9-2020

Privacy-preserving outsourced calculation toolkit in the cloud

Ximeng LIU
Singapore Management University, xmliu@smu.edu.sg

Robert H. DENG
Singapore Management University, robertdeng@smu.edu.sg

Kim-Kwang Raymond CHOO

Yang YANG

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

Published in IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 05, pp. 898-911, 2020.
doi: 10.1109/TDSC.2018.2816656

Privacy-Preserving Outsourced Calculation Toolkit in the Cloud

Ximeng Liu, *Member, IEEE*, Robert H. Deng, *Fellow, IEEE*,
Kim-Kwang Raymond Choo, *Senior Member, IEEE*,
Yang Yang, *Member, IEEE*, and HweeHwa Pang

Abstract—In this paper, we propose a privacy-preserving outsourced calculation toolkit, Pockit, designed to allow data owners to securely outsource their data to the cloud for storage. The outsourced encrypted data can be processed by the cloud server to achieve commonly-used plaintext arithmetic operations without involving additional servers. Specifically, we design both signed and unsigned integer circuits using a fully homomorphic encryption (FHE) scheme, construct a new packing technique (hereafter referred to as integer packing), and extend the secure circuits to its packed version. This achieves significant improvements in performance compared with the original secure signed/unsigned integer circuit. The secure integer circuits can be used to construct a new data mining application, which we refer to as secure k-nearest neighbours classifier, without compromising the privacy of original data. Finally, we prove that the proposed Pockit achieves the goal of secure computation without privacy leakage to unauthorized parties, and demonstrate the utility and efficiency of Pockit.

Index Terms—Privacy-preserving, outsourced computation, fully homomorphic encryption, cloud privacy

1 Introduction

With increasing digitization of our society (e.g., trend in Internet and Cloud of Things), significantly more data are created by digital devices. For example, it was estimated that data created by the Internet of Things (IoT) devices will be 507.5 ZB per year (42.3 ZB per month) by 2019, as compared to 134.5 ZB per year (11.2 ZB per month) in 2014 [1]. Due to the limited storage and computation ability of IoT devices [2], outsourced massive data and heavy-weight computation to the CP become more and more attractive. It is because cloud computing offering real-time and anytime, anywhere storage with large or unlimited capacity is a popular option for users, individuals and organizations (e.g., the U.S Federal Government's Cloud First policy [3]).

To mine or analyze data in the cloud, a number of techniques such as data mining have been proposed and deployed. For instance, Amazon uses item-to-item collaborative filtering recommendation [4] to help identify customer buying patterns and trends that lead to improved quality of customer service. Deep learning tools [5], [6] have also been used to integrate multiple modalities of very large, complex patient data in order to allow scientists and doctors to better predict clinical outcomes and work toward cures for diseases such as cancer [7].

In order to enjoy benefits afforded by the use of the remote data mining/analytics, the user's data need to outsource to the cloud or service provider for computation. For example, a patient can use his/her electronic health record (EHR) to achieve the remote disease prediction with some data mining techniques constructed by the hospital (for training the disease prediction model may cost hundreds of million dollars). However, data security and privacy remain areas of ongoing focus [8], [9], [10], [11]. One of the main challenges is to ensure that outsourced user data protected from unauthorized disclosure [12]. Without adequate protection, data owners may be reluctant to outsource personal or confidential data to a cloud storage. In order to perform data mining and other analytical tasks on outsourced data, basic commonly-used arithmetic operations, including comparison and multiplication, need to be supported. In the event that outsourced data stored in the cloud are encrypted, performing these essential arithmetic operations (directly on the encrypted data) without compromising the privacy of the original data remains a research challenge. For example, a number of frameworks designed to process encrypted data in outsourced cloud environment have been proposed in the literature [13], [14], [15], [16], [17], [18]; however, these existing frameworks generally require either multi-round communications between the user and the cloud, or additional server to perform the computations. That leads to additional energy/electricity

- *X. Liu and Y. Yang are with the School of Information Systems, Singapore Management University, Singapore 188065 and also with the College of Mathematics and Computer Science/College of Software, Fuzhou University, Fuzhou 350108, China. E-mail: {snbnix, yang.yang.research}@gmail.com.*
- *R.H. Deng and H. Pang are with the School of Information Systems, Singapore Management University, Singapore 188065. E-mail: {robertdeng, hhpang}@smu.edu.sg.*
- *K.-K.R. Choo is with the Department of Information Systems and Cyber Security, University of Texas at San Antonio, San Antonio, TX 78249-0631. E-mail: raymond.choo@fulbrightmail.org.*

consumption (which can be unrealistic for resource-constrained IoT devices) and increases the chance of data leakage, respectively. Thus, in this paper, we seek to address the following research challenge: *How do we design a system to securely perform commonly-used arithmetic operations on-the-fly without involving additional (non-colluding) server(s)?*

Specifically, in this paper, we propose a **P**rivacy-Preserving **O**utsourced **C**alculation Tool**kit** for the cloud computing environment, hereafter referred to as *Pockit*, with the following capabilities.

- *Secure Data Storage:* Pockit allows all parties in the system to outsource their data to a cloud platform for secure storage without compromising the privacy of the data.
- *Secure Data Processing On-the-fly:* Outsourced data can be securely computed and processed on-the-fly, including commonly used unsigned integer computation as well as signed integer computation. Moreover, our Pockit can be extended to support fix-point number calculation.
- *Computation without Additional Servers:* In most existing computation frameworks such as [13], [14], [15], [16], [17], [18], two non-colluding servers (including an additional server with decryption ability) are required to perform some integer calculation, such as multiplication, comparison, etc. In Pockit, all integer computations can be executed by a single server without involving additional (decryption) server. This decreases the chance of data leakage.
- *Support Iterative Calculations:* In order to perform unlimited iterative calculations with large circuit, the ciphertext should support the 'refresh' property. Moreover, for integer calculation, the overflow problem should be addressed such that newly computed ciphertext can be directly used as the input for the next secure computation.
- *Ease of Use:* Pockit does not require the data owner to perform any complex pre-processing prior to outsourcing. A data owner only needs to encrypt-and-outsource the data to the cloud. Moreover, interactions between data owner and the cloud server are kept to a minimal since a data owner only needs to send a query to the cloud server, for it to perform the computation, and respond with the computed results in a single round.

Technique Overview. In Pockit, we use one of the most efficiency fully homomorphic encryption scheme at present called BGV scheme as a basis. To achieve the outsourced arithmetic operations, we construct commonly- used unsigned/signed integer circuits (e.g., integer addition, comparison, multiplication, division) for both single-key and multiple-key setting. Also, we design a new integer packing technique for Pockit to store multiple integers in a single ciphertext according to the Single Instruction Multiple Data (SIMD) feature of BGV. More importantly, the basic commonly-used unsigned/signed integer circuits can be extended to support packed ciphertext. It can simultaneously manipulate multiple integers which are belonged to a single packed ciphertext. Also, the KeySwitch algorithm in BGV can be used to extend the single-key computation circuits to single-key settings, and we use above packed integer circuits to construct an important application called secure k-NN classifier for the outsourced scenario.

TABLE 1
Summary of Notations

Notation	Definition
pk_a/sk_a	Party a' public key/private key
$\Phi_m(X)$	m's cyclotomic polynomial
$\mathbb{A}$	A polynomial ring $\mathbb{Z}[X]/\Phi_m(X)$
$\mathbb{A}_n$	A polynomial ring $(\mathbb{Z}/n\mathbb{Z})[X]/\Phi_m(X)$
$a^\star, b^\star$	Element belonged to $\mathbb{A}_2$
$\tilde{\mathbf{c}}, \tilde{\mathbf{k}}, \tilde{\mathbf{t}}$	BGV ciphertexts
$\tilde{\mathbf{c}}(a^\star)$	BGV ciphertext with plaintext $a^\star$
$\mathfrak{a}, \mathfrak{b}$	Element belonged to $\mathbb{Z}_2$
$\|S\|$	the size of the set S

2 Background

In this section, we present the definition of a basic crypto primitive called BGV scheme, and some basic circuits as the building blocks to construct Pockit. Table 1 summarizes the key notations used in this paper.

2.1 Fully Homomorphic Encryption Scheme–BGV Scheme

The BGV scheme [19] contains five basic algorithms: `Setup` for system parameter setup, `KeyGen` for private key generation, `PKGen` for public key generation, `Enc` for message encryption, and `Dec` for ciphertext decryption (the construction of these five algorithms and basic homomorphic operations can be found in supplemental material Section B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2018.2816656). Here, we only provide three basic homomorphic operations provided by BGV scheme. Given two ciphertexts $\tilde{\mathbf{c}}(a^\star)$ and $\tilde{\mathbf{c}}(b^\star)$ under a common public key, homomorphic addition $\texttt{H.add}(\tilde{\mathbf{c}}(a^\star); \tilde{\mathbf{c}}(b^\star))$ generates a new ciphertext for the underlying plaintext $a^\star + b^\star \in \mathbb{A}_2$, while homomorphic multiplication $\texttt{H.mul}(\tilde{\mathbf{c}}(a^\star); \tilde{\mathbf{c}}(b^\star))$ produces a ciphertext for plaintext $a^\star \times b^\star \in \mathbb{A}_2$. Moreover, homomorphic constant multiplication $\texttt{H.cmul}(\tilde{\mathbf{c}}(a^\star); k^\star)$ computes a new ciphertext for plaintext $a^\star \times k^\star \in \mathbb{A}_2$, where $k^\star \in \mathbb{A}_2$. We omit the construction of the three homomorphic operations (here, and refer reader to supplemental material Section B, available online, or [20] for details).

