GuardML: Efficient Privacy-Preserving Machine Learning Services Through Hybrid Homomorphic Encryption

Eugene Frimpong
Tampere University
Tampere, Finland
eugene.frimpong@tuni.fi

Khoa Nguyen
Tampere University
Tampere, Finland
khoa.nguyen@tuni.fi

Mindaugas Budzys
Tampere University
Tampere, Finland
mindaugas.budzys@tuni.fi

Tanveer Khan
Tampere University
Tampere, Finland
tanveer.khan@tuni.fi

Antonis Michalas
Tampere University
Tampere, Finland
antonios.michalas@tuni.fi

ABSTRACT

Machine Learning (ML) has emerged as one of data science's most transformative and influential domains. However, the widespread adoption of ML introduces privacy-related concerns owing to the increasing number of malicious attacks targeting ML models. To address these concerns, Privacy-Preserving Machine Learning (PPML) methods have been introduced to safeguard the privacy and security of ML models. One such approach is the use of Homomorphic Encryption (HE). However, the significant drawbacks and inefficiencies of traditional HE render it impractical for highly scalable scenarios. Fortunately, a modern cryptographic scheme, Hybrid Homomorphic Encryption (HHE), has recently emerged, combining the strengths of symmetric cryptography and HE to surmount these challenges. Our work seeks to introduce HHE to ML by designing a PPML scheme tailored for end devices. We leverage HHE as the fundamental building block to enable secure learning of classification outcomes over encrypted data, all while preserving the privacy of the input data and ML model. We demonstrate the real-world applicability of our construction by developing and evaluating an HHE-based PPML application for classifying heart disease based on sensitive ECG data. Notably, our evaluations revealed a slight reduction in accuracy compared to inference on plaintext data. Additionally, both the analyst and end devices experience minimal communication and computation costs, underscoring the practical viability of our approach. The successful integration of HHE into PPML provides a glimpse into a more secure and privacy-conscious future for machine learning on relatively constrained end devices.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**;
• **Security and privacy** → **Public key (asymmetric) techniques**.

SAC '24, April 8–12, 2024, Avila, Spain

ACM ISBN 979-8-4007-0243-3/24/04.
https://doi.org/10.1145/3605098.3635983

KEYWORDS

Hybrid Homomorphic Encryption, Machine Learning as a Service, Privacy-Preserving Machine Learning

ACM Reference Format:
Eugene Frimpong, Khoa Nguyen, Mindaugas Budzys, Tanveer Khan, and Antonis Michalas. 2024. GuardML: Efficient Privacy-Preserving Machine Learning Services Through Hybrid Homomorphic Encryption. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC '24), April 8–12, 2024, Avila, Spain.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3605098.3635983

1 INTRODUCTION

Machine Learning (ML) has increasingly become one of the most impactful fields of data science in recent years, allowing various users to classify and make predictions based on multi-dimensional data. One of the main metrics for ML is the accuracy of the prediction or classification results. However, to achieve this, the results should be accompanied by a large amount of high-quality training data requiring the collaboration of several organizations. Currently, regulations such as the General Data Protection Regulations (GDPR) forbid the sharing and processing of sensitive data without the data subject's consent. It has, therefore, become crucial to uphold data privacy and confidentiality when obtaining data from other organizations. One solution to this problem is employing Privacy-Preserving Machine Learning (PPML). PPML ensures that the use of data protects user privacy and that data is utilized in a safe fashion, avoiding leakage of confidential and private information. To this end, researchers have proposed and implemented various PPML-achieving techniques, varying from secure cryptographic schemes to distributed, hybrid, and data modification approaches. This work focuses on cryptographic approaches, the most commonly used one being Homomorphic Encryption (HE) [14, 28], which exhibits a high potential in ML applications.

HE allows users to perform computations such as addition or multiplication on encrypted data [28]. One of the first fully HE (FHE) schemes was proposed by C. Gentry [14]. FHE allows infinite operations on encrypted data while still producing a valid result. Since Gentry's seminal work [14], multiple HE schemes have been proposed to improve the efficiency and applicability of HE. Currently, the most popular schemes used are **CKKS** [7], **TFHE** [8] and **BFV** [5, 13]. Because these schemes allow arbitrary

computation on encrypted data, FHE has created exciting new applications in several areas, such as Machine Learning as a Service (MLaaS)[18–24, 27]. In MLaaS applications, data is encrypted under an HE scheme and transferred to the cloud for processing. Furthermore, due to the versatility of HE, it can also be used to keep the model private through model parameter encryption. Despite the various advances in HE, it has yet to find mainstream use because of high computational complexity and extended ciphertext expansion, resulting in very large ciphertexts. To address these issues, researchers have turned to the Hybrid Homomorphic Encryption (HHE) concept [3, 12], which uses symmetric ciphers to make HE more accessible for users.

HHE in a Nutshell: In an HHE scheme, rather than encrypt data with only an HE scheme, users locally encrypt data via a symmetric key encryption scheme. Subsequently, they homomorphically encrypt the symmetric key used in the encryption process with an HE scheme. The two ciphertexts resulting from the abovementioned encryptions are then forwarded to the server. Upon receiving both ciphertexts, the server transforms the symmetric ciphertext into a homomorphic ciphertext using the homomorphically encrypted symmetric key. The use of HHE produces ciphertexts of a substantially smaller size compared to using only HE schemes because of the symmetric encryption. The reduced size drastically lowers the communication overhead a user would typically incur using only a traditional HE approach. However, due to high multiplicative depth, not all symmetric key encryption schemes are compatible with HHE. To this end, researchers have designed a number of HE-friendly symmetric ciphers for use in HHE schemes, such as HERA/Rubato [10, 16], Elisabeth [11] and PASTA [12]. Aside from reducing ciphertext size, HHE also provides a way for most consumer-grade devices to benefit from the advantages of HE schemes by transferring computationally expensive operations to the server or cloud.

For this work, we adopt the concept of HHE, as implemented in PASTA [12], to address the privacy problem of machine learning prediction as a service. Apart from the novelty of incorporating HHE in PPML, we aim to smooth out the big hurdles of implementing strong PPML models in a wide range of devices.

Contributions: We provide a realistic solution that carefully considers the vagaries of PPML, all while exploring the use of new technologies that might unleash creative ideas in the decades to come. Additionally, the paper makes the following contributions:

- **C**1. We first demonstrate the effective utilization of HHE to tackle the challenges of PPML. By extending the use of PASTA to ML applications, our approach introduces HHE as a key element in PPML, unlocking new possibilities. Our primary goal is to overcome the significant obstacles in implementing robust PPML models across diverse devices.
- **C**2. We present two formally designed protocols enabling an authorized entity (e.g., an analyst) to process encrypted data as if it were unencrypted efficiently. Rigorous security proofs demonstrate that our protocols preserve user privacy, ensuring no leakage that could compromise confidentiality.
- **C**3. Through extensive experiments, we showcase the practicality of our protocol in a real-world ML scenario using a sensitive medical dataset. The experimental results indicate that our PPML protocol achieves nearly comparable accuracy to the plaintext version while safeguarding both the dataset and the neural network's privacy. Additionally, most of the computation cost is effectively outsourced to a CSP.

2 RELATED WORKS

Homomorphic Encryption: Various recent works have proposed using HE schemes in implementing PPML. Gentry's work [14] revolutionized the field of HE and paved the way for multiple modern schemes, such as TFHE [9], BFV [13], and CKKS [7] in PPML applications. BFV was one of the first improvements to the original HE scheme proposed by Gentry and works by limiting expensive bootstrapping operations. BFV is, therefore, referred to as a Somewhat Homomorphic Encryption (SHE) scheme and allows a limited amount of operation to be performed on integer ciphertexts. Cheon *et al.* also proposed the CKKS scheme, which also allows a limited number of operations to be performed on ciphertext but allows computations on floating point data [7]. Chillotti *et al.* then proposed the TFHE [8] scheme. TFHE greatly improves the efficiency of bootstrapping and allows an unlimited amount of bitwise operations on binary data. Each of the aforementioned schemes has been applied to ML applications, requiring different techniques to implement non-linear activation functions. For example, TFHE relies on fast and efficient LUT searches to compute non-linear activations [24], while BFV and CKKS require polynomial approximations [17, 23]. Each HE scheme has been applied to PPML with high-accuracy results. Examples of TFHE-based PPML works are TAPAS [29], FHE-DiNN [4] and Glyph [24]. An example of a CKKS-based PPML protocol is POSEIDON [30], while an example of a BFV-based PPML work is HCNN [2]. HE schemes suffer from large ciphertext sizes and high computational complexities, which make them unsuitable for every environment. Enter HHE.

