Neural Computing and Applications (2022) 34:15555–15573
https://doi.org/10.1007/s00521-022-07202-8

ORIGINAL ARTICLE

Fast homomorphic SVM inference on encrypted data

Ahmad Al Badawi[1] · Ling Chen[2] · Saru Vig[1]

Received: 12 October 2021 / Accepted: 22 March 2022 / Published online: 25 April 2022
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2022

Abstract
Kernel methods are popular machine learning methods that provide automated pattern analysis of raw datasets. Of particular interest is Support Vector Machines that are used to solve supervised machine learning problems in many areas such as business, finance and healthcare. Nowadays, complex computations and data analytic tasks can be outsourced to specialized third parties. However, data owners might be reluctant to share their data especially when it includes sensitive information. Therefore, a need for privacy-preserving machine learning applications cannot be overstated. We present FHSVM: a **F**ast **H**omomorphic evaluation of non-linear **SVM** prediction on encrypted data using Fully Homomorphic Encryption. We provide design, implementation and several algorithmic and architectural optimizations such as novel packing strategies and parallel implementation to achieve real-time private prediction. We employed the CKKS FHE scheme to implement FHSVM under 128-bit security level. We evaluated FHSVM on a contemporary real-world large dataset compiled for anti-money laundering tasks in Bitcoin transactions. Empirical analysis demonstrates that homomorphic SVM prediction can be performed in 1.25 s on multi-core CPU platforms. In addition, FHSVM shows zero accuracy loss when compared to the non-privacy-preserving implementation. This shows that FHSVM is both computationally secure and fully utilizes the data.

Keywords Privacy-preserving computing · Data privacy · Homomorphic encryption · Support vector machines

1 Introduction

Support Vector Machine (SVM) is a widely used supervised machine learning algorithm in the fields of data mining and data science. Its uses are spread across various sectors such as marketing [1], finance [2], and healthcare [3] for data analytic tasks. Developing and deploying SVM-based solutions goes through two main phases: training an SVM model and using the model for inferencing. The training phase requires a dataset that is used to develop the SVM network by fine-tuning its parameters and outputs the SVM model as a result. In the deployment/testing phase, the generated SVM model is used to make inferences on newly unseen data samples.

Training an effective SVM model for certain applications is not a straightforward task. For instance, data collection and engineering might require expert domain knowledge to identify the relevant features that impact a response variable. It might also require massive computing capacity especially in scenarios that include big datasets. Moreover, model creation and validation includes scientific and artistic proficiency that are not commonly known. Developing these capabilities in-house can be very expensive and waste of resources especially for resource-limited clients. Due to the above challenges, users revert to cloud computing and adopt what is broadly known as Machine Learning as a Service (MLaaS) solutions [4]. In fact, several ML-solution infrastructures have been developed by cloud service provides such as Amazon, Google, Microsoft and IBM to provide MLaaS for clients at affordable subscription fees. In this paradigm, the cloud hosts an effective model that can be used for making

✉ Ahmad Al Badawi
aalbadawi@ra.ac.ae

Ling Chen
Ling_Chen@i2r.a-star.edu.sg

Saru Vig
Saru_Vig@i2r.a-star.edu.sg

1 Homeland Security, Rabdan Academy, Dhafeer St, 114646 Abu Dhabi, UAE

2 Cybersecurity, Institute for Infocomm Research, 1 Fusionopolis Way, Singapore 138632, Singapore

inferences on data points provided by the subscribed users. While this solution might overcome most of the challenges mentioned above, there are serious privacy concerns that arise when the data samples provided by the users are highly sensitive such as bio-metric traits, medical records, and financial information. This has motivated several research efforts to tackle the increasing privacy concerns of MLaaS and devising privacy-preserving machine learning inferencing technologies.

One of the most promising privacy-preserving technologies is the Fully Homomorphic Encryption (FHE). FHE gives the possibility of performing meaningful arithmetic operations (e.g. addition/multiplication) on encrypted data [5]. This can enable an untrusted third-party cloud-based server to perform computations on encrypted data without compromising the privacy of sensitive information. In theory, it is possible to apply machine learning algorithms on encrypted data and revert back calculated data/predictions in encrypted form, but there are computational limitations leaving with limited practicality. Machine learning algorithms generally require complex computations rendering FHE too slow to realize real-time prediction. As such, there is still a need to identify feasible algorithms with optimizations to make them adaptable and practical for FHE. Our privacy-preserving SVM prediction relies heavily on FHE and several optimizations to achieve an efficient solution.

The literature is abundant with several studies on realizing privacy-preserving SVM solutions. Broadly speaking, two different solution methodologies can be identified, namely, cryptographic and non-cryptographic methods. In this work, we focus on cryptographic methods and stress on the usage of two interesting privacy-preserving technologies: FHE and Multi-Party Computation (MPC). As MPC is maturer than FHE, the majority of the early proposed works employed MPC protocols. Recently, FHE started to gain more popularity in implementing privacy-preserving machine learning methods including SVMs. Other privacy-preserving methods such as Block Scrambling-based Encryption (BSE) schemes have also been employed. We briefly review the state of the art on privacy-preserving SVM frameworks focusing on those that employ HE.

BSE schemes, proposed for Encryption-then-Compression (EtC) systems, were demonstrated to be applicable for face recognition problems using SVMs in [6]. In this framework, the image is divided into non-overlapping blocks and block-based pre-processing is applied. The authors propose a similar solution using a random unitary transformation. The generated templates using the Random Unitary Matrix method were shown to be performant secure SVM algorithms. Despite the high efficiency of these methods, their security guarantees are questionable as each block in encrypted images with EtC, has a significant correlation with the corresponding block in the original images [7].

Moving on to HE solutions, the first known privacy-preserving protocol for SVM using Pailler homomorphic encryption was presented in [8]. The authors developed an interactive protocol that hides the client's input data and the server side's classification parameters for binary and multi-class classification. Even though the accuracy was shown to be similar to the plaintext version, the protocol is computationally intensive requiring 149.82 s for a small dataset with only 213 data points in addition to the interaction required among the parties during computation. Optimizing the inference time is one of the main factors we considered in our purposed solution. Another relevant FHE-based study was conducted in [9]. The authors performed image classification using SVM with polynomial kernels using Somewhat Homomorphic Encryption (SHE). They employed an output masking methodology to protect the privacy of the SVM parameters, which are used in plaintext. Their experimental evaluation on CIFAR-10 'car'/'not car' identification task [10], using an SVM with a polynomial kernel of degree 2, and 1191 support vectors of length 29 features required 8.97 s for prediction. We show later how our framework can perform SVM prediction for a very large real-world dataset with 165 features, 200K datapoints and 7780 support vectors in 1.25 s. Besides, their framework suffered from prediction accuracy loss due to the use of an integer-based FHE scheme (BGV [11]) unlike our framework that employs a real-numbers-friendly FHE scheme (the Cheon-Kim-Kim-Song (CKKS) scheme [12]).

More recently, the authors in [13] proposed an SVM training algorithm with FHE for binary classification tasks. They devised an adapted FHE friendly gradient descent method for the least square problem optimization. The algorithm has been implemented via CKKS [12] and tested on eight different datasets. The practicality of their framework is debatable as the authors used small datasets (with 6–60 features and less than 100 data points) for training requiring on average 30–60 s per single gradient-decent iteration. Moreover, their SVM models severed from low prediction accuracy ranging from 64 to 98% due to the low number of training iterations. It should be noted that the prediction latency was not reported in their framework. A more relevant framework for privacy-preserving clustering with SVM and FHE was proposed in [14]. The authors implemented an FHE-friendly SMV clustering algorithm and evaluated it on six different datasets. Their experimental evaluation showed that the framework did not scale as the size of the dataset is increased despite the employment of low-rank datasets of only (2–3) features. For instance, their system required 27.24 s and 220.38 s on the Hepta (3 features and 212 data points) and the Chainlink (3 features and 1000 data points) datasets, respectively.

With the advances in security with machine learning algorithms, their application to the medical industry has attracted interest [15, 16]. The authors in [17] proposed a secure pre-diagnosis system (eDiag) using a non-linear SVM kernel. Although accurate, the computational complexity is dependent on the number of support vectors. Thus, they choose lightweight multiparty random masking and polynomial aggregation methods to design the framework. There has been other research on employing MPC protocols as well [18, 19]. MPC solutions suffer, by design, from several interactions between the client and server and may require high bandwidth for complex computations. Our framework using FHE is conceptually simpler and more efficient for privacy-preserving SVM prediction tasks in terms of latency and bandwidth.

Our main contribution in this paper is a framework for privacy-preserving Fast Homomorphic SVM inference. We present FHSVM: a Fast Homomorphic SVM for secure evaluation of SVM models on encrypted data. FHSVM integrates different technologies such as machine learning and FHE to execute machine learning tasks on highly confidential data. A potential system deployment of FHSVM is depicted in Fig. 1. We assume that the HE evaluator has a pre-learned SVM model and is willing to provide SVM predictions on input instances provided by the end-user at some subscription rate. The SVM model is trained on plaintext data and it remains in unencrypted form. This can be done by either training the model on prehistorical or public data. The end-user provides encrypted data ENC $(\boldsymbol{x})$ to the HE evaluator. The latter runs FHSVM on ENC $(\boldsymbol{x})$ and generates an encrypted result ENC (SVM$(\boldsymbol{x})$). Eventually, the end-user who has access to the secret key can decrypt and learn SVM $(\boldsymbol{x})$.

Although FHE has been realized in practice more than a decade ago and undergone many optimizations towards improving its efficiency, the technology is still far from being an out-of-the-box tool for adoption in practical applications. Issues related to the choice of the suitable FHE scheme, the encryption parameters, noise management (more on that later), multiplicative depth optimization, vectorized execution mode, data expansion, and key management need to be addressed by FHE experts to realize efficient and practical privacy-preserving applications with FHE. In this work, we employ the CKKS FHE scheme that supports natively homomorphic computations on real numbers. Our second major contribution in this work is showing how to realize an efficient privacy-preserving algorithm for SVM prediction on encrypted samples (FHSVM) by utilizing several algorithmic and architectural optimizations such as intensive packing and parallel algorithms towards improving efficiency. To evaluate our framework, we have implemented FHSVM and evaluated it on a publicly available large (more than 200k instances) real-world dataset (the Elliptic Dataset [20]) for Anti-Money Laundering (AML) in Bitcoin transactions. With the proposed optimization methods of packing and parallel implementation, we were able to reduce computation time to about 1.25 s.