In order to achieve iterative computation, two methods are used to refresh a ciphertext, namely, Modulus-Switching (denoted as `ModSwitch`) & Key-Switching (denoted as `KeySwitch`) [19] technique, and Bootstrapping (denoted as `Bootstrap`) technique [21]. The former technique first using `ModSwitch` to convert a level-$(i+1)$ ciphertext into a level-i ciphertext without changing the private key (with the noise in the ciphertext reduced by a factor of q_i/q_{i+1}). Then, it uses `KeySwitch` procedure with transformation key $W_{\mathbf{s}_{i+1} \to \mathbf{s}_i}$ to transform the ciphertext with respect to level-$(i+1)$ key $\mathbf{s}_{i+1}$ into a ciphertext with respect to level-i key $\mathbf{s}_i$. Once the ciphertext reaches level-0, the embedded noise in it can no longer be reduced through `ModSwitch` & `KeySwitch`. At this stage, the `Bootstrap` is needed to "re-encrypt" the ciphertext $\tilde{\mathbf{c}}$ at level-0 to generate a new

ciphertext that encrypts the same element with respect to some higher level $i < (L-1)$. In other words, the level-0 private key $\mathbf{s}_0$ can be encrypted to level-$(L-1)$ ciphertext, and the ciphertext $\tilde{\mathbf{c}}$ can be encrypted as level-$(L-1)$ ciphertext. The scheme can homomorphically perform the decryption circuit inside the level-$(L-1)$ ciphertext, and finally to obtain the new ciphertext at level i ($i < (L-1)$), the noise control technique should be applied during re-encryption.

For ease of reading, we will omit the level representation and use $\tilde{\mathbf{c}}^{\langle pk_i \rangle}(a^\star)$ to represent data owner i's ciphertext (i.e., encryption of plaintext $a^\star \in \mathbb{A}_2$, and pk_i is data owner i's public key). We also omit $\langle pk_i \rangle$ in $\tilde{\mathbf{x}}^{\langle pk_i \rangle}$, and use $\tilde{\mathbf{x}}$ instead if all the ciphertexts belong to the same user.

2.2 Basic Unsigned Integer Circuits

Given two unsigned μ-bit integers $\mathbf{a} = (\mathfrak{a}_{\mu-1}, \ldots, \mathfrak{a}_0)$ and $\mathbf{b} = (\mathfrak{b}_{\mu-1}, \ldots, \mathfrak{b}_0)$, the following circuits output the result for the various functions. Note that $\oplus$ denotes bit-wise XOR operation while $\wedge$ denotes bit-wise AND operation. Also, we use $\mathbf{a}_{\mathtt{ten}} = \sum_{j=0}^{\mu-1} \mathfrak{a}_j 2^j$ to represent the integer value of $\mathbf{a}$, where $\mathbf{a}$ is the binary representation of $\mathbf{a}_{\mathtt{ten}}$. The operation $\mathbf{a}_{\mathtt{ten}} \circ \mathbf{b}_{\mathtt{ten}}$ denote the integer operation $\circ \in \{+, \times, \geq, <\}$ between integers $\mathbf{a}_{\mathtt{ten}}$ and $\mathbf{b}_{\mathtt{ten}}$. See supplemental material Section C, available online, and [16] and supplemental material, available online, for the construction of addition circuit, comparison circuit, and equality circuit (their secure version are called secure unsigned integer addition (`I.add`), secure unsigned integer comparison (`I.cmp`), and secure unsigned integer equality (`I.equ`)). Here, we give two unsigned integer multiplication and division circuits before constructing their corresponding secure versions.

Multiplication Circuit. The circuit produces a $2 \cdot \mu$-bit result $\mathbf{n}$. If initially $\mathbf{n} = \mathbf{0}$, then we use the integer addition circuit to add $\mathbf{c}_i$ ($i = 0$ to $\mu - 1$) to $\mathbf{n}$, where $\mathbf{c}_i$ is a 2μ length vector $(\mathfrak{c}_{i,2\mu-1}, \ldots, \mathfrak{c}_{i,0})$ with $\mathfrak{c}_{i,j+i} = \mathfrak{a}_j \wedge \mathfrak{b}_i$ for every $0 \leq i, j \leq \mu - 1$, and let $\mathfrak{c}_{i,t} = 0$ for every $0 \leq t \leq 2\mu - 1$ and $t \neq i + j$. Finally, it takes the 2μ least significant bits as the final output.

Division Circuit. Given dividend $\mathbf{a}$ and divisor $\mathbf{b}$, the circuit outputs a μ-bit quotient $\mathbf{q}$ and remainder $\mathbf{r}$ as follows. Construct a 2μ-bit vector $\mathbf{e}$, and denote the μ least significant bits as $(\mathfrak{e}_{\mu-1}, \ldots, \mathfrak{e}_0) = (\mathfrak{q}_{\mu-1}, \ldots, \mathfrak{q}_0) \leftarrow \mathbf{q}^*$ and the μ most significant bits as $(\mathfrak{e}_{2\mu-1}, \ldots, \mathfrak{e}_\mu) = (\mathfrak{r}_{\mu-1}, \ldots, \mathfrak{r}_0) \leftarrow \mathbf{r}^*$. Initialize $\mathbf{q}^* \leftarrow \mathbf{a}$ and $\mathbf{r}^* \leftarrow \mathbf{0}$. The following procedure will then execute for μ times: Shift $\mathbf{e}$ left by 1 bit, and compare $\mathbf{r}^*_{\mathtt{ten}}$ and $\mathbf{b}_{\mathtt{ten}}$. If $\mathbf{r}^*_{\mathtt{ten}} < \mathbf{b}_{\mathtt{ten}}$, denote $\mathfrak{e}_0 \leftarrow 0$, $\mathbf{d} \leftarrow \mathbf{0}$, and calculate $\mathbf{q}^*_{\mathtt{ten}} \leftarrow \mathbf{q}^*_{\mathtt{ten}} + \mathbf{d}_{\mathtt{ten}}$. If $\mathbf{r}^*_{\mathtt{ten}} \geq \mathbf{b}_{\mathtt{ten}}$, let $\mathfrak{e}_0 = 1$, $\mathbf{d}_{\mathtt{ten}} \leftarrow -\mathbf{b}_{\mathtt{ten}}$ and calculate $\mathbf{q}^*_{\mathtt{ten}} \leftarrow \mathbf{q}^*_{\mathtt{ten}} + \mathbf{d}_{\mathtt{ten}}$. After μ rounds of calculation, the outputs produced are $\mathbf{q} \leftarrow (\mathfrak{e}_{\mu-1}, \ldots, \mathfrak{e}_0)$ and $\mathbf{r} \leftarrow (\mathfrak{e}_{2\mu-1}, \ldots, \mathfrak{e}_\mu)$.

3 System Model and Privacy Requirement

3.1 System Model

The system model in Pockit comprises a key generation center (KGC), a cloud platform (CP), and data users (DUs)–see Fig. 1.

1. *KGC* is an entity trusted by all the other entities in the system, and tasked with distributing and managing all public and private keys.
2. *CP* effectively has 'unlimited' data storage space, and stores and manages encrypted data outsourced from the registered parties in the system. Also, CP provides computation power to perform homomorphic operations over encrypted data.
3. Generally, a *DU* uses its public key to encrypt data, before storing the encrypted data with CP. The data user may also request CP to perform calculations on the outsourced data.

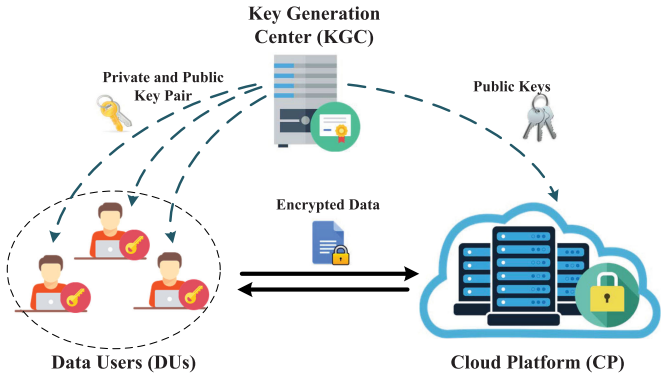


Fig. 1. System model.

We assume the system contains S DUs, one KGC, and one CP. KGC first generates public-private key pairs for the S DUs (denote as (pk_j, sk_j), $j = 0$ to $S-1$). Also, the transformation keys $W^{(j)}_{\mathbf{s}_{i+1} \to \mathbf{s}_i}$ for DU j to refresh ciphertext is sent to CP for storage.

3.2 Attack Model

In attack model, CP, DUs are *curious-but-honest* parties, which strictly follow the protocol, but are interested to learn data belonging to the challenger DU. Therefore, we introduce an active adversary $\mathcal{A}$ in our model. The goal of $\mathcal{A}$ is to decrypt challenger DU's ciphertext, and $\mathcal{A}$ possesses with the following capabilities: 1) $\mathcal{A}$ may *eavesdrop* on all the public communication links to obtain encrypted data. 2) $\mathcal{A}$ may *compromise* CP's storage which stores all the parities ciphertext. The goal of $\mathcal{A}$ is to guess the plaintext values of ciphertexts encrypted by the challenger DU's public key, even to the extent of including CP to collude with non-challenger DUs.

Adversary $\mathcal{A}$ is, however, restricted from compromising the challenger DU. We remark that such restrictions are typical in adversary models used in [18].

4 Secure SIMD Unsigned Integer Circuit and Integer Packing Technique

In this section, we explain how an integer plaintext is mapped to a ciphertext, create the basic secure SIMD circuits over ciphertexts, present our packing techniques and finally achieve the integer packed version of secure SIMD circuits.