Hybrid Homomorphic Encryption: The first approaches to designing HHE schemes relied on existing and well-established symmetric ciphers such as AES [15]. However, AES has been proven to not be a good fit for HHE schemes, primarily due to its large multiplicative depth [12]. Thus, research in the field of HHE took a different approach, where the main focus shifted to the design of symmetric ciphers with different optimization criteria, such as eliminating the ciphertext expansion [6] or using filter permutators [25]. However, to date, HHE has seen limited practical application [3] in real-world applications, and only a handful of works exist in the field of PPML. To the best of our knowledge, the *main* HHE schemes currently are HERA [10], Elisabeth [11] and PASTA [12]. The authors of HERA also proposed Rubato [16]; however, the specifications remain largely the same as in HERA. These proposed approaches have different specifications and can be applied to different use cases. HERA [10] is a stream cipher based on the CKKS HE scheme and allows computations on floating point data types. In comparison, Elisabeth [11] is designed to utilize the TFHE scheme, while PASTA [12] is based on BFV for integer data types.

HERA and PASTA are defined over $\mathbb{Z}_q$, where $q = 2^{16} + 1$, and can store up to 16-bit inputs. Meanwhile, Elisabeth is defined over $\mathbb{Z}_q$, where $q = 2^4$, and can store up to 4 bits of data. Each approach achieves the same security level of 128-bits. Additionally, HERA also provides tests for a security level of 80 bits. As HERA allows

	HERA [10]	**Elisabeth** [11]	**PASTA** [12]
Programming Language	Go	Rust	C
Data Type	Floating point	Binary	Integer
HE Scheme	CKKS	TFHE	BFV
Application in ML	No	Yes	No
Defined over	$\mathbb{Z}_q$, where $q > 2^{16}$	$\mathbb{Z}_q$, where $q = 2^4$	$\mathbb{F}_p$, where p is a 16-bit prime
Security Level	80 or 128-bit	128-bit	128-bit
Quantization	No	Yes	Yes

Table 1: Comparison of different HHE schemes

computations on floating point data types, it does not require quantization on certain inputs. Elisabeth and PASTA, on the other hand, require quantization to operate on floating point numbers, which introduces a rounding error, which can reduce the accuracy of certain applications, i.e. ML. To the best of our knowledge, HERA has yet to be applied to an ML application. One of the use cases provided by Elisabeth is a CNN classification task on the Fashion-MNIST dataset and shows equivalent accuracy (84.18%) to the cleartext model without HE. Table 1 provides an overview of the different popular HHE schemes. For this work, we aim to evaluate the suitability of PASTA when applied to PPML and investigate the feasibility of applying PASTA to various real-world ML datasets, particularly medical datasets, with the potential to achieve comparable accuracy and higher efficiency than conventional HE techniques.

3 PRELIMINARIES

3.1 Homomorphic Encryption

Definition 3.1 (Homomorphic Encryption). Let HE be a (public-key) homomorphic encryption scheme with a quadruple of PPT algorithms HE = (KeyGen,Enc, Dec, Eval) such that:

- **HE.KeyGen** : The key generation algorithm $(\mathsf{pk}, \mathsf{evk}, \mathsf{sk}) \leftarrow \mathsf{HE.KeyGen}(1^\lambda)$ takes as input a unary representation of the security parameter λ, and outputs a public key pk, an evaluation key evk and a private key sk.
- **HE.Enc** : The encryption algorithm $c \leftarrow \mathsf{HE.Enc}(\mathsf{pk}, x)$ takes as input the public key pk and a message x and outputs a ciphertext c.
- **HE.Eval** : The algorithm $c_f \leftarrow \mathsf{HE.Eval}(\mathsf{evk}, f, c_1, \ldots, c_n)$ takes as input the evaluation key evk, a function f, and a set of n ciphertexts, and outputs a ciphertext c_f.
- **HE.Dec** : The decryption algorithm $\mathsf{HE.Dec}(\mathsf{sk}, c) \rightarrow x$, takes as input the secret key sk and a ciphertext c, and outputs a plaintext x.

3.2 Hybrid Homomorphic Encryption

Definition 3.2 (Hybrid Homomorphic Encryption). Let HE be a Homomorphic Encryption scheme and SKE = (Gen, Enc, Dec) be a symmetric-key encryption scheme. Moreover, let $\mathcal{X} = (x_1, \ldots, x_n)$ be the message space and λ the security parameter. An HHE scheme consists of five PPT algorithms HHE = (KeyGen, Enc, Decomp, Eval, Dec) such that:

- **HHE.KeyGen**: The key generation algorithm takes as input a security parameter λ and outputs a HE public/private key pair (pk/sk) and a HE evaluation key (evk).
- **HHE.Enc**: The encryption algorithm consists of three steps:
  - SKE.Gen: The SKE generation algorithm takes as input the security parameter λ and outputs a symmetric key K.
  - HE.Enc: An HE encryption algorithm that takes as input pk and K, and outputs c_K – a homomorphically encrypted representation of the symmetric key K.
  - SKE.Enc: The SKE encryption algorithm takes as input a message x and K and outputs a ciphertext c.
- **HHE.Decomp**: This algorithm takes as an input the evaluation key evk, the symmetrically encrypted ciphertext c, and the homomorphically encrypted symmetric key c_K, and outputs c' – a homomorphic encryption of the message x.
- **HHE.Eval**: This algorithm takes as input n homomorphic ciphertexts c'_n, where $n \geq 2$, the evaluation key evk and a homomorphic function f, and outputs a ciphertext c'_{eval} of the evaluation results.
- **HHE.Dec**: The decryption algorithm takes as input a private key sk and the evaluated ciphertext c'_{eval} and outputs $f(x)$.

The correctness of an HHE scheme follows directly from the correctness of the underlying public-key HE scheme.

3.3 Machine Learning

ML is a set of algorithms that leverage already-available training data as input to train a model $f(\boldsymbol{\theta})$. The process of training is to find the optimal parameters $\boldsymbol{\theta} = (w, b)$, where w are weights matrices and b are biases that can provide accurate predictions on the training data. Once trained, $f(\boldsymbol{\theta})$ can be used to provide predictions on unseen input data, which is called the "inference" or "prediction" phase. In this work, we leverage HHE to build PPML protocols that preserve data and model privacy in inference phase.

4 SYSTEM MODEL

In this section, we introduce our system model by explicitly describing our protocol's main entities and their capabilities.

- **User**: Let $\mathcal{U} = \{u_1, \ldots, u_n\}$ be the set of all users. Each user generates a unique symmetric key $\mathsf{K_i}$ locally and encrypts their data. Then the generated ciphertexts are outsourced to the **CSP** along with an HE encryption $c_{\mathsf{K_i}}$ of the underlying symmetric key.
- **Cloud Service Provider (CSP)**: Primarily responsible for gathering symmetrically encrypted data from multiple users. The **CSP** is tasked with converting the symmetrically encrypted data into homomorphic ciphertexts and, upon request, performing blind operations on them.
- **Analyst (A)**: In one of our proposed protocols, there exists an analyst who owns an ML model and is interested in learning the output of ML operations on the encrypted data stored at the **CSP**. In this protocol, **A** decrypts the encrypted data from the HE evaluation of collected encrypted data and, thus, may gain insights from user data.