The main strengths and challenges of selected cryptographic solutions of privacy-preserving SVM on encrypted data are described in Table 1. We focus on several dimensions related to privacy-preserving SVM such as whether the model parameters were encrypted alongside the input samples, accuracy loss exhibited in the proposed solution, the guaranteed security level measured in bits, performance aspects in terms of communication and computation overhead, statistics of the datasets used in the evaluation and type of kernel functions supported by the solution. It can be noticed that no winning solution exists due to the complexity of the problem at hand and the tradeoff between security, utility and cost.

The rest of the paper is organised as follows. Section 2 provides some background on FHE, CKKS, and the SVM prediction algorithm. Our methods used to construct FHSVM are presented in Sect. 3. Section 4 shows our experimental methodology and security analysis. The empirical evaluation of FHSVM is presented in Sect. 5. Further discussion on potential use cases of FHSVM and security analysis is provided in Sect. 6. Finally, Sect. 7 concludes the work.

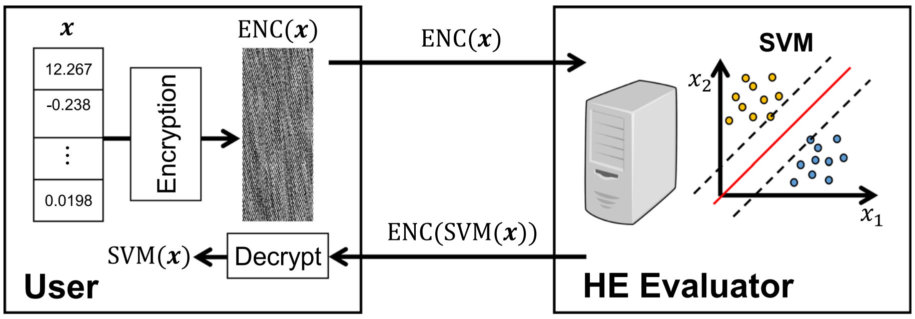

Fig. 1 Deployment of privacy-preserving SVM evaluation with HE

Table 1 Strengths and limitations of some cryptography-based approaches for privacy-preserving SVM with encrypted data

Method	Encrypted model	Accuracy loss	Security level (bits)	Comm. overhead	Comp. overhead	Dataset Statistics			Kernel type
						# Samples	# Features	# Classes	
[8]	No	Negligible	112	High	Low	150–768	4–2601	Multi	Polynomial
[9]	No	Low	80	Low	Mid	30,000	57	Binary	Polynomial
[13]	Yes	Low	?	Low	High	100	6–60	Binary	Polynomial RBF
[14]	No	Low	128	Low	Mid	212–1000	2–3	N.A.	Gauss-ian
Ours	No	Negligible	128	Low	Low	200,000	166	Multi	Polynomial

N.A. stands for non-applicable as the problem studied in that work was clustering, Comm. and Comp. stand for communication and computational, RBF stands for Radial Basis Function, and (?) stands for unknown

2 Background

In this section, we review the basic notions our work builds on. We start by describing the notations used, the basic primitives of the CKKS scheme and introduce the SVM machine learning algorithm.

2.1 Notations

Capital and small letters are used to refer to sets and elements of sets, respectively. We denote the sets of integers, reals and complex numbers by $\mathbb{Z}, \mathbb{R}$, and $\mathbb{C}$, respectively. Matrices and vectors are denoted by bold capital and small letters, respectively. The dot product between two vectors is denoted by $\langle \cdot \rangle$. We use the symbols $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$, and $\lfloor \cdot \rceil$ to refer to the round down, round up and round to nearest integer functions, respectively. For an integer a, $|a|_q$ denotes the remainder of a when divided by q. If a is a polynomial, the reduction is performed on each coefficient. The symbol $a \longleftarrow S$ refers to sampling an element a from the set S.

2.2 FHE

An FHE scheme is a cryptographic method that allows a not-necessarily trusted party to compute on encrypted data without access to the decryption key [5]. In simple terms, FHE maps plaintext messages $\mathcal{P}$ into ciphertexts $\mathcal{C}$ without affecting the algebraic structure between $\mathcal{P}$ and $\mathcal{C}$. The structure is mainly preserved under addition and multiplication. Concretely, denoting the FHE encryption procedure by ENC, let $m_1, m_2 \in \mathcal{P}$ be two plaintext messages, then the following property holds: $\mathsf{ENC}(a) \oplus \mathsf{ENC}(b) = \mathsf{ENC}(a+b)$ and $\mathsf{ENC}(a) \odot \mathsf{ENC}(b) = \mathsf{ENC}(a \cdot b)$, where $\oplus$ and $\odot$ are homomorphic addition and homomorphic multiplication, respectively. Since addition and multiplication in $\mathbb{Z}_2$ are Turing complete, FHE allows one to compute arbitrary functions with encrypted operands. The result itself is also encrypted and can only be decrypted successfully by the owner of the decryption key.

While working with FHE, one needs to keep in mind the following notions. Firstly, the magnitude of noise included in ciphertexts. The security of most FHE schemes relies on the Learning With Errors (LWE) problem and its ring variant Ring-LWE (RLWE). These problems use noise components that are sampled from certain distributions as part of their operation. The encryption procedure in FHE inherently introduces some noise in freshly encrypted messages. As we compute, the magnitude of this noise grows at a certain rate. The noise growth due to addition is much slower (almost negligible) when compared to that due to multiplication. Note that the noise should be maintained under a certain level so that the decryption procedure can remove it and retrieve the underlying plaintext message successfully.

The second important notion is the multiplicative depth of the target circuit which can be defined as the largest number of multiplications along any path in the circuit. If the multiplicative depth is known in advance and quite small, a levelled FHE scheme [11] can be used. These schemes allow a limited number of multiplications but can be much faster than conventional FHE schemes. There are several instantiations of FHE schemes in the literature [5, 12, 21–29] that differ in the underlying structure, capabilities and performance. In this work, we adopt a Residual Number System (RNS) variant of the CKKS levelled FHE scheme [30].

2.3 CKKS

In this section, we describe an RNS variant of the *levelled* CKKS scheme [30]. The scheme is parameterized by the ring dimension n that is a power of 2 integer, $L \in \mathbb{Z}$ the multiplicative depth, the ciphertext coefficients $q_L > q_{L-1} > \cdots > q_1$ all $\in \mathbb{Z}$ and the plaintext scale factor $p \in \mathbb{Z}$. The scheme works in the polynomial rings $R_{q_l} = \mathbb{Z}_{q_l}[X]/(X^n + 1)$, where $1 \le l \le L$ is the level number. A message is first scaled by p before encryption and the generated ciphertext stars at level L. Multiplying two ciphertexts $p \cdot m_1$ and $p \cdot m_2$ results in squaring the scale factor $p^2 m_1 \cdot m_2$. The scheme provides a rescale operation to divide approximately by $1/p$ and maintain the precision fixed. The CKKS scheme includes the following primitives:

- INIT: this procedure takes as input the security level λ, and maximum multiplicative depth L, initialize the scheme by setting the ring dimension n, 2 uniform random distributions: $\mathcal{X}_{key}$ over R_2 and $\mathcal{X}_{q_L}$ over R_{q_L}, and a discrete Gaussian distribution $\mathcal{X}_{err}$ with zero mean and standard deviation σ over R_{q_L}.
- KEYGEN: this procedure takes the system parameters and compute: 1) the secret key $s \leftarrow \mathcal{X}_{key} \in R_{q_L}$, and 2) the public key $(a, b) \in R^2_{q_L}$, s.t., $a \leftarrow \mathcal{X}_{q_L}$ and $b = -as + e$ with $e \leftarrow \mathcal{X}_{err}$.
- ENCODE(z, p): this procedure takes a vector of complex numbers $\mathbf{z} \in \mathbb{C}^{n/2}$ and precision p, and returns $\mu = \lfloor \mathsf{IDFT}(p \cdot z) \rceil \in R$, where IDFT is the Inverse Discrete Fourier Transform.
- ENC (μ): this procedure takes a plaintext message (μ) as input. It samples $u \leftarrow \mathcal{X}_{q_L}$ and $e_0, e_1 \leftarrow \mathcal{X}_{err}$ and returns the ciphertext $ct = (c_0, c_1) = (au + \mu + e_0, bu + e_1) \in R^2_{q_L}$.
- DEC(ct): this procedure takes a ciphertext $ct \in R^2_{q_l}$ and returns $\mu = c_0 + s \cdot c_1 \in R_{q_l}$.
- DECODE(μ, p): this procedure takes a plaintext message $\mu \in R$ and precision p as inputs. It returns $v = \mathsf{DFT}(\mu/2^p) \in \mathbb{C}^{n/2}$, where DFT is the Discrete Fourier Transform.
- HADD (ct_0, ct_1): homomorphic addition takes as input two ciphertexts and returns $ct^+ = ct_0 + ct_1 \in R^2_{q_l}$. Note that the input ciphertexts must be at the same level l.
- HMUL $(ct_0 = (c_{00}, c_{01}), ct_1 = (c_{10}, c_{11}))$: homomorphic multiplication takes two ciphertexts and computes $ct^\times = (c_{00}c_{10}, c_{00}c_{11} + c_{01}c_{10}, c_{01}c_{11}) \in R^3_{q_l}$. Note that the input ciphertexts are at the same level l. This operation is usually followed by a relinearization procedure that is used to reduce the number of components in $ct^\times$ from 3 to 2 elements $\in R^2_{q_l}$.
- HADDPLAIN(ct, pt): this procedure adds a ciphertext $ct = (c_0, c_1) \in R^2_{q_l}$ and a plaintext $pt \in R$. It computes the summation ciphertext using $ct^+ = (c_0 + pt, c_1) \in R^2_{q_l}$.
- HMULPLAIN(ct, pt): this procedure multiplies a ciphertext $ct = (c_0, c_1) \in R^2_{q_l}$ with a plaintext $pt \in R$. It computes the product ciphertext $ct^\times = (c_0 \cdot pt, c_1 \cdot pt) \in R^2_{q_l}$.
- RESCALE (ct, l'): this procedure scales an input ciphertext at level l and $l' = l - 1$ by computing $ct' = \lfloor q_{l'}/q_l \cdot ct \rceil \in R_{q_{l'}}$.
- ROTATE $(ct, \pi \in \mathbb{Z})$: this procedure can be used to rotate the encrypted vector in a ciphertext. The direction of the rotation (left or right) depends on the sign of π. ROTATE computes $ct' = (c_0(X^\pi), c_1(X^\pi))$. Note that ct' can only be decrypted by a rotated version of the secret key $s' = s(X^\pi)$. However, using a rotation key that is part of the public key, a procedure known as key switching can be used to make ct' decryptable under the original, "unrotated", secret key s.