4.1 System Initialization

Given an appropriate m, we obtain m's cyclotomic polynomial $\Phi_m(X)$, and decompose $\Phi_m(X) \bmod 2$ into $\ell = \phi(m)/d$ irreducible polynomials such that each has the same degree d, i.e., $\Phi_m(X) = \prod_{j=0}^{\ell-1} F_j(X) (\bmod\ 2)$. Thus, we have the isomorphism

$$\mathbb{A}_2 \cong \mathbb{L}_{\ell-1} \times \cdots \times \mathbb{L}_0 := A_2,$$

where $\mathbb{L}_i = (\mathbb{Z}/2\mathbb{Z})[X]/F_i(X)$ (for $i = 0, \ldots, \ell-1$). Note that the rings $\mathbb{L}_i$ are all isomorphic to $\mathbb{L}_0$, and their direct

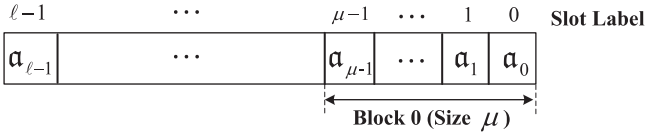


Fig. 2. Data format.

product A_2 is isomorphic to $\mathbb{A}_2$. Thus, we can think of operations in $\mathbb{A}_2$ as operations on $\mathbb{L}_{\ell-1} \times \cdots \times \mathbb{L}_0$; we call $\mathbb{L}_{\ell-1}, \ldots, \mathbb{L}_0$ as slot $\ell-1$ to slot 0 respectively. In designing the integer circuits, we encode a single bit in one slot for storage. Formally, we choose an irreducible polynomial $G(X) = X+\ 1 \in (\mathbb{Z}/2\mathbb{Z})[X]$ and define $\mathbb{K} = (\mathbb{Z}/2\mathbb{Z})[X]/G(X)$. Also, we define Ψ_i as a distinct homomorphic embedding from $\mathbb{K}$ into $\mathbb{L}_i = (\mathbb{Z}/2\mathbb{Z})[X]/F_i(X)$. Our basic plaintext space is defined as ℓ copies of $\mathbb{K}$, i.e., $\mathcal{M} = (\mathbb{K})^\ell$, where addition and multiplication are defined componentwise. We can therefore define a map

$$\Psi^* : \begin{cases} \mathcal{M} & \to A_2 \\ (\mathfrak{a}_{\ell-1}, \ldots, \mathfrak{a}_0) & \mapsto (\Psi_{\ell-1}(\mathfrak{a}_{\ell-1}), \ldots, \Psi_0(\mathfrak{a}_0)) \end{cases},$$

which map ℓ-bit plaintext to A_2. Using the Chinese Remainder Theorem (CRT), we can further batch element $\mathbf{a} \in A_2$ into $a^\star \in \mathbb{A}_2$ (denote as $a^\star = \mathtt{CRT}_2(\mathbf{a})$), and encrypt it as ciphertext. Using this method, we can "pack" a ℓ-bit message into a single ciphertext for storage.

Moreover, we can use another operation to manipulate elements in $\mathbb{A}_2$ called automorphism φ_j of the form $a^\star(X) \to a^\star(X^j)$, where $j \in \mathbb{Z}_m^*$ and $a^\star(X), a^\star(X^j) \in \mathbb{A}_2$. Using the automorphism, we can achieve slot rotation by implementing φ_{g^k} ($k = 1$ to $\ell-1$) for some element $g \in \mathbb{Z}_m^*$, where g has the order ℓ in both group $\mathbb{Z}_m^*$ and the quotient group $\mathbb{Z}_m^*/\langle 2\rangle$. That is, we can use φ_{g^k} to rotate the content of slot by k positions, thus moving the content of slot j to slot $j + k \bmod \ell$. Using the automorphism, we define another BGV operation called plaintext slot rotation, and we denote $\tilde{\mathbf{c}}' \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}; k)$. For more details, the interested reader is referred to [22], [23]. Next, we show how to achieve basic SIMD circuit calculations.

4.2 Secure SIMD Unsigned Integer Computation Circuit

Given the decimal representation unsigned of an integer $\mathbf{a}_{\mathtt{ten}}$, we can transform the integer into binary form $\mathbf{a} = (\mathfrak{a}_{\mu-1}, \ldots, \mathfrak{a}_0)$–see Table 2. We then use slot 0 to $\mu-1$ to store the μ-bit integer, i.e., pad $\ell-\mu$ "0" to the top of $\mathbf{a}$ to derive ℓ-bit vector $\mathbf{a} = (0, \ldots, 0, \mathfrak{a}_{\mu-1}, \ldots, \mathfrak{a}_0) \in \mathbb{Z}_2^\ell$ (usually, $\ell \gg \mu$). Then, we use the above method to encode the plaintext as $a^\star = \mathtt{CRT}_2(\Psi^*(\mathbf{a})) \in \mathbb{A}_2$, encrypt $a^\star$, and denote the ciphertext as $\tilde{\mathbf{c}}(a^\star)$ (or $\tilde{\mathbf{c}}(\mathtt{CRT}_2(\Psi^*(\mathbf{a})))$). We simply write the $\tilde{\mathbf{c}}(\mathtt{CRT}_2(\Psi^*(\mathbf{a})))$ as $\tilde{\mathbf{c}}(\mathbf{a})$ to make the notation more readable. Within a ciphertext, each μ-slot is considered as a block. We use block 0 (slot 0 to $\mu-1$) to store the integer; such a ciphertext is an *Unpacked Ciphertext* (see Fig. 2). For unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{a})$, we may easily verify that $\mathbf{a}_{\mathtt{ten}} = \sum_{j=0}^{\ell-1} \mathfrak{a}_j 2^j$. Before constructing the secure circuit, we define $\pi^\star{}_i, \bar{\pi}^\star{}_i \in \mathbb{A}_2$ where $\pi^\star{}_i$ stores 1 in slot i ($0 \le i < \ell$) and 0 in the other slots, while $\bar{\pi}^\star{}_i$ stores 0 in slot i and 1 in the other slots. Moreover, we define $\pi^\star{}_{I_{\mu-1}} \in \mathbb{A}_2$ which stores 1 from slot 0 to slot $\mu-1$, and 0 in the other slots.

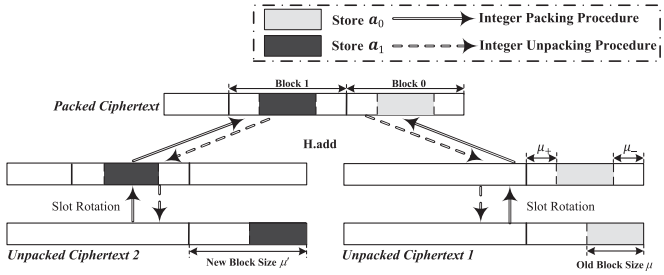


Fig. 3. Integer packing/unpacking procedure.

Note that `H.add` can be considered the bit-wise XOR operation for the individual plaintext slots of two ciphertexts, while `H.mul` can be considered the bit-wise AND operation for the individual plaintext slots.

4.2.1 *Secure Slot Copy* – $\mathtt{Scpy}(\tilde{\mathbf{c}}(\mathbf{a}); k)$

Given an unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{a})$ with data stored in slot z and 0 in the other slots, the `Scpy` algorithm outputs $\tilde{\mathbf{c}}(\mathbf{n})$ where $\mathbf{n} = \{\mathfrak{n}_{\ell-1}, \ldots, \mathfrak{n}_0\}$ with $\mathfrak{n}_{k+z} = \cdots = \mathfrak{n}_z = \mathfrak{a}_z$ and 0 otherwise. The algorithm works as follows:

1) Intialize $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \tilde{\mathbf{c}}(\mathbf{a})$.
2) If $k > 0$, calculates the following recurrently for $i = 1$ to k, $\tilde{\mathbf{c}}_i \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}(\mathbf{a}), i)$ and $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}_i)$.

Otherwise, $k < 0$ and calculate the following recurrently for $i = -1$ to k,

$$\tilde{\mathbf{c}}_{-i} \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}(\mathbf{a}), \ell + i);\ \tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}_{-i}).$$

4.2.2 *Secure Unsigned Integer SIMD Multiplication*

Given two unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{b})$, we obtain $\tilde{\mathbf{c}}(\mathbf{n})$ where $\mathbf{n} = \{\mathfrak{n}_{\ell-1}, \ldots, \mathfrak{n}_0\}$ with $\mathfrak{n}_{\ell-1} = \cdots = \mathfrak{n}_{2\mu} = 0$ and $\mathfrak{n}_{2\mu-1}, \ldots, \mathfrak{n}_0$ storing the multiplication result, and denote the algorithm as $\mathtt{I.mul}(\tilde{\mathbf{c}}(\mathbf{a}), \tilde{\mathbf{c}}(\mathbf{b}))$.

1) First, $\tilde{\mathbf{c}}(\mathbf{n})$ is set to 0. Then, for $i = 0$ to $\mu-1$, the following calculations are executed recurrently

$$\tilde{\mathbf{c}}_i \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{b}), \pi^\star{}_i); \tilde{\mathbf{k}}_i \leftarrow \mathtt{Scpy}(\tilde{\mathbf{c}}_i, \mu-1);$$
$$\tilde{\mathbf{t}}_i \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}(\mathbf{a}), i); \tilde{\mathbf{c}}'_i \leftarrow \mathtt{H.mul}(\tilde{\mathbf{t}}_i, \tilde{\mathbf{k}}_i).$$

2) We add $\tilde{\mathbf{c}}'_0, \ldots, \tilde{\mathbf{c}}'_{\mu-1}$ together using `I.add`, i.e.,

$$\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{I.add}(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}'_i). \quad (1)$$

After the calculation, $\tilde{\mathbf{c}}(\mathbf{n})$ contains the final output with information stored in block 0 (which has block size 2μ).