5 GUARDML

In this section, we discuss in detail the construction of `GuardML` – our Hybrid Homomorphic Privacy-Preserving protocols that constitute the core of this paper. `GuardML` is comprised of two protocols, `2GML` and `3GML`. The primary differences between the protocols mentioned above lie in the presence of an **analyst** and support for

a **multi-user** scenario. Each protocol may be suitable for different use cases. For example, 2GML can be used in commercial ML applications where the **CSP**, such as AWS, Azure, etc., owns the ML models. While 3GML, on the other hand, would be ideal for a setting where the analyst is owner of the model and does not wish to reveal the contents to **CSP**. In this setting, the **CSP**'s role is reduced to simply performing operations on encrypted data and models.

Building Blocks: Before proceeding to describe each protocol, we first define the building blocks used in our constructions.

- A secure symmetric cipher SKE = (Gen, Enc, Dec).
- A BFV-based HHE scheme HHE = (KeyGen, Enc, Dec, Deco mp, Eval).

Additionally, to provide secure communication, we define a public key encryption scheme, which supports message encryption and decryption, a signature scheme used for message signing and verification, and a secure cryptographic hash function to verify message integrity. We make the following assumptions:

- A CCA2 secure public-key encryption scheme PKE = (Gen, Enc, Dec).
- An EUF-CMA secure signature scheme σ = (sign, ver).
- A first and second pre-image resistant cryptographic hash function $\mathsf{H}(\cdot)$.

5.1 GuardML: 2-Party Setting

High-Level Overview: The first version of GuardML is 2GML – a 2-party protocol that consists of a **CSP** and a user u_i. In this setting, we assume that the **CSP** is the owner of a trained ML model with parameters (w, b), while u_i provides the input data x_i to the model. Initially, the user u_i generates the necessary HHE keys $(\mathsf{pk}_{\mathsf{u}_i}, \mathsf{sk}_{\mathsf{u}_i}, \mathsf{evk}_{\mathsf{u}_i})$. It then publishes the public key $\mathsf{pk}_{\mathsf{u}_i}$ and sends the evaluation key $\mathsf{evk}_{\mathsf{u}_i}$ to the **CSP**. Subsequently, u_i generates a unique symmetric key K_i. On completing the key generation phase, u_i begins the data upload phase by first generating $c_{\mathsf{K}_\mathsf{i}}$, a homomorphic encryption of the symmetric key K_i, and then calculates a symmetric encryption of the data x_i with K_i. The data upload phase concludes with the sending of both ciphertext values to the **CSP**. Upon reception, the **CSP** begins the secure evaluation phase by first transforming the symmetrically encrypted data c_{x_i} into a homomorphic ciphertext c'_{x_i}. On successful run, **CSP** uses $\mathsf{evk}_{\mathsf{u}_i}$, c'_{x_i} and f to produce an encrypted prediction c_{res}. The encrypted result is then sent to u_i for decryption.

Formal Construction: Figure 1 provides an overview of the 2GML protocol, which is divided into four distinct phases, namely 2GML.Setup, 2GML.Upload, 2GML.Eval, and 2GML.Classify.

2GML.Setup: In the setup phase, both parties generate their respective signing/verification key pairs for the signature scheme σ and publish the verification keys. The **CSP** runs the PKE.Gen algorithm to generate a public/private key pair $(\mathsf{pk}_{\mathsf{CSP}}, \mathsf{sk}_{\mathsf{CSP}})$. On the other hand, u_i runs the HHE.KeyGen algorithm to generate the public, private and evaluation keys $(\mathsf{pk}_{\mathsf{u}_i}, \mathsf{sk}_{\mathsf{u}_i}, \mathsf{evk}_{\mathsf{u}_i})$ used for HHE operations. Finally, u_i publishes $\mathsf{pk}_{\mathsf{u}_i}$ and outsources $\mathsf{evk}_{\mathsf{u}_i}$ to the **CSP** through m_1.

$$m_1 = \langle t_1, \mathsf{Enc}(\mathsf{pk}_{\mathsf{CSP}}, \mathsf{evk}_{\mathsf{u}_i}), \sigma_{u_i}(\mathsf{H}(t_1||\mathsf{evk}_{\mathsf{u}_i}))\rangle,$$

where σ_{u_i} is a signature created by u_i. Upon reception, the **CSP** verifies the signature by using u_i's verification key and the freshness of the message through the timestamp. If the verification fails, the **CSP** aborts the protocol and outputs $\perp$. Otherwise, the **CSP** stores $\mathsf{evk}_{\mathsf{u}_i}$ locally.

2GML.Upload: In this phase, u_i runs the HHE.Enc algorithm to generate a symmetric encryption key K_i through SKE.Gen and then encrypts the plaintext data x_i into ciphertext c_{x_i} using the SKE.Enc algorithm, which takes $(\mathsf{K}_\mathsf{i}, x_i)$ as an input. After encrypting the data, u_i also runs HE.Enc to homomorphically encrypt K_i into $c_{\mathsf{K}_\mathsf{i}}$ with their $\mathsf{pk}_{\mathsf{u}_i}$. After both encryptions are finished u_i sends both ciphertexts to the **CSP** with m_2:

$$m_2 = \langle t_2, c_{x_i}, c_{\mathsf{K}_\mathsf{i}}, \sigma_{u_i}(\mathsf{H}(t_2||c_{x_i}||c_{\mathsf{K}_\mathsf{i}}))\rangle,$$

On receiving m_2, the **CSP** verifies the signature by using u_i's verification key and the freshness of the message through the timestamp. If the verification fails, the **CSP** aborts the protocol and outputs $\perp$. Otherwise, the **CSP** continues to the secure evaluation phase.

2GML.Eval: The secure evaluation phase begins with the **CSP** transforming the received symmetric ciphertext c_{x_i} into HE ciphertext c'_{x_i} by running the HHE.Decomp algorithm. HHE.Decomp uses HE.Eval which takes as an input $(\mathsf{evk}_{\mathsf{u}_i}, \mathsf{SKE.Dec}, c_{\mathsf{K}_\mathsf{i}}, c_{x_i})$, where SKE.Dec is the symmetric cipher decryption algorithm to transform the ciphertext. Afterwards, the **CSP** takes c'_{x_i} and their ML model parameters (w, b) and inputs both of them into the HHE.Eval along with the $\mathsf{evk}_{\mathsf{u}_i}$ to compute an encrypted prediction c_{res} from the encrypted data. Finally, the **CSP** securely sends the encrypted result back to u_i via m_3:

$$m_3 = \langle t_3, c_{res}, \sigma_{CSP}(\mathsf{H}(t_3||c_{res}))\rangle,$$

Upon reception, u_i verifies the integrity and the freshness of the message. If the verification fails, u_i aborts the protocol and outputs $\perp$. Otherwise, continues to the final phase.

2GML.Classify: In the final phase of the protocol, u_i decrypts the received ciphertext to gain insight into the data. This is done by u_i running the HHE.Dec algorithm with inputs $\mathsf{sk}_{\mathsf{u}_i}, c_{res}$ and outputs the prediction res.