2.4 SVM

2.4.1 Training

There are a number of hyperplanes that can be used to classify data points into different classes. For a binary classification problem in 2-D data points, the hyperplane is a line. The SVM algorithm aims to maximize the distance between the hyper-planes and any data point so future data points can be classified with more confidence. This is done by finding the 'widest margin' between the two classes and placing the hyper-plane in the middle of this margin. Training involves running this algorithm on the feature vectors of the data points for which labels have already been assigned. This selection is done in lines 2–6 in Algorithm 1. It should be noted that, kernel functions can be used to enable SVM to operate in a high dimensional space. For our use case, we choose a polynomial kernel for the training phase with degree d.

2.4.2 Testing

To test an unclassified input feature vector $\boldsymbol{x}$ we evaluate the decision function in Eq. (1), where

$$c = \operatorname{sign}\left(\sum_{i \in |S|} \alpha_i y_i K(\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle) + b\right) \tag{1}$$

- $\boldsymbol{x}_i \in \mathbb{R}^m$ are the support vectors in the set S
- m is the number of features in input and support vectors
- b is the model intercept
- $\alpha_i \in \boldsymbol{\alpha}$ is the i-th Lagrange multiplier
- $y_i \in \boldsymbol{y}$ is the class of the i-th support vector

– $K(\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle)$ is the polynomial kernel function of degree = d, with γ as a scaling factor = (1/no. of features) evaluated on the dot product $\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle$ as $K(\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle) = (r + \gamma\langle \boldsymbol{x}_i, \boldsymbol{x} \rangle)^d$, where r is a scalar independent from the kernel function.

Equation 1 will be evaluated with encrypted $\boldsymbol{x}$ for FHSVM and the other parameters are obtained in clear text from SVM training done in Algorithm 1.

3 Fast homomorphic SVM prediction

In this section, we present the system components and methods employed to construct the FHSVM algorithm for enabling privacy-preserving prediction of nonlinear SVMs on encrypted data. We start by describing the threat model assumed for using FHSVM. Next, we briefly describe the training phase that is performed on data in the clear. Lastly, a detailed description of the homomorphic prediction phase of FHSVM is presented.

3.1 Threat model

In FHSVM, the HE evaluator is assumed to own a pre-learned SVM model that has been trained on an unencrypted dataset. The HE evaluator adopts a Software as a Service (SaaS) service model and provides an online SVM prediction as a service to end-users who provide their input vectors in an encrypted form using CKKS. Since the CKKS scheme is proven to be IND-CPA secure [12], the HE evaluator cannot infer anything from the encrypted input except maybe its length. Similar to other FHE applications, we assume that the HE evaluator is semi-honest, i.e., it follows the protocol utterly step for step while trying to infer as much as possible from the data it manipulates. Therefore, the HE evaluator is trusted to do the expected computation but it cannot decrypt to reveal the data. Since the homomorphic evaluation of SVM prediction is entirely performed while the data is encrypted, we can ensure that the confidentiality of the data cannot be compromised. As such, FHSVM is considered as secure as the security level of the employed CKKS scheme against semi-honest HE evaluators.

3.2 Phase I: Data pre-processing

Since SVM is a supervised machine learning algorithm, it expects an annotated dataset as input. We assume that the features in the dataset are all numeric. Our framework does not support categorical/nominal nor textual data. However, standard conversion methods from categorical data to numerical data such as integer encoding and one-hot encoding can be used to overcome this limitation. This is not a major limitation of FHSVM since several standard non-privacy-preserving machine learning algorithms typically follow this approach [31].

3.3 Phase II: Training SVM in plaintext

In the second phase, we train an SVM model in the plaintext domain. This step is independent of the FHE context and can be done via conventional machine learning frameworks while applying state-of-the-art optimizations to ensure optimum performance in terms of classification accuracy and evaluation overhead. This phase requires us to fix a dataset as input to the SVM learning algorithm. Also, there are a few hyperparameters that need to be fixed as shown in Algorithm 1. In this work, we adopt the scikit-learn machine learning module in python. While this module supports several kernel functions, the current version of FHSVM supports linear and polynomial kernels.

One particular requirement FHSVM imposes is the employment of polynomial kernel functions. This is due to the fact that evaluating polynomials on encrypted data in CKKS is naturally supported via homomorphic addition and multiplication operations. Moreover, it is of paramount importance to optimize the degree of the kernel polynomial to reduce the multiplicative depth of the SVM prediction computation. This has a large impact on the size of CKKS parameters, encryption keys and ciphertext sizes, computational overhead, security level, and correctness or the precision of the computation. We note that FHSVM can be adapted to support other kernel functions such as Radial Basis Functions (RBFs). These functions however need to be approximated as polynomials. Another way of evaluating these functions is via Look-up Table (LUT) search, but this might be more expensive to do in CKKS.

While training the SVM model in plaintext, one can still use standard machine learning evaluation practices such as precision, recall, prediction accuracy, Receiver Operating Characteristic (ROC) curves, and F scores to evaluate and improve the SVM model. Note that other than employing a low-degree polynomial kernel, FHSVM is oblivious to which learning algorithm is used in training. Moreover, advanced training optimizations such as weight initialization, and regularization can be applied without affecting FHSVM workflow. Once this phase is complete, the SVM model parameters $(S, \boldsymbol{\alpha}, \boldsymbol{y}, K, b)$ are extracted to be used in the third phase, homomorphic prediction of SVM on encrypted data.

Lastly, an important design aspect of FHSVM is that it natively supports binary classification problems, i.e., 2-class problems. In addition, it is suitable for single-label classification problems, i.e., assigning one label to each input instance. If the classification problem at hand includes multiple classes or multiple labels per instance are desired, some out-of-the-box machine learning methods can be used such as one vs. all classifiers and using a separate model for each label to adapt FHSVM to these problems.

```
Algorithm 1: SVM training with scikit-learn for binary classification tasks
  Input: Classes vector $\boldsymbol{c}$, matrix $\boldsymbol{F}$ comprising the feature vectors and
         polynomial kernel function with degree $d$
  Output: $svc$: SVM model parameters $S, \boldsymbol{\alpha}, \boldsymbol{y}, K, b$
1 $j = 0$
2 for $i \leftarrow 0$ to total number of transactions do
3     if $\boldsymbol{c}_i \in \{0, 1\}$ then
4         $Y_j = \boldsymbol{c}_i$
5         $\boldsymbol{X}_j = \boldsymbol{F}_i$
6         $j{+}{+};$
7 $\boldsymbol{X\_train}, Y\_train = \text{TRAIN\_TEST\_SPLIT}(\boldsymbol{X}, Y, test\_size = 0.25)$
8 $svc = \text{SVC}(\text{kernel} = poly, \text{degree} = d)$
9 $svc = \text{svc.fit}(\boldsymbol{X\_train}, Y\_train)$
```

3.4 Phase III: Homomorphic prediction of SVM on an encrypted feature vector

This is the core phase of FHSVM in which we evaluate the decision function in Eq. (1) homomorphically. A flowchart of FHSVM protocol for conducting this task is depicted in Fig. 2. FHSVM requires cooperation between the user (who has the sensitive data point) and the HE evaluator (who has the SVM model) to do the SVM inference securely—Step 0. The user selects suitable CKKS encryption parameters, instantiates the cryptographic context and generates the encryption keys. The cryptographic context alongside the public-key material are sent to the evaluator to be used in the homomorphic evaluation of the SVM function—Steps 1–2. The evaluator can now encode the SVM parameters using the cryptographic context and wait for input encrypted data—Step 3. The user encodes and encrypts the input data point **x** and sends the resultant ciphertext to the evaluator—Steps 4–5.

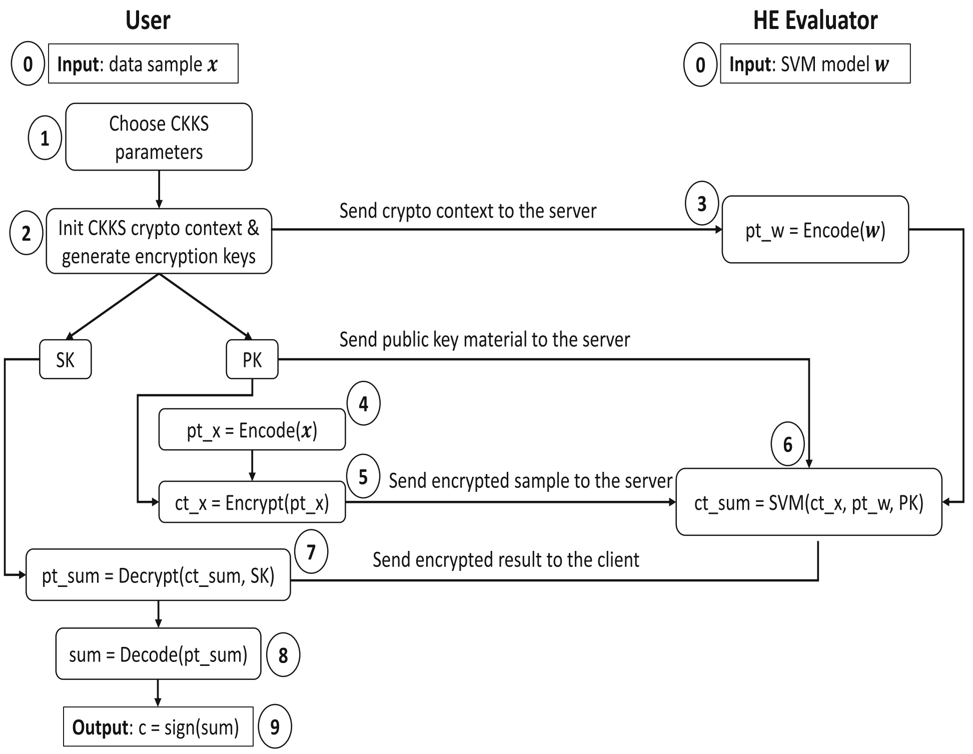
User
HE Evaluator
0 Input: data sample x
0 Input: SVM model w
1 Choose CKKS parameters
2 Init CKKS crypto context & generate encryption keys
Send crypto context to the server
3 pt_w = Encode(w)
SK
PK
Send public key material to the server
4 pt_x = Encode(x)
6
5 Send encrypted sample to the server
ct_x = Encrypt(pt_x)
ct_sum = SVM(ct_x, pt_w, PK)
7 Send encrypted result to the client
pt_sum = Decrypt(ct_sum, SK)
sum = Decode(pt_sum) 8
Output: c = sign(sum) 9

Fig. 2 Flowchart of FHSVM homomorphic prediction protocol. pt and ct refer to plaintext and ciphertext, respectively. SK and PK refer to the secret key and public key, respectively