4.3 Secure Packed Ciphertext Storage & Calculation

Beside using SIMD technique to accelerate the integer circuits [16], we also design a new technique called integer packing/unpacking technique to further increases both storage and computation efficiency for an integer circuit. Given α' unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a}_{\alpha'-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0)$ ($0 < \alpha' \le \lfloor \ell/\mu' \rfloor$), each ciphertext uses slot 0 to $\mu-1$ to store information (a.k.a., block label 0 with block size μ). The goal of integer packing is to better utilize a ciphertext $\tilde{\mathbf{c}}_{sp}$ such that its block i stores $\mathbf{a}_i$. (Fig. 3 illustrates packing two unpacked ciphertext into a packed one). To avoid overflow across different blocks, the new block size μ' ($\mu' = \mu_+ + \mu_- + \mu$) is related to different types of calculations, where μ_+ and μ_- are the parameters

used to solve computation overflow and underflow problems. For example, we can choose $\mu' \geq \mu + 1$ if we only need one round of secure integer addition, where $\mu_+ \geq 1$ and $\mu_- = 0$. For a circuit for once-off unsigned integer comparison, we need $\mu' \geq 2^{\lceil \log_2 \mu \rceil}$ where $\mu_+ \geq 2^{\lceil \log_2 \mu \rceil} - \mu$ and $\mu_- = 0$. If we only need to store information without any integer calculation, then we simply have $\mu' = \mu$. Next, we will show how to construct the integer packing/unpacking technique.

Integer Packing (`Ipack`): The `Ipack` first initializes $\tilde{\mathbf{c}}_{sp} \leftarrow \tilde{\mathbf{c}}(0)$. Then, for $i = 0, \ldots, \alpha' - 1$, the algorithm calculates the following two formulas recurrently

$$\tilde{\mathbf{c}}'_i \leftarrow \texttt{H.rotate}(\tilde{\mathbf{c}}(\mathbf{a}_i), \mu' \cdot i + \mu_-); \tilde{\mathbf{c}}_{sp} \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_{sp}, \tilde{\mathbf{c}}'_i).$$

Finally, the algorithm outputs $\tilde{\mathbf{c}}_{sp}$

$$\tilde{\mathbf{c}}_{sp} \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{a}_{\alpha'-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0) : \mu'),$$

which uses block i (slot $\mu' i$ to $\mu' i + \mu' - 1$) to store plaintext $\mathbf{a}_i$.

Integer Unpacking (`IUpack`): Given a packed ciphertext $\tilde{\mathbf{c}}_{sp}$, we can recover $\tilde{\mathbf{c}}(\mathbf{a}_{\alpha'-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0)$ as follows.

For $i = 0, \ldots, \alpha' - 1$, recurrently calculate the following $\tilde{\mathbf{c}}' \leftarrow \texttt{H.rotate}(\tilde{\mathbf{c}}_{sp}, \ell - \mu' i - \mu_-)$ and $\tilde{\mathbf{c}}(\mathbf{a}_i) \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}', \pi^{\star}{}_{I'})$, where $\pi_{I'^{\star}} \in \mathbb{A}_2$ stores 1 from slot 0 to $\mu' - 1$, while the other slots store 0.

Extension for Packed Ciphertext Calculation. The SIMD property allows the manipulation of block 0 to be reflected to all the other blocks simultaneously. Thus, we can use the unpacked SIMD circuit to achieve packed SIMD calculation, with changes to the following parameters: The slot selection parameter $\pi^{\star}{}_j$ (in which slot j ($0 \leq j < \mu$) stores 1, and the other slots store 0, where μ is the block size) is changed to $\eta^{\star}{}_{j,\mu'}$ (in which slot $j + \mu_- + k\mu'$ stores 1 and the other slots store 0, where $k = 0$ to $\alpha' - 1$). Also, $\bar{\pi}^{\star}{}_j$ (in which slot j ($0 \leq j < \mu$) stores 0, and the other slots store 1, where μ is the block size) is changed to $\bar{\eta}^{\star}{}_{j,\mu'}$ (in which slot $j + \mu_- + k\mu'$ stores 0 and the other slots store 1, where $k = 0$ to $\alpha' - 1$). Moreover $\pi^{\star}{}_{I_{\mu-1}}$ (in which slots 0 to $\mu - 1$ store 1, and the other slots store 0) is changed to $\eta^{\star}{}_{I_{\mu-1},\mu'}$ (in which slots $k\mu' + \mu_-$ to slot $\mu - 1 + k\mu' + \mu_-$ store 1 for $k = 0$ to $\alpha' - 1$, and the other slots store 0) for all the integer calculations. Furthermore, in `PI.add`, the ciphertext $\tilde{\mathbf{t}}_{0,j}$ in `I.add` is changed so that the ciphertext stores 1 in slot $j + k\mu' + \mu_-$ ($j = 0, \ldots, \mu - 1$, $k = 0, \ldots, \alpha' - 1$) and 0 in the other slots. The above transformation from unpacked integer SIMD computation to packed one is simple, and we denote the packed integer secure SIMD addition, equality, comparison and multiplication as `PI.add`, `PI.equ`, `PI.cmp` and `PI.mul`,[1] respectively. We next show how to use the packing technique to construct the secure protocols and optimize performance.

4.3.1 Secure Group Integer Sum Circuit (`GSum`)

Given unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a}_0), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_k)$, the goal is to calculate unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{n})$ such that $\mathbf{n}$ stores the integer sum of $\mathbf{a}_0, \ldots, \mathbf{a}_k$. A naive solution is to directly use `I.add` to add these ciphertexts successively. However, the overhead of such a naive solution is high when the number of inputs is large. Instead, we use `GSum` to improve the running efficiency, with the algorithm executing as follows:

1. Beside changing $\pi^{\star}{}_i$ to $\eta^{\star}{}_{i,\mu'}$, `I.add` in step 2 of `PI.mul` also needs to be changed to `PI.add`.

Put $\tilde{\mathbf{c}}(\mathbf{a}_0), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_k)$ in a set S, and denote them as $\tilde{\mathbf{c}}_0, \ldots, \tilde{\mathbf{c}}_{\|S\|-1}$. The following procedure is executed recurrently until only one element is left in S, i.e., if $\|S\| = 1$, the remaining element is the final output $\tilde{\mathbf{c}}(\mathbf{n})$; otherwise, the `GSum` processes according to the conditions.

- If $\|S\| \bmod 2 = 0$ and $\|S\| > 1$, 1) for $i = 0, \ldots, \lfloor \|S\|/(2\alpha) \rfloor$, calculate $\tilde{\mathbf{c}}_i \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}_{2\alpha i+2(\alpha-1)}, \ldots, \tilde{\mathbf{c}}_{2\alpha i} : \mu')$; $\tilde{\mathbf{c}}'_i \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}_{2\alpha i+2\alpha-1}, \ldots, \tilde{\mathbf{c}}_{2\alpha i+1} : \mu')$; $\tilde{\mathbf{c}}^*_i \leftarrow \texttt{PI.add}(\tilde{\mathbf{c}}_i, \tilde{\mathbf{c}}'_i)$ and $\{\tilde{\mathbf{c}}^*_{\alpha i+\alpha-1}, \ldots, \tilde{\mathbf{c}}^*_{\alpha i}\} \leftarrow \texttt{IUpack}(\tilde{\mathbf{c}}^*_i, \mu')$, where $\alpha = \lfloor \ell/\mu' \rfloor$, and add $\{\tilde{\mathbf{c}}^*_{\alpha i+\alpha-1}, \ldots, \tilde{\mathbf{c}}^*_{\alpha i}\}$ to set S'; 2) clear set S and let $S \leftarrow S'$.
- If $\|S\| \bmod 2 \neq 0$ and $\|S\| > 1$, pop the last element $\tilde{\mathbf{c}}_{\|S\|-1}$ from set S so that $\|S\|/2 = 0$. Execute the above procedure ($\|S\|/2 = 0$ and $\|S\| > 1$) to generate $\tilde{\mathbf{c}}^*_0, \ldots, \tilde{\mathbf{c}}^*_{(\|S\|-1)/2-1}$, clear set S, put $\tilde{\mathbf{c}}^*_0, \ldots, \tilde{\mathbf{c}}^*_{(\|S\|-1)/2-1}$, $\tilde{\mathbf{c}}_{\|S\|-1}$ into set S, and denote the algorithm as $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{GSum}(\tilde{\mathbf{c}}(\mathbf{a}_0), \ldots, \tilde{\mathbf{c}}(\mathbf{n}_k) : \mu')$.

Optimization of `I.mul`: Using `GSum`, we can further boost the running time of `I.mul`. Step-1 is similar to that for `I.mul`. For step-2, to sum up $\tilde{\mathbf{c}}'_0, \ldots, \tilde{\mathbf{c}}'_{\mu-1}$

$$\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{GSum}(\tilde{\mathbf{c}}'_0, \ldots, \tilde{\mathbf{c}}'_{\mu-1} : 2 \cdot \mu). \tag{2}$$

For the optimization of `PI.mul`, if $2\mu \cdot \eta \cdot > \ell/2$, no optimization method may be done as the ciphertext cannot be further packed. If $2\mu \cdot \eta \cdot \leq \ell/2$, to sum up $\tilde{\mathbf{c}}'_0, \ldots, \tilde{\mathbf{c}}'_{\mu-1}$, we combine $k^* = \lfloor \ell/(2\mu \cdot \eta) \rfloor$ partially packed ciphertext into a fully packed ciphertext, i.e., we can `GSum` to add $2k^*$ integers at the same time, which would improve performance by up to k^* times as compared to `PI.add`. Next, we use the packing technique to achieve another algorithm called Secure Group Integers Min Circuit (`GMin`).