5.2 GuardML: 3-Party Setting

High-Level Overview: We note that 2GML is unsuitable for multi-user scenarios, where data is collected from multiple users and stored at the **CSP**. In such a scenario, each user would be required to generate a unique set of HHE keys, while the **CSP** would need to store the $\mathsf{evk}_{\mathsf{u}_i}$ of each user to run the protocol successfully. To resolve this, we present 3GML – an extended version of 2GML for the multi-client model. 3GML consists of three parties: a set of users $\mathcal{U}$, **CSP**, and an analyst **A**. The protocol's steps are primarily the same as in 2GML (subsection 5.1). The fundamental distinction is that under the three-party setting, the HHE keys are generated by **A** instead of u_i. In the 3GML setting, we assume that an analyst **A** owns the ML model with parameters (w, b), while a user u_i provides the data x_i. **A** generates the required HHE keys $(\mathsf{pk}_\mathsf{A}, \mathsf{sk}_\mathsf{A}, \mathsf{evk}_\mathsf{A})$, publishes pk_A, and sends evk_A to the **CSP**. Each u_i generates a symmetric key K_i and encrypts their data x_i locally to output a symmetric ciphertext c_{x_i}. Additionally, u_i also generates $c_{\mathsf{K}_\mathsf{i}}$ – a homomorphic encryption of the symmetric key K_i using **A**'s public

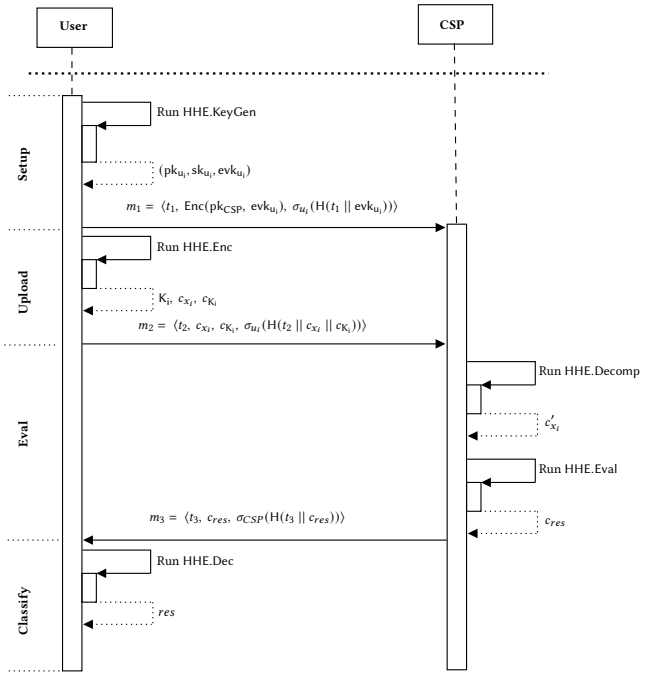

Figure 1: 2GML

key and sends both the encrypted data c_{x_i} and c_{K_i} to the **CSP**. Upon reception, the **CSP** stores the values locally. In the evaluation phase, **A** can request a prediction to be performed on the stored c'_{x_i} by first sending to the **CSP** their homomorphically encrypted pre-trained ML model parameters (c_w, c_b). On receiving the request from **A**, the **CSP** transforms c_{x_i} into homomorphic cihpertext c'_{x_i}. **CSP** produces an encrypted result c_{res} and sends the results back to **A**. Finally, **A** decrypts the prediction result res using sk_A.

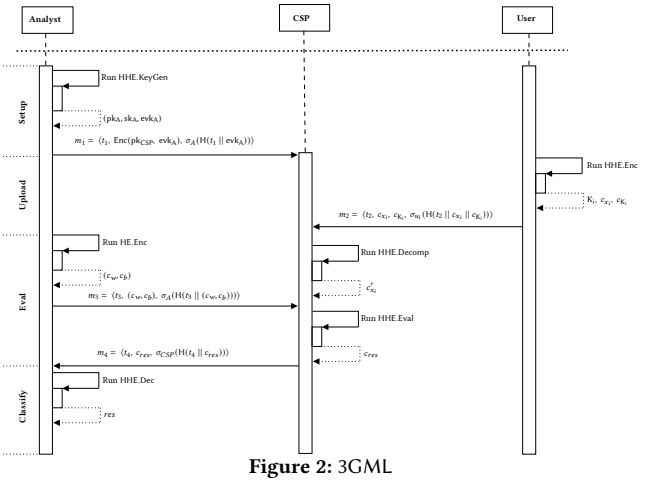

Figure 2: 3GML

Formal Construction: The overall flow of the protocol remains the same as in the two-party protocol. Figure 2 provides a visual overview of the protocol.

3GML.Setup: Each party generates their respective signing/verification key pair for the signature scheme σ and publishes the verification keys. **A** executes the HHE.KeyGen algorithm to output the public, private and evaluation keys (pk_A, sk_A, evk_A). The steps of the phase are summarized below.

u_i computes:	**A** computes:
$(pk_{u_i}, sk_{u_i}) \leftarrow \text{PKE.Gen}(1^\lambda)$	$(ver_A, sign_A) \leftarrow \text{PKE.Gen}(1^\lambda)$,
CSP computes:	$(pk_A, sk_A, evk_A) \leftarrow$
$(pk_{CSP}, sk_{CSP}) \leftarrow \text{PKE.Gen}(1^\lambda)$	$\text{HHE.KeyGen}(1^\lambda)$

A then publishes pk_A and sends evk_A to **CSP** via m_1.

$$m_1 = \langle t_1, \text{Enc}(pk_{CSP}, evk_A), \sigma_A(H(t_1 || evk_A))\rangle$$

3GML.Upload: With the three-party setting, there can be multiple users, namely $u_i \in \mathcal{U}$. Hence, each u_i independently runs HHE.Enc, to generate a unique symmetric key K_i through SKE.Gen. Then, u_i encrypts their data by computing SKE.Enc, which takes K_i and x_i as input and outputs c_{x_i}. Additionally, each u_i homomorphically encrypts its symmetric key K_i using HE.Enc with inputs pk_A and K_i and output c_{K_i}. The steps of this phase are summarized as:

$$K_i \leftarrow \text{SKE.Gen}(1^\lambda),$$
$$\text{SKE.Enc}(K_i, x_i) \rightarrow c_{x_i},$$
$$c_{K_i} \leftarrow \text{HE.Enc}(pk_A, K_i)$$

Finally, u_i sends the encrypted values (c_{x_i} and c_{K_i}) through m_2.

$$m_2 = \langle t_2, c_{x_i}, c_{K_i}, \sigma_{u_i}(H(t_2||c_{x_i}||c_{K_i}))\rangle$$

3GML.Eval: The secure evaluation phase in the three-party protocol is initiated by **A** to gain insight into the data provided by any user u_i. First, **A** homomorphically encrypts their ML model parameters (w, b) by running the HE.Enc algorithm, which takes as input $(pk_A, (w, b))$ and outputs encrypted parameters (c_w, c_b). **A** then sends m_3 to the **CSP** (Figure 2). Upon receiving the encrypted model, the **CSP** transforms c_{x_i} into homomorphic ciphertext c'_{x_i} by running the HHE.Decomp algorithm which takes as input evk_A, c_{x_i}, and c_{K_i}, and outputs c'_{x_i}. Subsequently, the **CSP** runs the HHE.Eval algorithm, which takes as input $(evk_A, (c_w, c_b), c'_{x_i})$ to output an encrypted prediction c_{res}. Afterwards, c_{res} is sent through m_4. The phase is summarized below.

A computes:	**CSP** computes:
$(c_w, c_b) \leftarrow \text{HE.Enc}(pk_A, (w, b))$,	$c'_{x_i} \leftarrow \text{HHE.Decomp}(evk_A, c_{x_i}, c_{K_i})$,
	$c_{res} \leftarrow \text{HHE.Eval}(evk_A, (c_w, c_b), c'_{x_i})$

$$m_3 = \langle t_3, (c_w, c_b), \sigma_A(H(t_3||(c_w, c_b)))\rangle$$
$$m_4 = \langle t_4, c_{res}, \sigma_{CSP}(H(t_4||c_{res}))\rangle$$

6 THREAT MODEL AND SECURITY ANALYSIS

In this section, we define the threat model used to prove the security of GuardML by formalizing the capabilities of an adversary $\mathcal{ADV}$. The PASTA [12] scheme we adopt as our underlying cryptographic

scheme has been proven to be resilient against differential and linear statistical attacks and their variations. The cipher construction incorporates changing linear layers during encryption, which ensures defence against statistical attacks. The authors provide rigorous proof that this layer instantiation is secure, ensuring full diffusion throughout the entire scheme, even in the best-case scenario for an attacker [12]. PASTA is also secure against algebraic attacks, such as Linearization and Gröbner Basis Attacks. To this end, we define a threat model focusing on the communication between entities in our protocols and not the underlying scheme itself. We consider a powerful adversary $\mathcal{ADV}$ capable of performing a variety of attacks aiming at breaking the security and privacy of the protocols. In general, $\mathcal{ADV}$ is capable of corrupting any number of users and the **CSP**. However, for 3GML, we assume that **A** does not collude with the **CSP**. Additionally, we assume that each entity can verify the owner of a public key. With this assumption, we eliminate the possibility of basic man-in-the-middle attacks. From these definitions and assumptions, we present the following possible attacks:

Attack 1 (Ciphertext Substitution Attack). *Let $\mathcal{ADV}$ be a malicious adversary. $\mathcal{ADV}$ successfully launches a Ciphertext Substitution Attack if she manages to replace the generated ciphertexts sent by any entity in an indistinguishable way.*

Attack 2 (ML Model Unauthorized Access Attack). *Let $\mathcal{ADV}$ be a malicious adversary. $\mathcal{ADV}$ successfully launches the ML model unauthorized access attack if she manages to learn information about the underlying ML model utilized by either the **CSP** or the analyst **A**.*

6.1 Security Analysis

We now prove the security of our protocols in the presence of the adversary defined in 6.