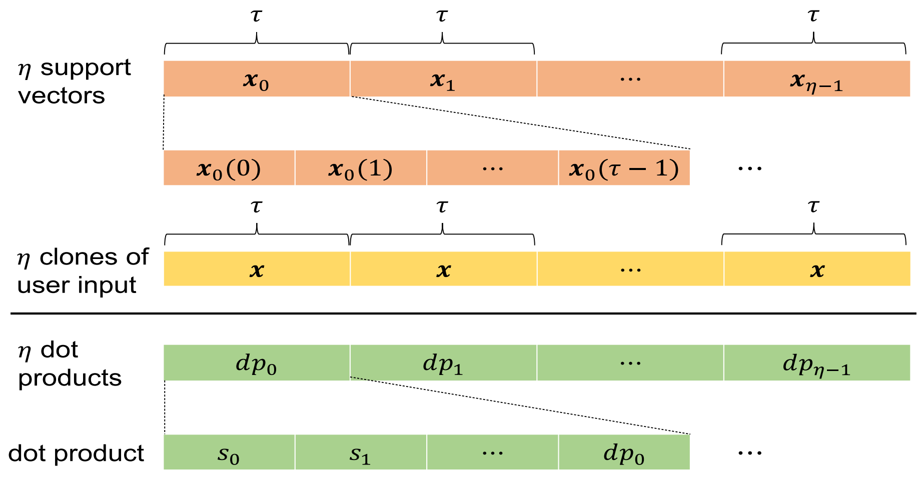

Fig. 3 Our ultra-packing strategy for the first batch of computing the decision function addends in parallel. The quantities s_i's are some inter-slot summations that will be masked out

Upon the receipt of the ciphertext encrypting the data point, the evaluator evaluates the SVM decision function homomorphically—Step 6. The resultant ciphertext is sent back to the user who can decrypt, decode and compute the sign function to obtain the classification result—Steps 7–9.

Note that all the SVM model parameters are used without encryption. The only encrypted parameter in the decision function is the end-user's input (the feature vector $\boldsymbol{x}$ of length m). A question that might arise at this stage is how to encrypt the feature vector. A naive solution is to encrypt each component of the feature vector (a scalar) in a separate ciphertext element. This solution results in the simplest FHE code design but would impose enormous computation and bandwidth requirements. For instance, the client needs to encrypt and communicate m ciphertexts to the HE evaluator. The HE evaluator would need m homomorphic multiplications and $m-1$ homomorphic additions to compute the dot product in the decision function. Due to the enormous computational and bandwidth requirements of this solution, we do not recommend it. Instead, we opt to exploit the plaintext/ciphertext packing method offered by the CKKS scheme which allows us to encrypt vectors instead of a single scalar.

3.4.1 Plaintext and ciphertext packing

The plaintext/ciphertext packing method is due to Smart and Vercauteren [32]. It allows one to encode multiple messages in one plaintext element that can be encrypted to generate only one packed ciphertext element. More concretely, an array of up to $t = n/2$ complex numbers can be encoded as one plaintext element. This enables the Single-Instruction Multiple-Data (SIMD) execution mode of homomorphic operations on packed ciphertexts for free. One may view the plaintext or ciphertext elements as a container with a fixed number of slots. In each slot, one input message (a numeric value) can be stored. Homomorphic addition/multiplication of two packed ciphertexts, say $\boldsymbol{a} = \mathsf{ENC}(v_0, \ldots, v_{t-1})$, and $\boldsymbol{b} = \mathsf{ENC}(u_0, \ldots, u_{t-1})$ results in component-wise homomorphic addition or multiplication of v_i and $u_i, \forall\, 0 \le i \le t-1$.

3.4.2 Design decisions

An important design decision that might arise here is how to compose the feature vector we would like to encrypt. Whether to encrypt the entire feature vector $\boldsymbol{x}$ as a whole in one ciphertext, or interleaving more than one feature vector, say matrix $\boldsymbol{X}^{r\times m}$, and storing them in m different ciphertexts, with ciphertext i contains $\boldsymbol{X}_{ji}, 0 \le j < r$ and $0 \le i < m$. The former is suitable for low-latency single-prediction-at-a-time applications and it imposes less communication overhead [33, 34]. The latter on the other hand is more suited for high-throughput batched-prediction applications, i.e., performing multiple predictions at a time, but it incurs higher communication and computation overhead [35, 36]. In FHSVM, we adopt the former design and offer a low-latency and low-bandwidth single prediction service.

We assume that the number of features m in $\boldsymbol{x}$ is less than the number of slots supported by CKKS. This is to ensure that the entire feature vector can be encrypted in a single ciphertext using the CKKS plaintext/ciphertext packing method. If m is larger than t, multiple ciphertexts ($\lceil \frac{m}{t} \rceil$) would be required to encrypt the whole feature vector. While the current version of FHSVM does not support this case, FHSVM can be easily adapted to tackle it.

3.4.3 FHSVM$_{\text{Vanilla}}$

Our first version of FHSVM, which we call FHSVM$_{\text{Vanilla}}$ is shown in Algorithm 2. The algorithm loops over all the support vectors in S to compute a single addend of the decision function. In each iteration, the SVM model parameters of the i-th support vector are loaded and encoded in plaintext objects (lines 3–7). The dot product between the encrypted end-user feature vector $\boldsymbol{x}$ and the i-th support vector $\boldsymbol{x}_i$ is computed via 1 HMULPLAIN, 1 RESCALE, and a call to Total_Sum in Algorithm 3 (see also Fig. 4 for illustration) to add up all the values in the ciphertext slots (lines 8–10). Next, we evaluate the kernel function K on the dot product result (line 11). Finally, we multiply the output of the kernel evaluation by the coefficient $\alpha_i y_i$ and accumulate the result in the output ciphertext.

We note that in FHSVM, we do not evaluate the sign function in the encrypted domain as it is not FHE friendly and would require expensive approximation. Instead, we send the summation value to the client who can decrypt and evaluate the sign function to learn the classification result.

The computational complexity of FHSVM$_{\text{Vanilla}}$ is of the order $\mathcal{O}(|S|)$ of 1 homomorphic computation of one addend in the decision function in Eq. (1). We will present algorithmic and architectural optimization strategies to reduce FHSVM computational overhead.

3.4.4 Algorithmic optimization: ultra-packing (FHSVM$_{\text{Pack}}$)

Our ultra-packed plaintext/ciphertext strategy stems from the fact that instead of packing a single support vector in one plaintext element, we pack multiple support vectors in one plaintext element. The only assumption we require for this strategy is to ensure that the number of features in a support vector is less than the number of slots in CKKS plaintexts. We believe that this is usually the case as the number of slots is usually in the orders of thousands whereas the number of features in SVM problems is in the orders of hundreds. Hereafter, we refer to this ultra-packed design of FHSVM as FHSVM$_{\text{Pack}}$. We provide below a concrete treatment of the packing strategy in FHSVM$_{\text{Pack}}$.

Let m denote the number of features in a support vector. As mentioned previously, the number of slots in CKKS plaintexts is $t = n/2$. Firstly, we compute $\tau = 2^{\lceil \log_2 m \rceil}$, which is, the smallest power of 2 number that is greater than m. We create a vector that contains $\eta = \frac{t}{\tau}$ support vectors. In this vector, support vector $\boldsymbol{x}_i$ is stored at index $i\tau \ \forall \ 0 \leq i < \eta$. Note that any unpopulated component in this vector should be filled with zeros. This vector of linearly

Algorithm 2: Homomorphic prediction of SVM via CKKS

```
Input: Given the plaintext SVM model parameters $S, \boldsymbol{\alpha}, \boldsymbol{y}, K, b$, and
       an encrypted feature vector $\boldsymbol{x}$
Output: ciphertext ct_sum encrypting the sum value in the decision
        function in Equation (1)
1  ct_sum = ENC(0)
2  for $i \leftarrow 0$ to S − 1 do
3      $\alpha_i = \boldsymbol{\alpha}[i]$
4      $y_i = \boldsymbol{y}[i]$
5      $\boldsymbol{x}_i = S[i]$
6      pt_$\boldsymbol{x}_i$ = ENCODE($\boldsymbol{x}_i, p$)
7      pt_coeff= ENCODE($\alpha_i y_i, p$)
       /* Compute the dot product                                   */
8      ct_dp = HMULPLAIN($\boldsymbol{x}_i, \boldsymbol{x}$) // Component-wise multiply
9      RESCALE(ct_dp, next_level) // Rescale to next level
10     ct_dp = Total_Sum(ct_dp) // Sum all slots in ct_dp
       /* Evaluate kernel function K on ct_dp                       */
11     ct_dp = K(ct_dp)
12     ct_dp = HMULPLAIN(ct_dp, pt_coeff)
13     RESCALE(ct_dp, next_level) // Rescale to next level
14     ct_sum = HADD(ct_sum, ct_dp)
15 return ct_sum
```

packed support vectors is encoded in a single plaintext element. A visual illustration can be found in Fig. 3.

We also require the client who provides the encrypted feature vector $\boldsymbol{x}$ to create a vector of linearly packed η clones of $\boldsymbol{x}$ similar to the way $\boldsymbol{x}_i$ is packed. The vector will be encoded, encrypted and sent to the server for homomorphic evaluation of SVM prediction. By doing so, we can compute η addends of *ct_sum* in each loop in Algorithm 2. To do that, two major changes to Algorithm 2 need to be done. Firstly, a procedure known as Partial_Sum (see Fig. 5) instead of Total_Sum needs to be invoked. The only difference between these two functions is the number of iterations we loop over the ciphertext (see line 1 in Algorithm 3). In Partial_Sum, we loop over $\log_2 \tau$ instead of all slots in the ciphertext. This results in the dot product result computed for support vector i stored in slot $(i+1)\cdot\tau - 1 \;\forall\; 0 \leq i < \eta$.

Secondly, the slots that contain the partial sum results need to be added only without the remaining slots in the packed resultant ciphertext. This can be done simply by using appropriate masking to keep the desired dot product results. The pseudo-code of the batched version of our FHSVM is shown in Algorithm 4.

In terms of computational complexity, since we compute a batch of η support vectors at a time, we expect that the performance of $\text{FHSVM}_{\text{Pack}}$ to be of the order $\mathcal{O}(|S|/\eta)$. Note that there is an extra masking operation included in $\text{FHSVM}_{\text{Pack}}$ which may lower the expected speedup.