4.3.2 Secure Group Integer Min Circuit (`GMin`)

Given unpacked ciphertexts $T_0 = (\tilde{\mathbf{c}}(\mathbf{a}_0), \tilde{\mathbf{c}}(\mathbf{I}_0)), \ldots, T_k = (\tilde{\mathbf{c}}(\mathbf{a}_k), \tilde{\mathbf{c}}(\mathbf{I}_k))$, the goal is to calculate the unpacked ciphertext $T = (\tilde{\mathbf{c}}(\mathbf{a}), \tilde{\mathbf{c}}(\mathbf{I}))$ such that $\mathbf{a}$ stores the minimum integer value among $\mathbf{a}_0, \ldots, \mathbf{a}_k$ and $\mathbf{I}$ is $\mathbf{a}$'s corresponding identity. The algorithm executes as follows: Put $T_0, \ldots, T_k$ in a set S. The following procedure is executed recurrently until one tuple is left in S, i.e., if $\|S\| = 1$, the tuple remaining is the final output; otherwise, the `GMin` processes according to the conditions.

- If $\|S\| \bmod 2 = 0$ and $\|S\| > 1$, 1) for $i = 0, \ldots, \lfloor \|S\|/(2\alpha) \rfloor$, pack the ciphertexts into a packed ciphertext before performing the comparison

  $$\tilde{\mathbf{c}}_i \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{a}_{2\alpha i+2(\alpha-1)}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_{2\alpha i}) : \mu');$$
  $$\tilde{\mathbf{c}}_{id,i} \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{I}_{2\alpha i+2(\alpha-1)}), \ldots, \tilde{\mathbf{c}}(\mathbf{I}_{2\alpha i}) : \mu'),$$

  where $\alpha = \lfloor \ell/\mu' \rfloor$. Similar, pack $\tilde{\mathbf{c}}(\mathbf{a}_{2\alpha i+2\alpha-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_{2\alpha i+1})$ into ciphertext $\tilde{\mathbf{c}}'_i$ and pack $\tilde{\mathbf{c}}(\mathbf{I}_{2\alpha i+2\alpha-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{I}_{2\alpha i+1})$ into $\tilde{\mathbf{c}}'_{id,i}$. Moreover, compare the plaintext relationship between $\tilde{\mathbf{c}}_i$ and $\tilde{\mathbf{c}}'_i$. If the integer in block j of $\tilde{\mathbf{c}}_i$ is less than the corresponding block of $\tilde{\mathbf{c}}'_i$, then the algorithm selects $\mathbf{a}_{2\alpha i+2j}$ and $\mathbf{I}_{2\alpha i+2j}$. Otherwise, the algorithm selects $\mathbf{a}_{2\alpha i+2j+1}$ and $\mathbf{I}_{2\alpha i+2j+1}$, i.e., compute $\tilde{\mathbf{c}}_p \leftarrow \texttt{PI.cmp}(\tilde{\mathbf{c}}_i, \tilde{\mathbf{c}}'_i)$; $\tilde{\mathbf{c}}'_p \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_p, \tilde{\mathbf{c}}(\eta^{\star}{}_{0,\mu'}))$; $\tilde{\mathbf{c}}_h \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}_p, \mu - 1)$; $\tilde{\mathbf{c}}'_h \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}'_p, \mu - 1)$, such that each slot of block j in $\tilde{\mathbf{c}}_h$ stores 1 if $\mathbf{a}_{2\alpha i+2j} < \mathbf{a}_{2\alpha i+2j+1}$ and 0 otherwise, while $\tilde{\mathbf{c}}'_h$ stores exclusive OR value

for each slot of $\tilde{\mathbf{c}}_h$. Next, both `H.mul` and `H.add` can be used to get $\tilde{\mathbf{c}}_i^*$ and $\tilde{\mathbf{c}}_{id,i}^*$, i.e., compute $\tilde{\mathbf{c}}_f \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}_i, \tilde{\mathbf{c}}_h)$; $\tilde{\mathbf{c}}'_f \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}'_i, \tilde{\mathbf{c}}'_h)$;$\tilde{\mathbf{c}}_d \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}_{id,i}, \tilde{\mathbf{c}}_h)$; $\tilde{\mathbf{c}}'_d \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}'_{id,i}, \tilde{\mathbf{c}}'_h)$, and calculate $\tilde{\mathbf{c}}_i^* \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_f, \tilde{\mathbf{c}}'_f)$; $\tilde{\mathbf{c}}_{id,i}^* \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_d, \tilde{\mathbf{c}}'_d)$. Finally, use `IUpack` to unpack $\tilde{\mathbf{c}}_i^*$ and $\tilde{\mathbf{c}}_{id,i}^*$ to obtain $\tilde{\mathbf{c}}(\mathbf{a}'_{2\alpha i+2(\alpha-1)}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}'_{2\alpha i})$ and $(\tilde{\mathbf{c}}(\mathbf{I}'_{2\alpha i+2(\alpha-1)}), \ldots, \tilde{\mathbf{c}}(\mathbf{I}'_{2\alpha i})$, respectively. Add $(\tilde{\mathbf{c}}(\mathbf{a}'_j), \tilde{\mathbf{c}}(\mathbf{I}'_j))$ $(j = 2\alpha i, \ldots, 2\alpha i + 2(\alpha-1))$ to set S'. 2) clear set S and let $S \leftarrow S'$.

- If $\|S\| \bmod 2 \neq 0$ and $\|S\| > 1$, pop the last tuple $(\tilde{\mathbf{c}}(\mathbf{a}_{\|S\|-1}), \tilde{\mathbf{c}}(\mathbf{I}_{\|S\|-1}))$ from set S so that $\|S\|/2 = 0$. Execute the above procedure ($\|S\|/2 = 0$ and $\|S\| > 1$) to generate $(\tilde{\mathbf{c}}(\mathbf{a}'_j), \tilde{\mathbf{c}}(\mathbf{I}'_j))$ $(j = 0, \ldots, \|S\|-1)/2-1)$, clear set S, and put $(\tilde{\mathbf{c}}(\mathbf{a}_{\|S\|-1}), \tilde{\mathbf{c}}(\mathbf{I}_{\|S\|-1}))$ and $(\tilde{\mathbf{c}}(\mathbf{a}'_j), \tilde{\mathbf{c}}(\mathbf{I}'_j))$ $(j = 0, \ldots, \|S\|-1)/2-1)$ into a set S.

We denote the algorithm $T \leftarrow \texttt{GMin}(T_0, \ldots, T_k : \mu')$.

5 Secure Signed Integer Computation Circuit

In this section, we will explain how to securely store the signed integer and achieve basic secure signed integer circuits over ciphertexts.

5.1 Two's Complement Representation

A two's-complement number system encodes positive and negative numbers in a binary number representation. The weight of each bit is a power of two, except for the most significant bit, whose weight is the negative of the corresponding power of two. The (integer) value $\mathbf{a}_{\texttt{ten}}$ of an μ-bit integer $\mathbf{a} = (\mathfrak{a}_{\mu-1}, \mathfrak{a}_{\mu-2}, \ldots, \mathfrak{a}_0)$ is given by the following formula: $\mathbf{a}_{\texttt{ten}} = -\mathfrak{a}_{\mu-1}2^{\mu-1} + \sum_{i=0}^{\mu-2} \mathfrak{a}_i 2^i$. Using the two's complement number system, all integers from $-2^{\mu-1}$ to $2^{\mu-1}-1$ may be represented. Given $(\mathfrak{a}_{\mu-1}, \mathfrak{a}_{\mu-2}, \ldots, \mathfrak{a}_0)$, we calculate $-a_{\texttt{ten}}$ by first executing $(1 \oplus \mathfrak{a}_{\mu-1}, 1 \oplus \mathfrak{a}_{\mu-2}, \ldots, 1 \oplus \mathfrak{a}_0)$, then adding integer $(0, \cdots 0, 1)$ to it. Next, we will show how to securely achieve the conversion (see supplemental material D, available online).

5.1.1 Secure Two's Complement Conversion (STC)

The protocol converts the plaintext of unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{a})$ into its two's complement form, stored in $\tilde{\mathbf{c}}(\mathbf{n})$. The construction is as follows: $\tilde{\mathbf{c}}(\mathbf{a}') \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{a}); \tilde{\mathbf{c}}(\pi^\star{}_{I_{\mu-1}}))$;

$$\tilde{\mathbf{c}}_1 \leftarrow \texttt{I.add}(\tilde{\mathbf{c}}(\mathbf{a}'), \tilde{\mathbf{c}}(\pi^\star{}_0));\ \tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}_1, \pi^\star{}_{I_{\mu-1}}).$$

Before designing the signed integer circuit, we need another protocol called obvious two's complement conversion, designed to convert an integer into its two's complement according to the symbol value.

5.1.2 Obvious Two's Complement Conversion (OTC)

Input unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{s})$, and output $\tilde{\mathbf{c}}(\mathbf{n})$, where $\mathfrak{s}_{\ell-1} = \cdots = \mathfrak{s}_\mu = \mathfrak{s}_{\mu-2} = \cdots = \mathfrak{s}_0 = 0$. The protocol can converts $\mathbf{a}$ to its two's complement $\mathbf{n}$ if $\mathfrak{s}_{\mu-1} = 1$; otherwise, the input remain unchanged.