Proposition 6.1 (Ciphertext Substitution Attack Soundness). *Let σ be an EUF-CMA secure signature scheme and* PKE *an INC-CPA public key encryption scheme. Then $\mathcal{ADV}$, cannot successfully launch the Ciphertext Substitution Attack against* GuardML.

Proof. To successfully perform the Ciphertext Substitution Attack, $\mathcal{ADV}$ needs to successfully attack the 2GML/3GML.Upload, 2GML.Eval or 3GML.Eval phase of GuardML by substituting the actual ciphertexts with a sequence of $\mathcal{ADV}$ generated ciphertexts. To this end, we categorize the attacks into the following cases:

Option 1 (2GML/3GML.Upload): When a user u_i outsources data to the **CSP** via $m_2 = \langle t_2, c_{x_i}, c_{\mathsf{K}_\mathsf{i}}, \sigma_{u_i}(\mathsf{H}(t_2||c_{x_i}||c_{\mathsf{K}_\mathsf{i}}))\rangle$, $\mathcal{ADV}$ needs to replace c_{x_i} with c'_{x_i} and $c_{\mathsf{K}_\mathsf{i}}$ with $c'_{\mathsf{K}_\mathsf{i}}$. By successfully performing this attack, $\mathcal{ADV}$ can control the outcome of a query to the **CSP** to manipulate **A**. To do this, $\mathcal{ADV}$ must:

- Generate a symmetric key $\mathsf{K}_{\mathsf{ADV}}$;
- Use $\mathsf{K}_{\mathsf{ADV}}$ to generate a series of ciphertexts c';
- Encrypt $\mathsf{K}_{\mathsf{ADV}}$ with pk_A or $\mathsf{pk}_{\mathsf{u_i}}$ to get $c'_{\mathsf{K}_{\mathsf{ADV}}}$;
- Tamper with m_2 in an indistinguishable way.

The first three tasks are straightforward to achieve. Additionally, substituting c_{x_i} with c'_{x_i} and $c_{\mathsf{K}_\mathsf{i}}$ with $c'_{\mathsf{K}_\mathsf{i}}$ in the non signature part of m_2 is trivial. However, both c_{x_i} and c'_{x_i} are included in the signature; hence, successfully substituting the terms is equivalent to forging u_i's signature. Given the EUF-CMA security of the signature scheme σ, this can only happen with negligible probability in the security parameter λ of σ.

Option 2 (2GML.Eval): When **CSP** returns the results of a secure classification to u_i via $m_3 = \langle t_3, c_{res}, \sigma_{CSP}(\mathsf{H}(t_3||c_{res}))\rangle$, $\mathcal{ADV}$ needs to replace c_{res} with c'_{res} for this instance of the attack to be successful. More specifically, since c_{res} is encrypted with $\mathsf{pk}_{\mathsf{u_i}}$, $\mathcal{ADV}$ simply encrypts fictitious results res' with $\mathsf{pk}_{\mathsf{u_i}}$ to produce c'_{res}. However, c_{res} is included in the signature part of m_3; hence tampering with m_3 requires forging the signature of **CSP**. Given the EUF-CMA security of the signature scheme σ, this can only happen with negligible probability.

Option 3 (3GML.Eval): When **A** initiates secure classification via $m_3 = \langle t_3, (c_w, c_b), \sigma_A(\mathsf{H}(t_3 \,||\, (c_w, c_b)))\rangle$, and **CSP** responds with the secure evaluation via $m_4 = \langle t_4, c_{res}, \sigma_{CSP}(\mathsf{H}(t_4||c_{res})\rangle$, $\mathcal{ADV}$ needs to replace (c_w, c_b) with (c'_w, c'_b) and c_{res} with c'_{res} to successfully perform this attack. The proof for this attack is similar to the one provided for the attack on 2GML.Eval with the only difference being that $\mathcal{ADV}$ attacks both m_3 and m_4 instead of just m_3. Using the same reasoning (i.e., the negligible probability of forging the signature), we conclude that $\mathcal{ADV}$ has a negligible probability of tampering with m_3 and m_4; hence this attack fails.

□

Proposition 6.2 (ML Model Unauthorized Access Attack Soundness). *Let f be a multi-layered ML model and* HE *semantically secure encryption scheme. Then $\mathcal{ADV}$ cannot successfully launch the ML Model Unauthorised Access attack for any of the GuardML protocols.*

Proof. To successfully launch the ML Model Unauthorized Access attack, $\mathcal{ADV}$ must collude or corrupt multiple entities in GuardML depending on the use case. More specifically, $\mathcal{ADV}$ can either attack 2GML by colluding with multiple users $(u_j)_{j\in S}$ where $S \subseteq [n]$ or attack 3GML by colluding with a user u_i and the **CSP**. To this end, we distinguish the attack into the following;

Option 1 (Attacking 2GML): Let f be a multi-layered ML model owned by **CSP**, and assume that $\mathcal{ADV}$ colludes with $n' = card(S)$ users. On completion of the secure evaluation phase (Figure 1), each user receives $res_i = f(x_i)$, where x_i is the input from the user. $\mathcal{ADV}$ successfully launches this attack if with the help of the n' colluding users, $\mathcal{ADV}$ can solve for f given $(res_j)_{j\in S}$ and $(x_j)_{j\in S}$. With the assumption that f is a multi-layered ML model, the likelihood of this attack is negligible.

Option 2 (Attacking 3GML): Let f be a multi-layered ML model owned by **A** with (c_w, c_b) as the HE encrypted parameters sent to **CSP** via $m_3 = \langle t_3, (c_w, c_b), \sigma_A(\mathsf{H}(t_3 \,||\, (c_w, c_b)))\rangle$. $\mathcal{ADV}$ successfully launches this attack if given a corrupt user u_i and a corrupt **CSP**, $\mathcal{ADV}$ successfully retrieves f in the form of (w, b). HE has been proven semantically secure;

hence, the likelihood of $\mathcal{ADV}$ decrypting the ciphertext (c_w, c_b) is considered negligible.

□

7 EXPERIMENTS SECTION

In this section, we report the process and results of building a privacy-preserving inference protocol on sensitive medical data (ECG) based on our 3GML construction subsection 7.1. Subsequently, we extensively evaluated the computational performance of the 2GML and 3GML protocols (subsection 7.2). For these evaluations, we utilized a dummy dataset where each data input was a vector of four random integers, and the weights and biases were also integer vectors of length four. Finally, to provide concrete evidence of the efficiency of our protocols, we compared the performance of our constructions against a basic BFV HE scheme subsection 7.3. For these experiments, our primary testbed was a commercial desktop with a 12th Generation Intel i7-12700 CPU with 20 cores and 32GB of RAM running on an Ubuntu 20.04 operating system. Furthermore, in all the evaluations, we utilized the SEAL cryptographic library[1] for basic HE operations and PASTA library[2] to implement the secure symmetric cipher. To ensure statistical significance, each experiment was repeated 50 times, with the average results considered to provide a comprehensive overview of the performance of each algorithm under evaluation.