Algorithm 3: Total_Sum [36]

```
Input: ciphertext $ct$ encrypting vector $\mathbf{v}$ and number of slots $t$
Output: ciphertext encrypting the total sum of elements in $v$,
        duplicated in slots
1 for $j \leftarrow 0$ to $(\log_2 t) - 1$ do
2     c = ROTATE(ct, $2^j$)
3     ct= HADD(ct, c)
4 return ct
```

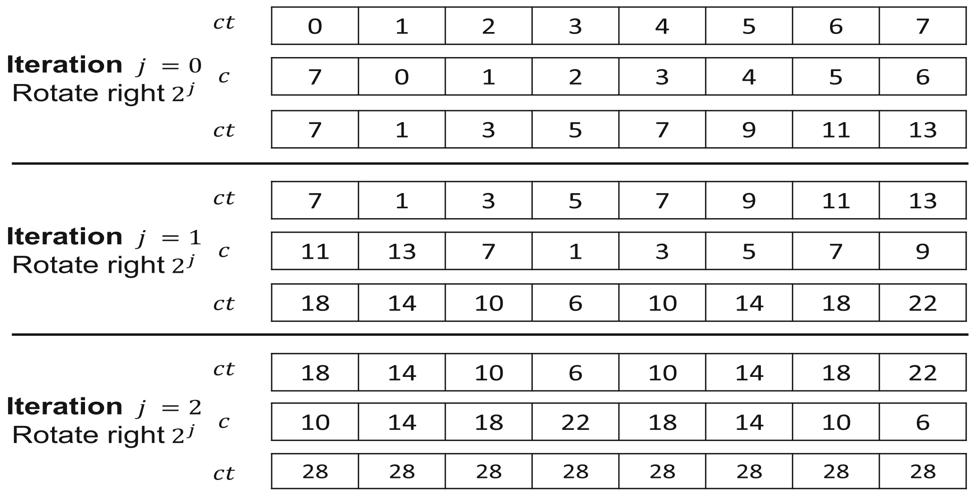

Fig. 4 Illustration of Total_Sum for a vector of eight slots

```
Algorithm 4: Batched Homomorphic prediction of SVM via CKKS
  Input: Given the plaintext SVM model parameters $S, \boldsymbol{\alpha}, \boldsymbol{y}, K, b$, and
         a batched encrypted feature vector $\bar{\boldsymbol{x}}$ constructed as described
         in Section 3.4.4.
  Output: ciphertext ct_sum encrypting the sum value in the decision
          function in (1)
1 ct_sum = ENC(0)
  /* $k$ is the batch index, process $\eta$ support vectors per batch */
2 for $k \leftarrow 0$ to $\frac{S}{\eta} - 1$ do
      /* Populate parameters in batches */
3     $\bar{\alpha}_k = \boldsymbol{\alpha}[k]$
4     $\bar{y}_k = \boldsymbol{y}[k]$
5     $\bar{\boldsymbol{x}}_k = S[k]$
6     pt_$\bar{\boldsymbol{x}}_k$ = ENCODE($\bar{\boldsymbol{x}}_k, p$)
7     pt_coeff= ENCODE($\bar{\alpha}_k \bar{y}_k, p$)
      /* Compute the dot product */
8     ct_dp = HMULPLAIN($\bar{\boldsymbol{x}}_k, \bar{\boldsymbol{x}}$) // Component-wise multiply
9     RESCALE(ct_dp, next_level) // Rescale to next level
10    Partial_Sum(ct_dp) // Sum every consecutive $\tau$ slots
      /* Evaluate kernel function K on ct_dp */
11    ct_dp = K(ct_dp)
12    ct_dp = HMULPLAIN(ct_dp, pt_coeff)
13    RESCALE(ct_dp, next_level) // Rescale to next level
      /* Extract dot products by masking with pt_mask =
         $\{0, \cdots, 1, 0, \cdots, 1, \cdots, 0, \cdots, 1\}$ where the 1's are at indices
         $i\tau - 1, \forall\ 0 \le i < \eta$ */
14    ct_dp = HMULPLAIN(pt_mask, ct_dp)
15    RESCALE(ct_dp, next_level) // Rescale to next level
      // Summation of $\eta$ SVs evaluations per batch
16    ct_dp = Total_Sum(ct_dp)
17    ct_sum = HADD(ct_sum, ct_dp)
18 return ct_sum
```

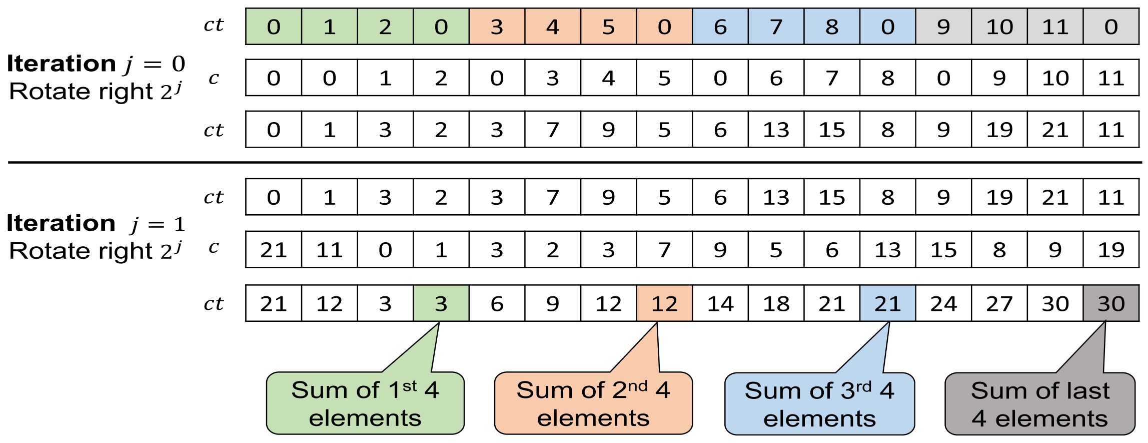

Fig. 5 Illustration of Partial_Sum for a vector of 16 slots, $m = 3$ and $\tau = 4$

4 Experimental methodology

In this section, we describe how to select the CKKS parameters to achieve the desired security level and enable efficient and correct implementation of FHSVM. We also present another architectural optimization suitable for parallel execution platforms.

4.1 CKKS parameters choice

As mentioned previously, we adopt the RNS variant of the CKKS scheme [30] to implement the two versions of FHSVM: $\mathrm{FHSVM_{Vanilla}}$ and $\mathrm{FHSVM_{Pack}}$. The CKKS scheme can be viewed as a calculator for performing arithmetic on real numbers represented as fixed-point numbers. The typical workflow in this scheme is as follows. Firstly, as part of the encoding step, the input operands are multiplied by a predefined scale factor p and rounded to the nearest integer. Thus, the first parameter that needs to be fine-tuned is the scale factor p. This can be done by simulating the target computation using a fixed-point arithmetic simulator to find the best value of p that provides sufficient precision. Then, the scaled data can be encrypted to be used as operands of the homomorphic computation.

In fixed-point representation, adding/subtracting two numbers can be carried out by adding/subtracting the underlying integers if they have the same scale factor. The result will also have the same scale factor. Extra caution is required that no overflow occurs during the addition/subtraction of the underlying integers. If the numbers have different scale factors, one of them needs to be scaled down to match the scale factor of the other number. This scale down operation might result in precision loss due to rounding. To multiply two numbers, we only need to multiply the two underlying integers and compute the new scaling factor of the result as the product of operands scaling factors. To maintain the number of bits for the product, we can scale down the integer part by the number of bits required. CKKS emulates fixed-point arithmetic on encrypted data such as addition, subtraction, multiplication and truncation.

To ensure high precision homomorphic computation, we scale down the product ciphertext after each multiplication. This results in consuming one level in RNS CKKS. We use 2^{50} as the input scale factor and set the ciphertext coefficient q_L as the product of 60-bit and 50-bit primes. Denote to a 60-bit and 50-bit prime numbers by $\rho^{(60)}$ and $\rho^{(50)}$, respectively. We set $q_L = \prod\{\rho_0^{(60)}, \rho_0^{(50)}, \ldots, \rho_{L-2}^{(50)}, \rho_1^{(60)}\}$. Note that in RNS CKKS, scale down from level l to level $l-1$ is done by integer division of q_l/ρ_l. Hence, there is even another source of precision loss as we are not scaling down by 2^{50}.

Next, we estimate the number of levels required by FHSVM. In the $\mathrm{FHSVM_{Vanilla}}$, we need 2 plus the multiplicative depth of evaluating the kernel function K. Whereas one extra level is required in $\mathrm{FHSVM_{Pack}}$ due to the masking step in line 14 in Algorithm 4. Once the size of the ciphertext coefficient and the desired security level are fixed, one can refer to the LWE hardness estimator [38] to find the polynomial ring dimension n. We target a 128-bit security level. These parameters are application-dependent; therefore, specific parameters will be provided later when we present a concrete use case application of FHSVM.

As for the CKKS encryption keys generation, we follow the recommendations of the draft version of the FHE standard [39]. The secret key is sampled from a uniform ternary distribution (i.e., polynomials with coefficients in $\mathcal{X}_{key} = \{-1, 0, 1\}$ sampled uniformly). The Gaussian distribution of the noise $\mathcal{X}_{err}$ has a mean of zero and standard deviation of 3.19.

4.2 Architectural optimization: multi-threading

An immediate optimization of FHSVM is to employ a multi-threaded implementation. We can exploit the parallelism in the main loop in Algorithms 2 and 4 using a parallel implementation. We used the OpenMP C++ library to develop a parallel version of FHSVM that we call $\mathrm{FHSVM_{OMP}}$. In the following paragraphs, we describe this optimization on Algorithm 2. However, the same approach can be used to parallelize Algorithm 4.

We basically parallelize the main loop that runs over all the support vectors. To do that, we create a vector of ciphertexts of length $|S|$ to store all the addends in ct_sum. The homomorphic computation of each addend in ct_sum is computed independently in parallel by an individual CPU thread. The loop in Algorithm 2 is followed by another loop that is used to sum all the addends in ct_sum.

The cost of this optimization is the need to create a vector of ciphertext addends in memory. As the size of CKKS ciphertexts is quite large (approximately $2 \cdot n \cdot \log_2 q$ bits), this can be quite expensive especially when the number of support vectors is large. If the system memory is not sufficient to support this execution, a batched execution mode would be preferable where a batch (a subset of all addends) of addends is computed at a time.

During the evaluation, we vary the number of CPU threads and observe the latency of $\mathrm{FHSVM_{OMP}}$. Our experiments suggest that a number of OMP threads that is equal to the number of CPU cores on the system provides the best performance. This will be empirically proven later in the subsequent section.

In terms of computational complexity, since we compute in parallel as many batches of η support vectors as the number of threads (T), we expect that the performance of $\text{FHSVM}_{\text{OMP\_Pack}}$ to be of the order $\mathcal{O}(|S|/(\eta T))$.