1) Use STC to compute $\tilde{\mathbf{c}}(\mathbf{a}') \leftarrow \texttt{STC}(\tilde{\mathbf{c}}(\mathbf{a}))$.
2) The final output is securely selected according to $\mathfrak{s}_{\mu-1}$, i.e., select $\mathbf{a}'$ if $\mathfrak{s}_{\mu-1} = 1$ and $\mathbf{a}$ if $\mathfrak{s}_{\mu-1} = 0$. The algorithm calculates: $\tilde{\mathbf{c}}(\mathbf{s}') \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{s}), \tilde{\mathbf{c}}(\pi^\star{}_{\mu-1}))$; $\tilde{\mathbf{c}}(\mathbf{s}_1) \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{s}), \mu^*)$; $\tilde{\mathbf{c}}(\mathbf{s}_2) \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{s}'), \mu^*)$; $\tilde{\mathbf{c}}_1 \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{s}_2), \tilde{\mathbf{c}}(\mathbf{a}))$; $\tilde{\mathbf{c}}_2 \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{s}_1), \tilde{\mathbf{c}}(\mathbf{a}'))$; $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$, where $\mu^* = -(\mu-1)$. Note that, if $\mathfrak{s}_{\mu-1} = \mathfrak{a}_{\mu-1}$, the plaintext of $\tilde{\mathbf{c}}(\mathbf{n})$ in the OTC protocol is the absolute value of $\mathbf{a}$, i.e., $\mathbf{n}_{\texttt{ten}} = |\mathbf{a}_{\texttt{ten}}|$. Next, we introduce secure signed integer circuits.

5.2 Secure Signed Integer Computation Operations

Here, we introduce basic signed integer SIMD operations for addition, subtraction, comparison, multiplication and division.

5.2.1 Secure Signed Integer SIMD Addition Circuit– I.Sadd($\tilde{\mathbf{c}}(\mathbf{a}); \tilde{\mathbf{c}}(\mathbf{b})$)

Given two unpacked ciphertexts which store signed integers $\mathbf{a}$ and $\mathbf{b}$, `I.Sadd` outputs two ciphertexts, namely, $\tilde{\mathbf{c}}(\mathbf{n})$ and $\tilde{\mathbf{c}}(\mathbf{f})$ which store the addition result and error information, respectively. The construction directly uses `I.add`, only takes μ bits, and discards the carry-out.

Step-1. As we use the two's-complement number system, `I.add` is used to add the two numbers and keep the μ bit, i.e., $\tilde{\mathbf{c}}_1 \leftarrow \texttt{I.add}(\tilde{\mathbf{c}}(\mathbf{a}), \tilde{\mathbf{c}}(\mathbf{b}))$ and $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}_1, \pi^\star{}_{I_{\mu-1}})$.

Step-2. Any of the following two situations signifies an error: 1) two positive numbers produce a negative addition result ($\mathfrak{a}_{\mu-1} = 0$, $\mathfrak{b}_{\mu-1} = 0$, $\mathfrak{n}_{\mu-1} = 1$), 2) two negative numbers produce a positive addition result ($\mathfrak{a}_{\mu-1} = 1$, $\mathfrak{b}_{\mu-1} = 1$, $\mathfrak{n}_{\mu-1} = 0$). We use slot 0 of $\tilde{\mathbf{c}}(\mathbf{f})$ to store the overflow information, i.e., $\mathfrak{f}_0 = (1 \oplus \mathfrak{a}_{\mu-1} \oplus \mathfrak{b}_{\mu-1}) \wedge (\mathfrak{b}_{\mu-1} \oplus \mathfrak{n}_{\mu-1})$, such that overflow occurs when $\mathfrak{f}_0 = 1$, and $\mathfrak{f}_0 = 0$ otherwise. Step-2 proceeds as follows: $\tilde{\mathbf{c}}'_1 \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{a}), \tilde{\mathbf{c}}(\mathbf{b}))$; $\tilde{\mathbf{c}}'_2 \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}'_1, \tilde{\mathbf{c}}(\pi^\star{}_{\mu-1}))$; $\tilde{\mathbf{c}}'_3 \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{b}), \tilde{\mathbf{c}}(\mathbf{n}))$; $\tilde{\mathbf{c}}_a \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}'_2, \tilde{\mathbf{c}}'_3))$; $\tilde{\mathbf{c}}_b \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}_a, \pi^\star{}_{\mu-1})$; $\tilde{\mathbf{c}}(\mathbf{f}) \leftarrow \texttt{H.rotate}(\tilde{\mathbf{c}}_b, \ell-(\mu-1))$.

5.2.2 Secure Signed Integer SIMD Subtraction Circuit

Given two unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{b})$, output ciphertext $\tilde{\mathbf{c}}(\mathbf{n})$. Using two's complement, we can convert any subtraction operation into an addition, i.e., $\mathbf{a}_{\texttt{ten}} - \mathbf{b}_{\texttt{ten}} = \mathbf{a}_{\texttt{ten}} + (-\mathbf{b}_{\texttt{ten}})$. The secure signed integer subtraction circuit (`I.Ssub`) involves calculating $\tilde{\mathbf{c}}' \leftarrow \texttt{STC}(\tilde{\mathbf{c}}(\mathbf{b}))$ and $(\tilde{\mathbf{c}}(\mathbf{n}); \tilde{\mathbf{c}}(\mathbf{f})) \leftarrow \texttt{I.Sadd}(\tilde{\mathbf{c}}(\mathbf{a}); \tilde{\mathbf{c}}(\mathbf{b}))$.

5.2.3 Secure Signed Integer SIMD Comparison Circuit

Given two unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{b})$, output ciphertext $\tilde{\mathbf{c}}(\mathbf{n})$. If the sign bits are different, we choose the number with a positive sign bit as the larger one. Otherwise, we directly use `I.cmp` to compare the two integers. `I.Scmp` contains the following steps:

Step-1: Construct ciphertext $\tilde{\mathbf{c}}_a^*$ and $\tilde{\mathbf{c}}_b^*$, in which slot 0 stores $\mathfrak{a}_{\mu-1}$ and $\mathfrak{b}_{\mu-1}$, respectively. Moreover, we use $\tilde{\mathbf{c}}(\mathbf{d})$ to store the comparison result in slot 0. i.e., $\tilde{\mathbf{c}}_a \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{a}); \pi^\star{}_{\mu-1})$; $\tilde{\mathbf{c}}_b \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{b}); \pi^\star{}_{\mu-1})$; $\tilde{\mathbf{c}}_a^* \leftarrow \texttt{H.rotate}(\tilde{\mathbf{c}}_a; \ell-(\mu-1))$; $\tilde{\mathbf{c}}_b^* \leftarrow \texttt{H.rotate}(\tilde{\mathbf{c}}_b; \ell-(\mu-1))$; $\tilde{\mathbf{c}}(\mathbf{d}) \leftarrow \texttt{I.cmp}(\tilde{\mathbf{c}}(\mathbf{a}); \tilde{\mathbf{c}}(\mathbf{b}))$.

Step-2: Compute $(\mathfrak{a}_{\mu-1} \wedge (\mathfrak{a}_{\mu-1} \oplus \mathfrak{b}_{\mu-1})) \oplus [(1 \oplus \mathfrak{a}_{\mu-1} \oplus \mathfrak{b}_{\mu-1}) \wedge \mathfrak{d}_0]$ and store it in slot 0 of final result $\tilde{\mathbf{c}}(\mathbf{n})$, i.e., $\tilde{\mathbf{c}}_x \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_a^*, \tilde{\mathbf{c}}_b^*)$; $\tilde{\mathbf{c}}_y \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_x, \tilde{\mathbf{c}}(\pi^\star{}_0))$; $\tilde{\mathbf{c}}_1 \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}_a^*, \tilde{\mathbf{c}}_x)$; $\tilde{\mathbf{c}}_2 \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{d}), \tilde{\mathbf{c}}_y)$; $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2)$.

5.2.4 Secure Signed Integer SIMD Multiplication Circuit

Given two unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{b})$, output ciphertext $\tilde{\mathbf{c}}(\mathbf{n})$ in which slots 0 to $2\mu-1$ store the result.

Step-1. Same as Step-1 of `I.mul`.

Step-2. Invert the plaintext bit in slot $i+\mu-1$ of ciphertext $\tilde{\mathbf{c}}_i^*$ $(i=0,\ldots,\mu-2)$, i.e., for $i=0$ to $\mu-2$, calculate $\tilde{\mathbf{c}}_i^* \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}_i', \tilde{\mathbf{c}}(\pi^\star{}_{i+\mu-1}))$. For $\tilde{\mathbf{c}}_{\mu-1}'$, we need to invert the plaintext bits stored from slot $\mu-1$ to slot $2\mu-3$, i.e., calculate $\tilde{\mathbf{c}}_{\mu-1}^* \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}_{\mu-1}', \tilde{\mathbf{c}}(\pi^\star{}_x))$ where $\pi^\star{}_x$ stores 1 between slots $\mu-1$ and $2\mu-3$, and 0 in the other slots. Then, for $i=0$ to $\mu-1$, calculate $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{I.add}(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}_i^*)$. After performing `I.add` for μ times, compute $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{I.add}(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}(\pi^\star{}_y))$ where $\pi^\star{}_y$ stores 1 in slots $2\mu-1$ and μ, and 0 in the other slots. Finally, keep the plaintext from slots 0 to $2\mu-1$ of $\tilde{\mathbf{c}}(\mathbf{n})$, i.e., $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{n}), \pi^\star{}_{I_{2\mu-1}})$.

