I am also grateful to my fellow member of the Homomorphic Encryption Subgroup, Gwyneth Rose C. Rosario, for her assistance and feedback in the implementation of the project and editing. I would like to extend my sincere thanks to my blockmates, especially my group-mates in the Security and Cryptography Research Group, Mirai Reyes Yoshizaki, Kyle Mari Angelo M. Aquino, Julius Allen Reyes, and Gabriel Austin Untalan, for their feedback, assistance in formatting, and moral support.

Lastly, I would be remiss in not mentioning my family, especially my mother, Sarah Jane M. Pañares, and my brother, Ian Miguel P. Vivas, as well as my partner, Anjelle Julene A. Cadeliña, for providing me with mental, emotional, and occasional financial support as I saw this project through to its completion. I would not have gotten this far without your faith in me. I would also like to thank my dogs and cats for their emotional support.