5 Experimental results

In this section, we evaluate FHSVM algorithm in a real-world use case scenario related to the detection of Anti-Money Laundering (AML) in Bitcoin transactions. We describe the AML classification task, dataset employed, FHSVM implementation parameters, execution platform, and experimental configurations. We also provide an extensive list of experiments to analyze the performance of FHSVM with and without the optimization strategies presented earlier.

5.1 AML classification use case

We employ FHSVM for private detection of AML in bitcoin transactions. Given a labeled dataset, we train an SVM model that is able to classify bitcoin transactions into licit and illicit transactions.

5.1.1 Dataset

For our evaluation, we employed the Elliptical Dataset of real bitcoin transactions [20]. The dataset is presented as a time series graph with nodes representing transactions and edges of the directed payment flow between nodes. It consists of over 200K transactions with each having 166 features. All the features are standardized, i.e., with zero-mean and a unit standard deviation/variance. The first 94 features are based on local information e.g. time steps, transaction fees. The remaining are aggregated features obtained by one hop backward/forward transactions. The training dataset consists of around 23% of the transactions labeled as illicit and licit, while the remaining unlabelled ones become a part of the testing dataset. Statistics of the dataset are shown in Table 2.

Table 2 Statistics of the elliptic dataset

Statistic	Value
Number of nodes	203,769
Number of edges	234,355
Number of features	166
Number of licit transactions	42,019
Number of Illicit transactions	4,545
Number of unknown transactions	187,791

Table 3 Illicit classification evaluation metrics of our SVM model with polynomial kernel. LR refers to logistic regression, RF refers to random forest, MLP refers to multi-layer perceptron and GCN refers to graph convolutional networks

Method	Precision	Recall	F1	MicroAvg-F1
LR [20]	0.404	0.593	0.481	0.931
RF [20]	0.956	0.670	0.788	0.977
MLP [20]	0.694	0.617	0.653	0.962
GCN [20]	0.812	0.623	0.705	0.966
Ours	0.840	0.376	0.520	0.964

5.1.2 Training an SVM model in the clear

Following the procedure shown in Sect. 3.3, we train an SVM model in the plaintext domain using a polynomial kernel of the form $K = \gamma X^2$. The generated SVM model has 7780 support vectors. We evaluate the performance of the SVM model on the testing dataset (25% of the entire dataset) in the plaintext domain. The classification accuracy of the generated SVM model is found to be 96.43%. We believe that this classification accuracy is quite high as state-of-the-art methods show slightly higher accuracy figures (96.60%) using Graph Neural Networks [20]. These classification accuracy figures are useful to assess the classification accuracy achieved by FHSVM on the encrypted testing dataset and have been shown in Table 3. Precision is intuitively the ability of the classifier not to label as positive a sample that is negative. The recall metric is the ability of the classifier to find all the positive samples. F1-score is a way of combining the precision and recall into a single number by finding their harmonic mean. The microAvg-F1 score can be considered the classifier's

Table 4 CKKS and FHSVM parameters for the AML task

Framework	CKKS parameters				FHSVM parameters				
	n	$\log_2 q$	p	L	$\lvert S\rvert$	m	r	d	γ
$\text{FHSVM}_{\text{Vanilla}}$	2^{14}	320	50	4	7780	165	0	2	$1/m$
$\text{FHSVM}_{\text{Pack}}$	2^{14}	370	50	5	7780	165	0	2	$1/m$

The SVM polynomial kernel is parameterized by d, r and γ. $|S|$ denote the number of support vectors in the SVM model. The security level of CKKS under these parameters is $\lambda > 128$ bit against the Unique Shortest Vector Problem (uSVP), Dual and Decoding attacks [38, 39] overall accuracy: the proportion of correctly classified samples out of all the samples.

5.2 Implementation of FHSVM

In this section, we describe in detail our implementation of FHSVM for the AML use case.

5.2.1 Development environment

We used Microsoft SEAL version 3.6.2 to implement FHSVM in C++. Microsoft SEAL includes an efficient C++ implementation of an RNS variant of the CKKS scheme. FHSVM has been developed and compiled on a machine equipped with the ArchLinux operating system vecsion 5.9.14-arch1-1 and the C++ compiler GCC version 10.2.0.

5.2.2 FHSVM and CKKS parameters

As mentioned in Sect. 5.1.2, the kernel function is of degree 2 of the form $K = \gamma X^2$. Therefore; the multiplicative depth for the homomorphic evaluation of K (line 11 in Algorithms 2 and 4) is 2. This means the total multiplicative depth of $\text{FHSVM}_{\text{Vanilla}}$ and $\text{FHSVM}_{\text{Pack}}$ is 4 and 5, respectively. The concrete FHSVM and CKKS parameters to support this computation with $\lambda = 128$ bit security can be found in Table 4.

5.3 Performance evaluation

To evaluate the performance of FHSVM, we use three main experiments. In Experiment I, we analyze the performance of $\text{FHSVM}_{\text{Vanilla}}$ without optimizations reporting the average prediction latency. In Experiment II, we evaluate the performance of $\text{FHSVM}_{\text{OMP}}$ to analyze the effect of multi-threading execution. Lastly, we study the performance of FHSVM after applying both the multi-threading optimization and our packing strategy in Experiment III. This implementation is termed as $\text{FHSVM}_{\text{OMP\_Pack}}$. Note that in our timing analysis, we only report the homomorphic computation time. Any system initialization operations or encoding of the constant SVM model parameters are done once at the system setup. The latency of FHSVM prediction is calculated using the std::chrono C++ library [40].

5.3.1 Platform configuration

We ran our experiments on a server that hosts 64-bit Intel Xeon Platinum 8170 CPUs rated @ 2.10 GHz with 2 CPU sockets. There are 26 physical CPU cores on each socket. Each physical CPU core is able to run two logical threads. Therefore, the total number of physical cores on the machine is 52 cores, whereas the total number of logical cores is 104 cores. The server is equipped with a 188 GB RAM with a speed of 2666 MT/s.

5.3.2 Experiment I: $\text{FHSVM}_{\text{Vanilla}}$ performance

We implemented $\text{FHSVM}_{\text{Vanilla}}$ as described in Algorithm 2 as a baseline unoptimized implementation. This implementation is single-threaded and does not include any advanced optimizations except for ensuring that the SVM model constants are all encoded in CKKS plaintext objects before the homomorphic computation is carried out. For instance, the plaintext objects in lines 3–7 in Algorithm 2 are pre-computed and stored in memory.

Table 5 shows the number of tested examples and average latency per example using $\text{FHSVM}_{\text{Vanilla}}$. The recorded average prediction latency per example is found to be 687.75 s (or 11.46 min). This experiment has not been tested for all examples in the testing dataset due to the high prediction latency. We ran this experiment for 100 randomly selected examples from the testing dataset. We do not report the accuracy results in this experiment due to the limited number of examples tested. Later on, we will provide prediction accuracy results over all the examples in the testing dataset. The main purpose of $\text{FHSVM}_{\text{Vanilla}}$ is to serve as a baseline for studying the effect of our optimization strategies on the prediction latency in FHSVM.

5.3.3 Experiment II: $\text{FHSVM}_{\text{OMP}}$ performance

We parallelized Algorithm 2 using OMP threads to construct $\text{FHSVM}_{\text{OMP}}$. Table 6 shows the number of CPU threads used, average latency per example, and the achieved speed up by $\text{FHSVM}_{\text{OMP}}$ over 500 examples randomly selected from the testing dataset (except for the

Table 5 Performance analysis of $\text{FHSVM}_{\text{Vanilla}}$ in terms of average prediction latency per example (in s)

Framework	# Examples	Avg latency	StdDev	Min	Max
$\text{FHSVM}_{\text{Vanilla}}$	100	687.75	18.72	659.34	705.35

Avg and StdDev stand for the average and standard deviation, respectively

Table 6 Performance analysis of FHSVM$_{OMP}$ in terms of prediction latency per example (in s) while varying the number of CPU threads

# Threads	# Examples	Avg latency	Speed up	StdDev	Min	Max
1	100	687.75	1.00×	18.72	659.34	705.35
2	100	350.10	1.96×	25.60	313.34	385.33
4	100	187.01	3.68×	18.68	154.18	210.85
8	500	96.53	7.12×	12.89	76.76	115.46
16	500	51.21	13.43×	13.72	28.45	76.14
26	500	34.88	19.72×	12.28	20.33	55.13
27	500	34.15	20.14×	7.33	24.79	44.27
32	500	30.76	22.36×	6.42	21.91	39.23
52	500	**25.21**	**27.28** ×	3.75	18.83	30.19
64	500	27.93	24.62×	5.05	21.95	36.61
104	500	30.62	22.46×	8.67	19.88	42.28
128	500	32.25	21.33×	7.76	17.55	45.55

Table 7 Performance analysis of FHSVM$_{OMP\_Pack}$ in terms of average prediction latency (over all testing examples) and overall speed up factors (with respect to FHSVM$_{Vanilla}$) while varying the number of CPU threads

# Threads	Avg Latency (sec)	Overall Speed up	StdDev	Min	Max
1	29.89	23.01×	6.33	20.30	39.23
2	15.30	44.95×	3.05	11.45	20.86
4	8.14	84.49×	2.47	4.60	12.22
8	4.33	158.83×	1.23	2.47	6.17
16	2.33	295.17×	0.77	1.19	3.82
26	1.62	295.17×	0.39	0.77	2.25
27	1.64	424.01×	0.95	0.65	3.89
32	1.51	419.36×	0.45	0.35	1.85
52	**1.25**	**550.20** ×	0.28	0.79	1.71
64	1.47	467.86×	0.71	0.43	2.87
104	1.38	498.37×	0.19	1.15	1.74
128	1.41	487.77×	0.53	1.01	2.69

FHSVM$_{OMP\_Pack}$ achieves 96.43% prediction accuracy

1, 2, and 4 threads configurations which we ran only for 100 examples). The speedup is calculated as the ratio between single-threaded latency (i.e., average latency per example in FHSVM$_{Vanilla}$ to the multi-threaded latency in FHSVM$_{OMP}$). As we can see, FHSVM$_{OMP}$ improves the latency by almost one order of magnitude when the number of threads is 8 or above. The speedup factors show decent scalability results as the number of threads is increased until it plateaus around 52 threads. The highest performance speed up (27.28×) is achieved at 52 threads, which is the number of physical CPU cores on the testing machine.

5.3.4 Experiment III: FHSVM$_{OMP\_Pack}$ performance

In the last experiment, we evaluate the performance of FHSVM after including both optimizations: the multi-threaded execution and the packing strategy. We implemented FHSVM$_{Pack}$ as described in Algorithm 4. We also parallelized the main loop using OMP threads to build FHSVM$_{OMP\_Pack}$. Table 7 shows the performance results of FHSVM$_{OMP\_Pack}$ while varying the number of CPU threads. The overall speed up factors shown are computed with respect to the FHSVM$_{Vanilla}$ average latency. As expected, the highest overall speed up factor (550.20×) is achieved at 52 CPU threads. The best average latency reported is 1.25 s demonstrates that FHSVM$_{OMP\_Pack}$ is suitable for real-time predictions. In addition, we notice that there is a zero loss in the prediction accuracy when the system is tested over all the examples in the testing dataset.