Optimize `I.Smul`: Similar to the optimization method in Section 4.3, we use `GSum` method to optimize our `I.Smul`, i.e., for step-2-2, all $\tilde{\mathbf{c}}_0^*,\ldots,\tilde{\mathbf{c}}_{\mu-1}^*$ and $\tilde{\mathbf{c}}(\pi^\star{}_y)$ can be added together where $\pi^\star{}_y$ stores 1 in slots $2\mu-1$ and μ, and 0 in the other slots, i.e., $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{GSum}(\tilde{\mathbf{c}}_0^*,\ldots,\tilde{\mathbf{c}}_{\mu-1}^*, \tilde{\mathbf{c}}(\pi^\star{}_y) : 2\cdot\mu+1)$. Then, calculate $\tilde{\mathbf{c}}(\mathbf{n}) \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{n}), \pi^\star{}_{I_{2\mu-1}})$.

5.2.5 Secure Signed/Unsigned Integer SIMD Division Circuit

Given unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{a})$ and $\tilde{\mathbf{c}}(\mathbf{b})$, the `I.Sdiv` output unpacked ciphertexts $\tilde{\mathbf{c}}(\mathbf{q})$ and $\tilde{\mathbf{c}}(\mathbf{r})$ which store the quotient and remainder results.

Step-1. Construct ciphertexts $\tilde{\mathbf{c}}_a^*$ and $\tilde{\mathbf{c}}_b^*$ for the two plaintexts or their two's complement, depending on their sign bits, i.e., $\tilde{\mathbf{c}}_{sa} \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{a}),\ \pi^\star{}_{\mu-1})$; $\tilde{\mathbf{c}}_{sb} \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}(\mathbf{b}),\ \pi^\star{}_{\mu-1})$; $\tilde{\mathbf{c}}(\mathbf{a}^*) \leftarrow \mathtt{OTC}(\tilde{\mathbf{c}}(\mathbf{a}), \tilde{\mathbf{c}}_{sa})$; $\tilde{\mathbf{c}}(\mathbf{b}^*) \leftarrow \mathtt{OTC}(\tilde{\mathbf{c}}(\mathbf{b}), \tilde{\mathbf{c}}_{sb})$;

Step-2. Initialize $\tilde{\mathbf{c}}_{RQ} \leftarrow \tilde{\mathbf{c}}(\mathbf{a}^*)$, execute phase 1-3 below μ times.

1) Use block 0 of $\tilde{\mathbf{c}}_{RQ}$ to store the intermediate result of $\mathbf{q}$ and use block 1 to store $\mathbf{r}$, rotate the plaintext slots of $\tilde{\mathbf{c}}_{RQ}$ and unpack them into $\tilde{\mathbf{c}}(\mathbf{q}^*)$ and $\tilde{\mathbf{c}}(\mathbf{r}^*)$, i.e., $\tilde{\mathbf{c}}_1 \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}_{RQ}, 1)$; $\tilde{\mathbf{c}}(\mathbf{q}^*) \leftarrow \mathtt{H.cmul}\ (\tilde{\mathbf{c}}_1, \pi^\star{}_{I_{\mu-1}})$; $\tilde{\mathbf{c}}_2 \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}_1, \ell-\mu)$; $\tilde{\mathbf{c}}(\mathbf{r}^*) \leftarrow \mathtt{H.cmul}(\tilde{\mathbf{c}}_2, \pi^\star{}_{I_{\mu-1}})$.
2) Compare $\mathbf{r}^*$ and $\mathbf{b}^*$ under the encrypted domain. If $\mathbf{r}^*_{\mathtt{ten}} < \mathbf{b}^*_{\mathtt{ten}}$, then set slot 0 of $\mathbf{q}^*$ to 0 and compute $\mathbf{r}'_{\mathtt{ten}} = \mathbf{r}^*_{\mathtt{ten}}$; otherwise, set slot 0 of $\mathbf{q}^*$ to 1 and calculate $\mathbf{r}'_{\mathtt{ten}} = \mathbf{r}^*_{\mathtt{ten}} - \mathbf{b}^*_{\mathtt{ten}}$, i.e., $\tilde{\mathbf{c}}_q \leftarrow \mathtt{I.cmp}(\tilde{\mathbf{c}}(\mathbf{r}^*),\ \tilde{\mathbf{c}}(\mathbf{b}^*))$; $\tilde{\mathbf{c}}_w \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}_p, \tilde{\mathbf{c}}(\pi^\star{}_0))$; $\tilde{\mathbf{c}}(\mathbf{q}') \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}(\mathbf{q}^*), \tilde{\mathbf{c}}_w)$; $\tilde{\mathbf{c}}_3 \leftarrow \mathtt{Scpy}(\tilde{\mathbf{c}}_w, \mu-1)$; $\tilde{\mathbf{c}}_b \leftarrow \mathtt{H.mul}(\tilde{\mathbf{c}}(\mathbf{b}^*), \tilde{\mathbf{c}}_3)$; $\tilde{\mathbf{c}}(\mathbf{r}') \leftarrow \mathtt{I.Ssub}(\tilde{\mathbf{c}}(\mathbf{r}^*), \tilde{\mathbf{c}}_b)$.
3) If this is the μth round, send $\tilde{\mathbf{c}}(\mathbf{q}')$ and $\tilde{\mathbf{c}}(\mathbf{r}')$ to the next step; otherwise, pack $\tilde{\mathbf{c}}(\mathbf{q}')$ and $\tilde{\mathbf{c}}(\mathbf{r}')$ together and denote it as the new $\tilde{\mathbf{c}}_{RQ}$, i.e.,

$$\tilde{\mathbf{c}}_4 \leftarrow \mathtt{H.rotate}(\tilde{\mathbf{c}}(\mathbf{r}'), \mu) \text{ and } \tilde{\mathbf{c}}_{RQ} \leftarrow \mathtt{H.add}(\tilde{\mathbf{c}}_4, \tilde{\mathbf{c}}(\mathbf{q}')).$$

Step-3. Determine the sign of the remainder and quotient. The sign of the remainder is the same as that of the divisor $\mathbf{a}$, while the sign of the quotient is the XOR of the signs of divisor $\mathbf{a}$ and dividend $\mathbf{b}$. i.e., $\tilde{\mathbf{c}}(\mathbf{r}) \leftarrow \mathtt{OTC}(\tilde{\mathbf{c}}(\mathbf{r}'), \tilde{\mathbf{c}}_{sa})$ and $\tilde{\mathbf{c}}(\mathbf{q}) \leftarrow \mathtt{OTC}(\tilde{\mathbf{c}}(\mathbf{q}'), \mathtt{H.add}(\tilde{\mathbf{c}}_{sa}, \tilde{\mathbf{c}}_{sb}))$.

Moreover, if dividend $\mathbf{b}_{\mathtt{ten}} = 0$, use $\tilde{\mathbf{c}}(\mathbf{f})$ to store the exception information, $\tilde{\mathbf{c}}(\mathbf{f}) \leftarrow \mathtt{I.equ}(\tilde{\mathbf{c}}(\mathbf{b}, \tilde{\mathbf{c}}(0))$.

Construction of `I.div`: If both divisor $\mathbf{a}$ and dividend $\mathbf{b}$ are unsigned integers, then unsigned version of `I.Sdiv` is much simpler, which we denote it as `I.div`. The construction is as follows.

(1) Let $\tilde{\mathbf{c}}(\mathbf{a}^*) \leftarrow \tilde{\mathbf{c}}(\mathbf{a})$, and $\tilde{\mathbf{c}}(\mathbf{b}^*) \leftarrow \tilde{\mathbf{c}}(\mathbf{b})$. (2) Same as Step-2 in `I.Sdiv`. (3) Let $\tilde{\mathbf{c}}(\mathbf{q}) \leftarrow \tilde{\mathbf{c}}(\mathbf{q}')$, $\tilde{\mathbf{c}}(\mathbf{r}) \leftarrow \tilde{\mathbf{c}}(\mathbf{r}')$, and calculate $\tilde{\mathbf{c}}(\mathbf{f}) \leftarrow \mathtt{I.equ}(\tilde{\mathbf{c}}(\mathbf{b}), \tilde{\mathbf{c}}(0))$.

5.3 Extension for Packed Signed Integer Calculation

Adopting the two's complement number system, we directly use `Ipack` to pack the ciphertext. The input and output of all the secure signed circuits have the same μ. We pay special attention to the setting of μ and μ'. For example, -1 is stored as (111) with $\mu=3$. If we directly use `Ipack` to pack the ciphertext with a new size $\mu'=4$, the datum is changed to (0111). If we then use $\mu'=4$ to decrypt, then the result is 7 which is erroneous. If instead we use $\mu=3$ to decrypt, then the plaintext is still -1. Usually, μ' is larger than μ due to overflow and underflow problems (for a similar reason as that in the packed unsigned Integer calculation, see Section 4.3). Extending the secure signed integer computation circuit to its packed version is same as in Section 4.3, i.e., change $\pi^\star{}_j$ to $\eta^\star{}_{j,\mu'}$, $\bar{\pi}^\star{}_j$ to $\bar{\eta}^\star{}_{j,\mu'}$, $\pi^\star{}_{I_{\mu-1}}$ to $\eta^\star{}_{I_{\mu-1},\mu'}$, and the unpacked unsigned integer circuit to its packed version. We denote the packed version of `STC` and `OTC` as `P.STC` and `P.OTC`, respectively. Also, the packed version of secure signed integer SIMD addition, comparison, multiplication, and division are denoted as `PI.Sadd`, `PI.Scmp`, `PI.Smul`[2] and `PI.Sdiv`,[3] respectively (see supplemental material D, available online, for construction).

6 Application and Extension

In this section, we use the above secure computation circuits to construct an application called secure k-Nearest Neighbors (k-NN) classifier, and extend the secure integer circuits to support multiple keys and achieve fix-point number storage and calculation.