7.1 PPML Application

In the first phase of our evaluations, we demonstrated the real-world applicability of our 3GML protocol by applying it to a PPML application with a sensitive heartbeat dataset to classify whether a heartbeat is subjected to heart disease or not. More specifically, we employed the MIT-BIH dataset [26], which is a dataset of human heartbeats obtained from 47 subjects from 1975 to 1979.

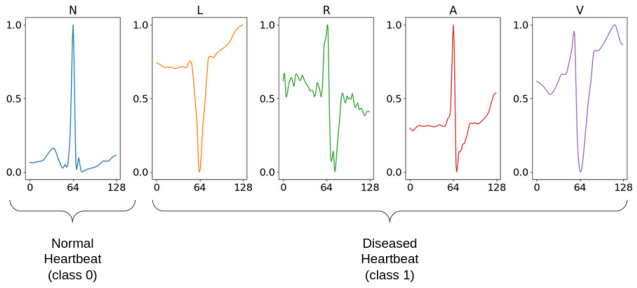

Figure 3: Example heartbeats from the MIT-BIH dataset.

7.1.1 MIT-BIH dataset. In total, the dataset contains 48 half-hour excerpts of two-channel ECG recordings. We used the processed ECG data from [1], which contains a train split and a test split, each comprising 13,245 ECG examples that belong to five classes: normal heartbeat (N), right bundle branch block (R), left bundle branch block (L), atrial premature contraction (A), ventricular premature contraction (V). Each ECG example is a float 1D time series signal of length 128 with values in the range of $[0, 1]$. We further grouped all ECG examples from the later four classes (L, R, A, V) into one super class called "diseased heartbeats" (Figure 3). By doing so, we simplified our problem into a binary classification problem. Based on the input ECG signal, we tried to classify if it is a normal or diseased heartbeat. As previously discussed, our HHE protocols are based on the BFV scheme, which only works with integer data and arithmetic. However, our ECG data are floating-point numbers in the range of $[0, 1]$, and normally, training neural networks also produces models with weights and biases in floating-point numbers. To this end, we first quantized the ECG data into 4-bit integer data with values in the range of $[0, 15]$ (Figure 4).

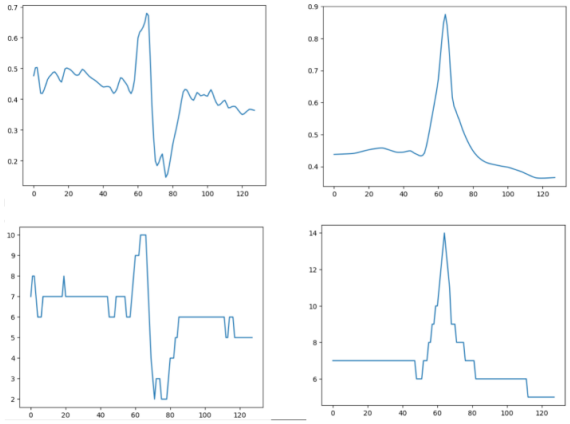

Figure 4: Quantizing ECG data into 4-bit integers. Top: Floating-point ECG data. Bottom: Corresponding quantized ECG data.

On preparing the data, we trained a simple neural network with one fully connected (FC) layer with a sigmoid activation function on the floating-point ECG dataset. The neural network can be written as $f(\boldsymbol{\theta}) = \text{sigmoid}(wx + b)$, where $\boldsymbol{\theta} = (w, b)$ are the model's weights and biases, and x is the training data. For the integer ECG data, we used the PocketNN framework [31] to train the same neural network in integer arithmetic. In both cases, we trained until the models' predictive accuracy on the test data split no longer improved. Training in floating-point arithmetic produced the best train accuracy of 89.36% at epoch 482 and the best test accuracy of 88.93% at epoch 500. With respect to training in integer arithmetic, we got the best train accuracy of 86.65% at epoch 42 and the best test accuracy of 87.06% at epoch 31. From these results, we observed that the test accuracy of the integer neural network was only 1.87% lower than the test accuracy produced by the floating-point neural network. Note that when training in integer arithmetic, we constrained the values of the weight and biases to be in the range of $[-2047, 2048]$, or $[-2^{11} + 1,\ 2^{11}]$. The reason for this is that the HHE protocol works in $\mathbb{Z}_q$ where $q = 2^{16} + 1$; therefore, all computation results need to be in the range of $[-2^{15} + 1,\ 2^{15}]$; otherwise they will be wrapped around by the modulo operation and produce incorrect predictions. As mentioned above, our ECG data are 4-bit integers and in the range of $[0, 2^4 - 1]$. In the 1 FC neural network, we needed to do one encrypted element-wise matrix multiplication between the data and the weights, and hence, if the weights are in $[-2^{11} + 1,\ 2^{11}]$, the results will be constrained in $[-2^{11} + 1,\ 2^{11}] \times 2^4 = [-2^{15} + 2^4,\ 2^{15}] \approx [-2^{15} + 1,\ 2^{15}]$, which is what we needed. After training and getting the trained integer model, we built ecgPPML – a privacy-preserving inference protocol

[1] https://github.com/microsoft/SEAL
[2] https://github.com/IAIK/hybrid-HE-framework

on the quantized integer ECG data based on our 3GML protocol construction and ran the experiments with the results reported below.

To begin our evaluations, we ran the encrypted inference protocol on a different number of examples from the test split, then compared the predictions with the ground-truth outputs to get the encrypted test accuracy. For each experiment, we also made inferences on plaintext ECG data in floating point and integer arithmetic to compare the results. Table 2 shows the encrypted and plaintext accuracies on a different number of data inputs. Overall, the accuracy of plaintext inference in integer arithmetic is very similar to encrypted inference accuracies. When the number of input examples is low (1-500), integer arithmetic and encrypted accuracies are comparable or even higher than plaintext floating-point inference. When the number of data inputs increases (1000-2000 examples), plaintext integer and encrypted inference produce slightly lower accuracies (0.5-0.8% lower). For 1000-2000 input data samples, encrypted inference had higher accuracy than plaintext integer inference but with a minimal margin (0.1-0.15%). We note that this is due to HE noise which helps make a few more correct predictions.

Data Inputs	Plaintext (Float)	Plaintext (Integer)	Encrypted
1	100 %	100 %	100 %
10	90 %	90 %	90 %
20	90 %	95 %	90 %
50	88 %	92 %	90 %
100	86 %	91 %	90 %
500	87 %	87.2 %	86.8 %
1000	87.9 %	87.3 %	87.4 %
2000	88.2 %	87.4 %	87.55 %

Table 2: Accuracy Analysis – ecgPPML

Subsequently, we evaluated the computational overhead of the ecgPPML protocol (Table 3). In the integer and float plaintext inference protocol, the client and analyst are not required to perform any computation as they outsource all their data and neural network model to the CSP. Therefore, in Table 3, the client and analyst only have a single column for the encrypted inference results. Looking at these results (i.e. across all data input examples for the encrypted inference protocol), we observed that the CSP is responsible for the most computational overhead (99% or even more), which increased linearly with the number of data inputs. Compared to plaintext inference, encrypted inference is more computationally expensive. However, encrypted inference for one data sample takes 12.18 seconds on a commercial desktop, which is a promising result. Furthermore, these experimental results align with our goal and expectation for the HHE protocol, as we want the client and analyst to do minimal work, and most of the computations take place in the CSP. Finally, we analyzed the communication cost of the ecgPPML protocol and reported the results in Table 4. For the communication between the client and the CSP, we observed that when the number of data inputs was low (1-500), the encrypted communication cost was very high compared to the plaintext costs due to the size of the HE ciphertext of the symmetric key (1.8 Mb). However, once the number of input examples increased, the size of the symmetrically encrypted data being sent to the CSP increased linearly, similar to the plaintext size, making this difference unimportant. There is no communication between the client and the analyst in plaintext protocols, while in the encrypted inference protocol, the client only transfers the HE public key to the analyst, which has a fixed size of 2.06 Mb. Overall, the communication cost for the client was minimal and increased linearly with the number of data inputs submitted to the CSP. The communication cost between the analyst and CSP was also minimal for plaintext inference since the plaintext weights, biases, and results were small. On the other hand, the majority of the communication cost for the encrypted inference protocol was incurred between the analyst and CSP. This increased cost is caused by the HE-encrypted output of the linear layer that needs to be sent from the CSP to the analyst. Hence, if the number of data inputs increases, the communication cost between the analyst and CSP will increase linearly. We observed that the communication cost for 2000 data input examples is 5548.21 Mb, or about 5 Gb of data, which is a reasonable result for today's internet bandwidth.