To verify the speedup we got from employing our algorithmic and architectural optimizations in FHSVM$_{OMP\_Pack}$, we calculate an estimated value of the expected speedup on the AML task on our testbed machine.

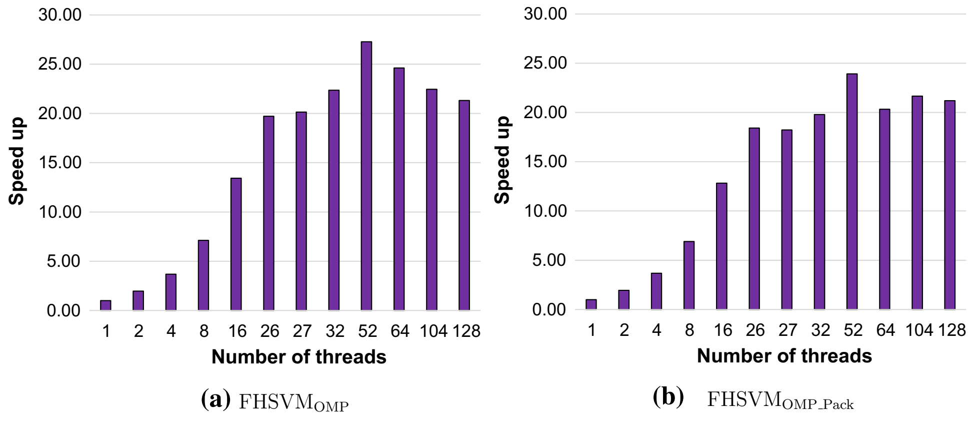

Fig. 6 Speedup factors achieved in FHSVM$_{\mathrm{OMP}}$ and FHSVM$_{\mathrm{OMP\_Pack}}$ as we vary the number of CPU threads. The i-th speed up factor is computed as the ratio between the average latency achieved in FHSVM$_{\mathrm{OMP}}$ and FHSVM$_{\mathrm{OMP\_Pack}}$ with T threads to the average latency with 1 thread

As described in Sect. 3.4.4, the speed up factor achieved due to packing is found to be η. In this problem, $\eta = \frac{n = 2^{14}}{2\tau = 2^{\log_2 \lceil m=165 \rceil)}} = 32\times$ achieved from packing alone. Referring to Table 6, we estimate that the speed up we get from multi-threading is 23.91×. Thus, the overall theoretical speed up expected from both optimizations is close to $32 \cdot 23.91 = 764.8\times$ which is not much higher than the actual maximum speed up (550.20×) reported in Table 7.

5.3.5 Scalability

We are also interested in the scalability (or parallelism efficiency) of our implementations of the optimized FHSVM. We study the improvement in average prediction latency in both optimized implementations: FHSVM$_{\mathrm{OMP}}$ and FHSVM$_{\mathrm{OMP\_Pack}}$ as we vary the number of threads. We aim to show that the performance of FHSVM can be dramatically increased as we increase the number of CPU threads. Figure 6 can be useful to study the scalability of our implementations. As we can see, both implementations show consistent scalability trajectories. The speedup factors are almost doubled as we double the number of CPU threads from 1 to 16 threads. This shows a perfect scalability situation which is expected when the computational problem is embarrassingly parallel and the workload is balanced among the worker threads; both apply to the FHSVM case. Starting from 32 threads, the rate at which the speedup factors grow becomes slower. We found that this is a common problem in high-end servers where the physical CPU cores are hosted across multiple sockets [41]. These systems show linear scalability as long as all CPU threads run on the same socket. However, the performance can deteriorate if a single thread runs on a different socket. Recall that our testbed server hosts 26 CPUs on each socket. The reason behind this behavior can be attributed to any of the following: data bus contention, memory contention, higher cache misses, cache coherency protocols overhead, and synchronization overhead [42].

In both FHSVM frameworks, the speedup factors keep increasing until they plateau at 52 threads, which is the total number of physical CPU cores on the server. After that, the scalability starts to decrease slowly as the CPU threads become more overloaded. We believe that the performance of FHSVM can be improved further on machines that are equipped with a higher number of CPUs on one socket. Or on execution platforms that do not suffer from the aforementioned scalability issue.

5.3.6 Message size

We are also interested in the encryption keys and ciphertext message sizes used in FHSVM. Table 8 shows the sizes in MiB of all the keys and messages used in FHSVM$_{\mathrm{Vanilla}}$ and FHSVM$_{\mathrm{Pack}}$. The public, relinearization and Galois

Table 8 CKKS encryption keys and ciphertext messages sizes in MiB (2^{20} bytes) used in FHSVM

Framework	Public key	Secret key	Relin key	Galois keys	Input ciphertxt	Output ciphertxt
FHSVM $_{\mathrm{Vanilla}}$	1.399	0.699	6.995	182.184	1.149	0.250
FHSVM $_{\mathrm{Pack}}$	1.624	0.812	9.742	253.671	1.374	0.250

The Relinearization key refer to the keys used CKKS RELINEARIZE. The Galois keys refer to the keys used in CKKS ROTATE. The input ciphertext refers to the ciphertext message communicated from the end user to the HE evaluator, whereas the output ciphertext refers to the encrypted SVM prediction result communicated from the HE evaluator back to the end user

Table 9 Comparison of prediction latency (s) between prior privacy-preserving SVM and our FHSVM. RM and PA stand for random masking and polynomial aggregation

Framework	Year	$\mid S \mid$	m	Method	Latency	Dataset
[8]	2014	< 213	2601	Pailler + MPC	149.82	JAFFE
[9]	2017	1191	29	FHE	8.97	CIFAR-10
[17]	2017	100	10	RM + PA	2.00	PID
[13]	2020	< 100	60	FHE	–	Sonar
[14]	2021	< 1000	3	FHE	220.38	Chainlink
Ours	2021	7780	165	FHE	1.25	Elliptic

PID stands for Pima Indians Diabetes

keys are transferred once and can be stored at the HE evaluator. This should be done per each end-user. The input ciphertext is an encrypted instance communicated from the end-user to the HE evaluator. The output ciphertext is the homomorphic SVM prediction result communicated by the HE evaluator back to the end-user. It can be noticed that $\text{FHSVM}_{\text{Pack}}$ requires larger data transfers due to the larger ciphertext coefficient size q. The output ciphertext is of the same size as we drop all the smaller primes from q when we reach the last level configured in CKKS.

5.4 Comparison with prior works

Lastly, we compare our best FHSVM results with state-of-the-art solutions. Table 9 contrasts the reported prediction time results of prior works on privacy-preserving SVM prediction on different tasks and datasets. FHSVM clearly outperforms existing solutions in the prediction latency. The results show that FHSVM is also efficient at evaluating SVMs with a large number of support vectors and a non-trivial number of features on large real-world datasets. We note that this is not a fair comparison due to the employment of different datasets, execution platforms, and FHE libraries.

6 Discussion

We have seen that FHSVM provides fast and accurate SVM prediction on encrypted data using CKKS. In this section, we discuss some of the issues related to the real-world deployment of FHSVM such as security, potential improvements, and limitations.

6.1 Security

It should be noted here that FHSVM assumes that the input samples have higher importance compared to the SVM model owned by the HE evaluator. In fact, a malicious end-user can recover the SVM model parameters if he or she is given arbitrary access to the homomorphic SVM prediction service. The reason is that in FHSVM and most existing privacy-preserving machine learning applications that employ FHE and multi-party computing, the HE evaluator returns the class scores vector instead of the class which is opted for to improve efficiency. It has been shown that this approach can lead to model inversion and extraction attacks [43]. Despite the large computational overhead of such attacks as the client needs to communicate a large number of queries with the server, nevertheless, these attacks can be tackled in several ways such as 1) evaluating the sign function in SVM prediction on the server using polynomial approximation but this can be costly, 2) adding extra noise to the computed prediction result before returning it to the end-user and 3) limiting the number of queries the end-user can instantiate. Note also that machine learning models undergo constant and frequent updates as new data and training methods are used, which makes these attacks less realistic.

6.2 Hardware acceleration

Hardware acceleration of FHE has shown dramatic improvement over single-threaded and multi-threaded CPU implementations [44]. Recent work showed that the performance of CKKS on GPUs can be improved by almost two orders of magnitude compared with Microsoft SEAL [45]. By analogical reasoning, for the encryption parameters used in FHSVM here, the GPU implementation may provide about $30\times$ improvement in the performance pushing FHSVM latency down to 42 ms. This can make FHSVM even more attractive for real-time applications.

6.3 Limitation of FHSVM

Besides the model inversion and extraction attacks (see Sect. 6.1) that FHSVM is vulnerable to, the main limitation of FHSVM is the support for polynomial kernels only. Other kernel functions such as RBFs are commonly used in SVMs. Support for these kernels is possible via approximation methods or LUT search at a reasonable cost.

Further investigation is required to estimate the feasibility, limitations, and computational cost of these approaches. Another limitation of FHSVM is the lack of training SVMs on encrypted datasets.

7 Conclusion

We presented FHSVM: a method for fast and accurate privacy-preserving implementation of SVM prediction on encrypted data using FHE. In particular, the CKKS scheme has been used to realize FHSVM while ensuring correct and precise functionality (full utility of SVM) and more than a 128-bit security level (satisfactory protection of data privacy). At the core of FHSVM is the employment of algorithmic and architectural optimization strategies to overcome the profound FHE computational overhead. We showed how we can fully utilize the plaintext space in CKKS through packing algorithms that result in reducing the total number of homomorphic operations. The architectural optimization was driven by parallel programming methods and show high scalability on multi-core execution platforms. We evaluated FHSVM on a large real-world dataset related to AML in Bitcoin transactions. Our analysis shows that FHSVM provides very fast prediction latency (about 1.25 s) per example. Moreover, FHSVM does not suffer from prediction accuracy loss when compared with the unencrypted SVM prediction.

We provided a thorough performance analysis to show the effect of each optimization included in FHSVM on the system overall performance. Our experiments showed that the packing method can provide 23.01$\times$ improvement in prediction latency compared to the non-packed (vanilla) version of FHSVM. We also showed that FHSVM scales well when the number of computing cores is increased. Including both optimizations generated more than 550$\times$ overall speed up in the prediction latency. We believe that FHSVM is suitable for real-time prediction as a service cloud application for secure SVM algorithms. We plan to evaluate FHSVM on other datasets using other alternative kernel functions. We will also investigate methods to expand the functionality of FHSVM and include support for SVM training on encrypted datasets.