Data Inputs	Client	Analyst	CSP		
	Encrypted	Encrypted	Plaintext (float)	Plaintext (integer)	Encrypted
1	0.16	0.52	0	0.23	12.18
10	0.18	0.58	0	0.24	120.44
20	0.2	0.63	0	0.23	241.21
50	0.27	0.803	0	0.25	601.95
100	0.4	1.091	0	0.24	1212.38
500	1.31	3.38	0.05	0.24	6021.98
1000	2.46	6.2	0.1	0.25	12058.2
2000	4.8	11.92	0.2	0.27	24153.5

Table 3: Computation Analysis – ecgPPML. All numbers in seconds.

These experimental results show that our ecgPPML protocol produces comparable results in accuracy compared to inference on plaintext data. Furthermore, the CSP is responsible for most of the computation costs, and the majority of communication cost also occurs between the CSP and the analyst. These results align with our vision of using HHE for PPML applications and show the potential for HHE when applied in real-world PPML applications.

Data Inputs	Client - CSP			Client - Analyst	Analyst - CSP		
	Plaintext (float)	Plaintext (integer)	Encrypted	Encrypted	Plaintext (float)	Plaintext (integer)	Encrypted
1	0.0002	0.0002	1.8	2.06	0.0017	0.000734	72.46
10	0.002	0.002	1.8	2.06	0.0017	0.000734	97.11
20	0.005	0.005	1.81	2.06	0.0017	0.000734	124.51
50	0.012	0.012	1.81	2.06	0.0017	0.000734	206.692
100	0.029	0.029	1.83	2.06	0.0017	0.000734	343.643
500	1.1	1.1	2.9	2.06	0.0017	0.000734	1439.27
1000	2.3	2.3	4.1	2.06	0.0017	0.000734	2809.02
2000	4.6	4.6	6.4	2.06	0.0017	0.000734	5548.21

Table 4: Communication Analysis – ecgPPML. All numbers are in Megabytes (Mb).

7.2 Computational Analysis

This subsection focused on the computational performance of the core algorithms executed by each entity in the 2GML and 3GML protocols. More precisely, we measured the time taken to execute each HHE algorithm in each protocol phase by the responsible party. For both the 2GML.Setup and 3GML.Setup phases, we observed the time taken to generate a set of HHE keys at the user and the analyst, respectively. For implementation purposes, the HHE.KeyGen algorithm involved the generation of encryption parameters parms, secret key sk, public key pk, relinkey rk, and a Galois key gk, and took 243 milliseconds to execute. Subsequently, in the 2GML.Upload and 3GML.Upload phases, we measured the time taken to homomorphically encrypt a symmetric key K using the HE.Enc and the

time taken to execute the SKE.Enc for various inputs ranging from 1 to 300 (each input is an integer vector of length 4). HE.Enc took 7 milliseconds to run. For a single data input, SKE.Enc executed in 2 milliseconds and 600 milliseconds for 300 data inputs (Table 5).

Inputs	SKE.Enc	HHE.Decomp	HHE.Eval (2P)	HHE.Eval (3P)	HHE.Dec
1	2 ms	11.9 s	7 ms	0.038 s	3 ms
50	100 ms	599.1 s	350 ms	1.96 s	150 ms
100	200 ms	1197.7 s	700 ms	3.93 s	300 ms
150	300 ms	1794.5 s	1050 ms	5.77 s	450 ms
200	400 ms	2394.2 s	1400 ms	7.69 s	600 ms
250	500 ms	2989.2 s	1750 ms	9.61 s	750 ms
300	600 ms	3595.6 s	2100 ms	11.61 s	900 ms

Table 5: Computational Analysis

When evaluating the `2GML.Eval` phase; we measured the cost of executing the HHE.Decomp algorithm for various numbers of symmetric ciphertexts from 1 to 300, and the cost of executing the HHE.Eval algorithm for various homomorphic ciphertexts (1 to 300). For a single input, HHE.Decomp took 11.9 seconds, while HHE.Eval took 7 milliseconds. On the other hand, for 300 inputs, HHE.Decomp took 3595.6 seconds, while HHE.Eval took 2100 milliseconds (Table 5). When evaluating `3GML.Eval`, we first measured the performance of HE.Enc at **A** and then the performance of the HHE.Decomp and HHE.Eval algorithms on various number of symmetric ciphertext inputs from 1 to 300 at the **CSP**. HE.Enc took 16 milliseconds to execute, while the results for HHE.Decomp were similar to those from `2GML.Eval`. For a single input, HHE.Eval took 38 milliseconds to execute and 11.6 seconds for 300 inputs.

Phase	User	Server	Total
2GML.Setup	243 ms	–	243 ms
2GML.Upload	607 ms	–	607 ms
2GML.Eval	–	3597.7 s	3597.7 s
2GML.Classify	900 ms	–	900 ms

Table 6: Total Computation Cost – 2GML for 300 data inputs

Finally, in both `2GML.Classify` and `3GML.Classify` phases, we focused primarily on measuring the cost of HHE.Dec for a range of homomorphic ciphertext inputs from 1 to 300. For a single input, HHE.Dec ran in 3 milliseconds, and for 300 inputs, it ran in 900 milliseconds (Table 5). Table 6 and Table 7 provide the computational analysis of `2GML` and `3GML` protocols respectively for 300 inputs.

Phase	Analyst	User	Server	Total
3GML.Setup	243 ms	–	–	243 ms
3GML.Upload	–	607 ms	–	607 ms
3GML.Eval	16 ms	–	3607.21 s	3607.23 s
3GML.Classify	900 ms	–	–	900 ms

Table 7: Total Computation Cost–3GML for 300 data inputs

7.3 Comparison with plain BFV

To provide concrete evidence of the efficiency of our proposed construction, we implemented a plain BFV scheme with a similar architecture to `3GML` and compared the results. More precisely, we measured the performance of a plain BFV scheme, where a user continuously encrypts data input homomorphically before outsourcing them to the **CSP**. The same encryption parameters were used for all implementations. For these experiments, we only focused on comparing the total computational and communication costs of running the `3GML.Upload` phase of our protocol, with the cost of continuously using HE encryption in the plain BFV. We varied the number of data inputs from 1 to 300 (each data input is an integer vector of length 4).

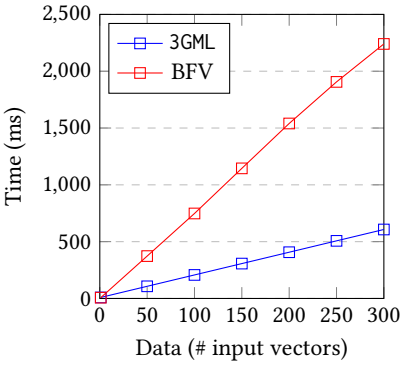

Figure 5: Computation Costs

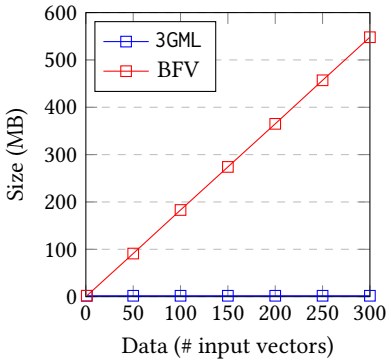

Figure 6: Communication Costs

For a single data input, `3GML.Upload` took 9 milliseconds to execute, while the plain BFV scheme took 7 milliseconds to perform a single HE encryption. It is worth noting that the plain BFV scheme is marginally faster for a single data value. However, this is due to the fact that `3GML.Upload` involves two operations (a symmetric encryption operation and an HE encryption operation), while the plain BFV scheme involves just one HE encryption operation. However, when the number of data values was increased to 300, `3GML.Upload` ran in 0.608 seconds, while the plain BFV scheme ran in 2.2 seconds. Figure 5 provides an overview of the computational comparison results obtained from this phase of our experiments. It is worth pointing out the fact that, in most cases, uploading just one single input is unrealistic since most PPML services will require a plethora of data to properly evaluate a problem. Subsequently, we compared the communication expenses by measuring the total size of transferable ciphertext data in bytes from a user u_i to **CSP**. Overall, `3GML.Upload` sent approximately 1.8 MB of ciphertext data for both a single input and 300 inputs. This is primarily because the size of a symmetric ciphertext is almost negligible as compared to that of a homomorphic ciphertext. The plain BFV, on the other

hand, sent approximately 1.82 MB of ciphertext data for a single data input and 547.8 MB for 300 data inputs. Figure 6 provides an overview of the comparison of the communication costs for 1 to 300 different inputs. From these results, it is evident that GuardML reduces the communication and computational burden of u_i and transfers them to **CSP**.

Open Science & Reproducible Research To support open science and reproducible research and provide other researchers with the opportunity to use, test, and hopefully extend our work, the source codes used for the evaluations have been made available online[3,4].

8 CONCLUSION

This paper is one of the first attempts to effectively use the novel concept of HHE to address the problem of privacy-preserving machine learning. We have provided a realistic solution that carefully considers the vagaries of PPML. The designed approach is able to carefully balance ML functionality and privacy so as to allow the use of PPML techniques in a wide range of areas, such as pervasive computing, where, in many cases, the underlying infrastructure presents certain inbuilt limitations. By using HHE, we managed to overcome the main difficulties of PPML application in real-life scenarios, where the majority of data is collected and processed by constraint devices. Certain that the future of cryptography goes hand in hand with ML, we believe we have made the first step towards implementing PPML services with strong security guarantees, which operate efficiently in a wide range of architectures.

ACKNOWLEDGMENTS

This work was funded by the HARPOCRATES EU research project (No. 101069535) and the Technology Innovation Institute (TII), UAE, for the project ARROWSMITH.

REFERENCES

[1] Sharif Abuadbba, Kyuyeon Kim, Minki Kim, Chandra Thapa, Seyit A Camtepe, Yansong Gao, Hyoungshick Kim, and Surya Nepal. 2020. Can we use split learning on 1d cnn models for privacy preserving training?. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*.

[2] Ahmad Al Badawi, Chao Jin, Jie Lin, Chan Fook Mun, Sim Jun Jie, Benjamin Hong Meng Tan, Xiao Nan, Khin Mi Mi Aung, and Vijay Ramaseshan Chandrasekhar. 2020. Towards the alexnet moment for homomorphic encryption: Hcnn, the first homomorphic cnn on encrypted data with gpus. *IEEE Transactions on Emerging Topics in Computing* 9, 3 (2020), 1330–1343.

[3] Alexandros Bakas, Eugene Frimpong, and Antonis Michalas. 2022. Symmetrical Disguise: Realizing Homomorphic Encryption Services from Symmetric Primitives. In *International Conference on Security and Privacy in Communication Systems*. Springer, 353–370.

[4] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*. Springer.

[5] Zvika Brakerski. 2012. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*. Springer, 868–886.

[6] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. 2018. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. *Journal of Cryptology* 31, 3 (2018), 885–916.

[7] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 409–437.

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. 2016. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*. Springer, 3–33.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91.

[10] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. 2021. Transciphering framework for approximate homomorphic encryption. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 640–669.

[11] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. 2023. Towards Case-Optimized Hybrid Homomorphic Encryption: Featuring the Elisabeth Stream Cipher. In *Advances in Cryptology – ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security* (Taipei, Taiwan). Springer-Verlag, Berlin, Heidelberg, 32–67.

[12] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. 2023. Pasta: a case for hybrid homomorphic encryption. *Transaction on Cryptographic Hardware and Embedded Systems 2023 Issue 3* (2023).

[13] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive* (2012).

[14] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Stanford university.

[15] Craig Gentry, Shai Halevi, and Nigel P Smart. 2012. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*. Springer.

[16] Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Jooyoung Lee, and Mincheol Son. 2022. Rubato: Noisy Ciphers for Approximate Homomorphic Encryption. In *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*. Springer, 581–610.

[17] Ehsan Hesamifard, Hassan Takabi, Mehdi Ghasemi, and Rebecca N Wright. 2018. Privacy-preserving machine learning as a service. *Proc. Priv. Enhancing Technol.* 2018, 3 (2018), 123–142.

[18] Tanveer Khan, Alexandros Bakas, and Antonis Michalas. 2021. Blind faith: Privacy-preserving machine learning using function approximation. In *2021 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 1–7.

[19] Tanveer Khan and Antonis Michalas. 2023. Learning in the Dark: Privacy-Preserving Machine Learning using Function Approximation. (2023).

[20] Tanveer Khan, Khoa Nguyen, and Antonis Michalas. 2023. A More Secure Split: Enhancing the Security of Privacy-Preserving Split Learning. In *Nordic Conference on Secure IT Systems*. Springer, 307–329.

[21] Tanveer Khan, Khoa Nguyen, and Antonis Michalas. 2023. Split Ways: Privacy-Preserving Training of Encrypted Data Using Split Learning. In *2023 Workshops of the EDBT/ICDT Joint Conference, EDBT/ICDT-WS 2023, 28 March 2023*. CEUR-WS.

[22] Tanveer Khan, Khoa Nguyen, Antonis Michalas, and Alexandros Bakas. 2023. Love or Hate? Share or Split? Privacy-Preserving Training Using Split Learning and Homomorphic Encryption. In *2023 20th Annual International Conference on Privacy, Security and Trust (PST)*. IEEE Computer Society, 1–7.

[23] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. 2022. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access* 10 (2022), 30039–30054.

[24] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. 2020. Glyph: Fast and accurately training deep neural networks on encrypted data. *Advances in Neural Information Processing Systems* 33 (2020), 9193–9202.

[25] Pierrick Méaux, Claude Carlet, Anthony Journault, and François-Xavier Standaert. 2019. Improved filter permutators for efficient FHE: Better instances and implementations. In *International Conference on Cryptology in India*. Springer.

[26] George B Moody and Roger G Mark. 2001. The impact of the MIT-BIH arrhythmia database. *IEEE engineering in medicine and biology magazine* 20, 3 (2001), 45–50.

[27] K Nguyen, T Khan, and A Michalas. 2023. Split Without a Leak: Reducing Privacy Leakage in Split Learning. In *19th EAI International Conference on Security and Privacy in Communication Networks (SecureComm'23)*. Springer.

[28] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. 1978. On data banks and privacy homomorphisms. *Foundations of secure computation* 4, 11 (1978), 169–180.

[29] Amartya Sanyal, Matt Kusner, Adria Gascon, and Varun Kanade. 2018. TAPAS: Tricks to accelerate (encrypted) prediction as a service. In *International Conference on Machine Learning*. PMLR, 4490–4499.

[30] Sinem Sav, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. 2021. POSEIDON: privacy-preserving federated neural network learning. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

[31] Jaewoo Song and Fangzhen Lin. 2022. PocketNN: Integer-only Training and Inference of Neural Networks via Direct Feedback Alignment and Pocket Activations in Pure C++. *Proceedings of tinyML Research Symposium* (2022).

[3] https://github.com/iammrgenie/hhe_ppml

[4] https://github.com/khoaguin/PocketHHE