Acknowledgements Ahmad Al Badawi was supported by Rabdan Academy. Ling Chen and Saru Vig were supported by A*STAR under its RIE2020 Advanced Manufacturing and Engineering (AME) Programmatic Programme (Award A19E3b0099).

Declarations

Declarations The authors have no conflicts of interest to declare that are relevant to the content of this article.

References

1. Cui D, Curry D (2005) Prediction in marketing using the support vector machine. Market Sci 24(4):595–615
2. Yu H, Chen R, Zhang G (2014) A SVM stock selection model within PCA. Proc Comput Sci 31:406–412
3. Venkatesan C, Karthigaikumar P, Paul A, Satheeskumaran S, Kumar R (2018) ECG signal preprocessing and SVM classifier-based abnormality detection in remote healthcare applications. IEEE Access 6:9767–9773
4. Ribeiro M, Grolinger K, Capretz MAM (2015) MLaaS: Machine learning as a service. In: 2015 IEEE 14th international conference on machine learning and applications (ICMLA), pp 896–901. https://doi.org/10.1109/ICMLA.2015.152
5. Gentry C (2009) Fully homomorphic encryption using ideal lattices. In: STOC '09. New York, NY, USA: Association for Computing Machinery, pp 169–178. https://doi.org/10.1145/1536414.1536440
6. Maekawa T, Kawamura A, Kinoshita Y, Kiya H (2018) Privacy-preserving svm computing in the encrypted domain. In: Asia-Pacific signal and information processing association annual summit and conference (APSIPA ASC). IEEE, pp 897–902
7. Chuman T, Kurihara K, Kiya H (2017) Security evaluation for block scrambling-based ETC systems against extended jigsaw puzzle solver attacks. In: 2017 IEEE international conference on multimedia and expo (ICME), pp 229–234. https://doi.org/10.1109/ICME.2017.8019487
8. Rahulamathavan Y, Phan RCW, Veluru S, Cumanan K, Rajarajan M (2014) Privacy-preserving multi-class support vector machine for outsourcing the data classification in cloud. IEEE Trans Dependable Secure Comput 11(5):467–479. https://doi.org/10.1109/TDSC.2013.51
9. Barnett A, Santokhi J, Simpson M, Smart NP, Stainton-Bygrave C, Vivek S et al (2017) Image classification using non-linear support vector machines on encrypted data. IACR Cryptol ePrint Arch 2017:857
10. Krizhevsky A, Nair V, Hinton G (2010) Cifar-10 (canadian institute for advanced research), vol 5, no. 4. http://www.cs.toronto.edu/kriz/cifar.html
11. Brakerski Z, Gentry C, Vaikuntanathan V (2014) (Leveled) fully homomorphic encryption without bootstrapping. ACM Trans Comput Theory 6(3):1–36
12. Cheon JH, Kim A, Kim M, Song Y (2017) Homomorphic encryption for arithmetic of approximate numbers. In: Takagi T, Peyrin T (eds) Advances in cryptology – ASIACRYPT 2017. Lecture notes in computer science, vol 10624. Springer, Cham. https://doi.org/10.1007/978-3-319-70694-8_15
13. Park S, Byun J, Lee J, Cheon JH, Lee J (2020) HE-friendly algorithm for privacy-preserving SVM training. IEEE Access 8:57414–57425
14. Byun J, Lee J, Park S (2021) Privacy-preserving evaluation for support vector clustering. Electron Lett 57(2):61–64
15. Liu X, Lu R, Ma J, Chen L, Qin B (2016) Privacy-preserving patient-centric clinical decision support system on Naïve Bayesian classification. IEEE J Biomed Health Inform 20(2):655–668. https://doi.org/10.1109/JBHI.2015.2407157
16. Gong Y, Fang Y, Guo Y (2016) Private data analytics on biomedical sensing data via distributed computation. IEEE/ACM Trans Comput Biol Bioinform 13(3):431–444. https://doi.org/10.1109/TCBB.2016.2515610
17. Zhu H, Liu X, Lu R, Li H (2017) Efficient and privacy-preserving online medical prediagnosis framework using nonlinear SVM. IEEE J Biomed Health Inform 21(3):838–850. https://doi.org/10.1109/JBHI.2016.2548248

18. Teo SG, Han S, Lee VC (2013) Privacy preserving support vector machine using non-linear kernels on hadoop mahout. In: 2013 IEEE 16th international conference on computational science and engineering, pp 941–948. https://doi.org/10.1109/CSE.2013.200
19. Rahulamathavan Y, Phan RCW, Veluru S, Cumanan K, Rajarajan M (2013) Privacy-preserving multi-class support vector machine for outsourcing the data classification in cloud. IEEE Trans Dependable Secure Comput 11(5):467–479
20. Weber M, Domeniconi G, Chen J, Weidele DKI, Bellei C, Robinson T et al (2019) Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. arXiv preprint. arXiv:1908.02591
21. van Dijk M, Gentry C, Halevi S, Vaikuntanathan V (2010) Fully homomorphic encryption over the integers. In: Gilbert H (ed) Advances in cryptology – EUROCRYPT 2010. Lecture notes in computer science, vol 6110. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-13190-5_2
22. Brakerski Z, Vaikuntanathan V (2011) Efficient fully homomorphic encryption from (standard) LWE. SIAM J Comput 43(2):831–871. https://doi.org/10.1137/120868669
23. Brakerski Z, Vaikuntanathan V (2011) Fully homomorphic encryption from ring-LWE and security for key dependent messages. In: Rogaway P (ed) Advances in cryptology – CRYPTO 2011. Lecture notes in computer science, vol 6841. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-22792-9_29
24. Brakerski Z (2012) Fully homomorphic encryption without modulus switching from classical GapSVP. In: Cryptology ePrint Archive, Report 078. http://eprint.iacr.org/2012/078. Accessed 1 Mar 2021
25. López-Alt A, Tromer E, Vaikuntanathan V (2012) On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: Proceedings of the forty-fourth annual ACM symposium on theory of computing (STOC '12). Association for computing Machinery, New York, USA, pp. 1219–1234. https://doi.org/10.1145/2213977.2214086
26. Fan J, Vercauteren F (2012) Somewhat practical fully homomorphic encryption. In: Cryptology ePrint archive, Report /144. http://eprint.iacr.org/2012/144. Accessed 1 Mar 2021
27. Gentry C, Sahai A, Waters B (2013) Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti R, Garay JA (eds) Advances in cryptology – CRYPTO 2013. Lecture notes in computer science, vol 8042. Springer, Berlin, Heidelberg, pp 75–92. https://doi.org/10.1007/978-3-642-40041-4_5
28. Chillotti I, Gama N, Georgieva M, Izabachène M (2016) Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. Cryptology ePrint Archive, Report 2016/870. http://eprint.iacr.org/2016/870. Accessed 23 Mar 2021
29. Cheon JH, Han K, Kim A, Kim M, Song Y (2019) A full RNS variant of approximate homomorphic encryption. In: Cid C, Jacobson M Jr (eds) Selected areas in cryptography – SAC 2018. Lecture notes in computer Science, vol 11349. Springer, Cham, pp 347–368. https://doi.org/10.1007/978-3-030-10970-7_16
30. Potdar K, Pardawala TS, Pai CD (2017) A comparative study of categorical variable encoding techniques for neural network classifiers. Int J Comput Appl 175(4):7–9
31. Smart NP, Vercauteren F (2014) Fully homomorphic SIMD operations. Des Codes Cryptogr 71(1):57–81
32. Brutzkus A, Gilad-Bachrach R, Elisha O (2019) Low latency privacy preserving inference. In: Proceedings of the 36th international conference on machine learning. Proceedings of machine learning research, vol 97, pp 812–821. https://proceedings.mlr.press/v97/brutzkus19a.html
33. Jin C, Badawi AA, Unnikrishnan B, Lin J, Mun CF, Brown JM et al (2019) CareNets: efficient homomorphic CNN for high resolution images. In: NeurIPS workshop on privacy in machine learning. NeurIPS; 2019, pp 1–6. https://oar.a-star.edu.sg/communities-collections/articles/14613. Accessed 25 Mar 2021
34. Gilad-Bachrach R, Dowlin N, Laine K, Lauter K, Naehrig M, Wernsing J (2016) CryptoNets: applying neural networks to encrypted data with high throughput and accuracy. In: Proceedings of the 33rd international conference on machine learning. Proceedings of machine learning research, vol 48, pp 201–210. https://proceedings.mlr.press/v48/gilad-bachrach16.html
35. AlBadawi A et al (2020) Towards the alexNet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with GPUs. IEEE Trans Emerg Top Comput 9(3):1330–1343. https://doi.org/10.1109/TETC.2020.3014636
36. Halevi S, Shoup V (2014) Algorithms in helib. In: Annual cryptology conference. Springer, pp 554–571
37. Albrecht MR, Player R, Scott S (2015) On the concrete hardness of learning with errors. J Math Cryptol 9(3):169–203
38. Albrecht MR, Chase M, Chen H, Ding J, Goldwasser S, Gorbunov S et al (2019) Homomorphic encryption standard. IACR Cryptol ePrint Arch 2019:939
39. Chrono.: C++ Chrono time library. http://en.cppreference.com/w/cpp/chrono. Accessed 2021 Online
40. Brown T, Kogan A, Lev Y, Luchangco V (2016) Investigating the performance of hardware transactions on a multi-socket machine. In: Proceedings of the 28th ACM symposium on parallelism in algorithms and architectures (SPAA '16). Association for Computing Machinery, New York, NY, USA, pp 121–132. https://doi.org/10.1145/2935764.2935796
41. Bardhan S, Menascé DA (2014) Predicting the effect of memory contention in multi-core computers using analytic performance models. IEEE Trans Comput 64(8):2279–2292
42. Boemer F, Cammarota R, Demmler D, Schneider T, Yalame H (2020) MP2ML: a mixed-protocol machine learning framework for private inference. In Proceedings of the 15th international conference on availability, reliability and security (ARES '20). Association for Computing Machinery, New York, NY. https://doi.org/10.1145/3407023.3407045
43. Al Badawi A, Polyakov Y, Aung KMM, Veeravalli B, Rohloff K (2019) Implementation and performance evaluation of RNS variants of the BFV homomorphic encryption scheme. IEEE Trans Emerg Top Comput 9(2):941–956. https://doi.org/10.1109/TETC.2019.2902799
44. Al Badawi A, Hoang L, Mun CF, Laine K, Aung KMM (2020) Privft: private and fast text classification with homomorphic encryption. IEEE Access 8:226544–226556

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.