6.1 Secure k-Nearest Neighbors Classifier

The k-NN classifier [24] is one of the most important data mining methods, and its application ranges from language recognition [25] to computational geometry [26] to graphs [27], and so on. k-NN classifier classifies an object by the majority vote of its neighbors; in other words, an object is assigned to the class that is most common among its k nearest neighbors (where k is a positive, typically small integer). Formally, a data set contains β instances $\{(\vec{\mathbf{x}}_0, y_0, id_0),\ldots,(\vec{\mathbf{x}}_{\beta-1}, y_{\beta-1}, id_{\beta-1})\}$, where $\vec{\mathbf{x}}_i$ is an input sample with χ features $(x_{i,\chi-1},\ldots,x_{i,0})$, $y_i \in \{c_1, c_2\}$ is the class label of $\vec{\mathbf{x}}_i$, and id_i is the identity label of instance i. Given input data $\vec{\mathbf{x}}_p \leftarrow (x_{p,\chi-1},\ldots,x_{p,0})$, the goal of k-NN classifier is to determine the class label of $\vec{\mathbf{x}}_p$. The classifier proceeds as follows. I. Calculate the distance between each instance $\vec{\mathbf{x}}_i$ from $\vec{\mathbf{x}}_p$, denoting it by a_i. II. Find the k smallest distances and the corresponding instances, denoting them as $(\vec{\mathbf{x}}'_0, y'_0, id'_0),\ldots,(\vec{\mathbf{x}}'_{k-1}, y'_{k-1}, id'_{k-1})$. III. Set the class label of $\vec{\mathbf{x}}_p$ according to those k instances. If the majority of y'_i $(i=0$ to $k-1)$ belong to class c_1, let $y_p \leftarrow c_1$; otherwise, $y_p \leftarrow c_2$. For secure data storage, all data are stored in encrypted form, i.e., $x_{i,j}$ is stored as unpacked ciphertext $\tilde{\mathbf{c}}(\mathbf{x}_{i,j})$, id_i as $\tilde{\mathbf{c}}(\mathbf{I}_i)$ and y_i as $\tilde{\mathbf{c}}(\mathbf{y}_i)$. When $\tilde{\mathbf{c}}(\mathbf{x}_{p,\chi-1}),\ldots,\tilde{\mathbf{c}}(\mathbf{x}_{p,0})$ are given, the secure k-NN

2. In addition to changing $\pi^\star{}_i$ to $\eta^\star{}_{i,\mu'}$, `I.add` in step 2 should be changed to `PI.add`. Moreover, $\pi^\star{}_x$ should be changed to $\eta^\star{}_x$ that stores 1 from slots $\mu-1+\mu_-+k\mu'$ to $2\mu-3+\mu_-+k\mu'$, and $\pi^\star{}_y$ is changed to $\eta^\star{}_y$ so that stores 1 in slots $2\mu-1+\mu_-+k\mu'$ and $\mu+\mu_-+k\mu'$, 0 in the other slots, where $k=0$ to $\lfloor \ell/\mu' \rfloor$.

3. The new packed block size requires $\mu' \geq 3\mu$. Also, all the secure circuits in `PI.Sdiv` should be changed to their packed versions.

classifier derives the class label of $\vec{\mathbf{x}}_p$ in a privacy-preserving manner as follows:

Step-1. Calculate the Manhattan distance between $\mathbf{x}_i$ and $\mathbf{x}_p$, i.e., $a_i = \sum_{j=0}^{\chi-1} |x_{i,j} - x_{p,j}|$.

1) The goal is to calculate $h_{i,j} = |x_{i,j} - x_{p,j}|$. As the ciphertext may not fit within a single packed ciphertext, for $i = 0, \ldots \beta - 1; t_1 = 0, \ldots, \lfloor \chi/\alpha_1 \rfloor$, calculate

$$\tilde{\mathbf{c}}_{i,t_1} \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{x}_{i,\alpha_1 t_1+\alpha_1-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{x}_{i,\alpha_1 t_1}) : \mu_1);$$
$$\tilde{\mathbf{c}}'_{i,t_i} \leftarrow \texttt{PI.Ssub}(\tilde{\mathbf{c}}_{i,t_i}, \tilde{\mathbf{c}}_{i,p});$$
$$\tilde{\mathbf{c}}^*_{i,t_i} \leftarrow \texttt{H.cmul}(\tilde{\mathbf{c}}'_{i,t_i}, \eta^* \mu_{-1,\mu_1}); \ \tilde{\mathbf{c}}''_{i,t_i} \leftarrow \texttt{P.OTC}(\tilde{\mathbf{c}}'_{i,t_i}, \tilde{\mathbf{c}}^*_{i,t_i});$$
$$(\tilde{\mathbf{c}}(\mathbf{h}_{i,i,\alpha_1 t_1+\alpha_1-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{h}_{i,\alpha_1 t_1})) \leftarrow \texttt{IUpack}(\tilde{\mathbf{c}}''_{i,t_i}, \mu_1);$$

where $\mu_1 \geq \mu + 1$ and $\alpha_1 = \lfloor \ell/\mu_1 \rfloor$. After this step, CP obtains $\tilde{\mathbf{c}}(\mathbf{h}_{i,\chi-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{h}_{i,0})$, where $h_{i,j}$ resides in block 0 of $\tilde{\mathbf{c}}(\mathbf{h}_{i,j})$.

2) For each instance i, sum $h_{i,j}$ to get $a_i = \sum_{j=0}^{\chi-1} h_{i,j}$. We use `Ipack` to pack different instances i with the same specific feature j before performing the addition, i.e, for $t_2 = 0, \ldots, \lfloor \beta/\alpha_2 \rfloor$ and $j = 0, \ldots, \chi - 1$, pack and generate $\tilde{\mathbf{c}}(\mathbf{z}_{t_2,j})$ as

$$\tilde{\mathbf{c}}(\mathbf{z}_{t_2,j}) \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{h}_{\alpha_2 t_2+\alpha_2-1,j}), \ldots, \tilde{\mathbf{c}}(\mathbf{h}_{\alpha_2 t_2,j}) : \mu_2);$$

where $\mu_2 \geq \mu + \chi$ and $\alpha_2 = \lfloor \ell/\mu_2 \rfloor$. After packing, initialize $\tilde{\mathbf{c}}(\mathbf{v}_{t_2}) = \tilde{\mathbf{c}}(\mathbf{z}_{t_2,0})$. Then, for $j = 1$ to $\chi - 1$, calculate $\tilde{\mathbf{c}}(\mathbf{v}_{t_2}) \leftarrow \texttt{PI.add}(\tilde{\mathbf{c}}(\mathbf{v}_{t_2}), \tilde{\mathbf{c}}(\mathbf{z}_{t_2,j}); \mu_2)$.

3) Finally, unpack all ciphertexts for $t_2 = 0, \ldots, \lfloor \beta/\alpha_2 \rfloor$, execute

$$(\tilde{\mathbf{c}}(\mathbf{a}_{\alpha_2 t_2+\alpha_2-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_{\alpha_2 t_2})) \leftarrow \texttt{IUpack}(\tilde{\mathbf{c}}(\mathbf{v}_{t_2}), \mu_2).$$

At the end of Step-1, we obtain $\tilde{\mathbf{c}}(\mathbf{a}_{\beta-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0)$ which stores $a_{\beta-1}, \ldots, a_0$ in block 0 with block size μ_2 of each ciphertext, respectively.

Step-2. Find the smallest k Manhattan distances among $\mathbf{a}_{\beta-1}, \ldots, \mathbf{a}_0$, execute for k times ($t_3 = 0, \ldots, k-1$). (1) Use `GMin` to find the encrypted minimum Manhattan distance value and its corresponding identity $\tilde{\mathbf{c}}(\mathbf{n})$ and $\tilde{\mathbf{c}}(\mathbf{I})$ in $\tilde{\mathbf{c}}(\mathbf{a}_{\beta-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0)$; specifically, calculate $(\tilde{\mathbf{c}}(\mathbf{n}), \tilde{\mathbf{c}}(\mathbf{I})) \leftarrow \texttt{GMin}((\tilde{\mathbf{c}}(\mathbf{a}_{\beta-1}), \tilde{\mathbf{c}}(\mathbf{I}_{\beta-1})), \ldots, (\tilde{\mathbf{c}}(\mathbf{a}_0), \tilde{\mathbf{c}}(\mathbf{I}_0)) : \mu_3)$, where $\mu_3 \geq 2^{\lceil log_2 \mu_2 \rceil}$ and $\alpha_3 = \lfloor \ell/\mu_3 \rfloor$. (2) Use identity $\mathbf{I}$ to test all β identities to determine whether $\mathbf{I}$ is equal to $\mathbf{I}_i$ ($i = 0, \ldots, \beta - 1$). If $\mathbf{I} = \mathbf{I}_i$, then set $\mathbf{a}_i$ to the system maximum value. Otherwise, the value of $\mathbf{a}_i$ remains unchanged. For $i = 0, \ldots, \lfloor \beta/(2\alpha_3) \rfloor$, calculate $\tilde{\mathbf{c}}(\mathbf{a}') \leftarrow \texttt{Ipack}\ (\tilde{\mathbf{c}}(\mathbf{a}_{\alpha_3 i+\alpha_3-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_{\alpha_3 i}) : \mu_3); \tilde{\mathbf{c}}(\mathbf{w}) \leftarrow \texttt{Ipack}\ \ (\tilde{\mathbf{c}}(\mathbf{I}_{\alpha_3 i+\alpha_3-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{I}_{\alpha_3 i}) : \mu_3); \ \ \tilde{\mathbf{c}}(\mathbf{w}') \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{I}), \ldots, \tilde{\mathbf{c}}(\mathbf{I}) : \mu_3)$.

Then, `PI.equ` is used to compare the relationship between $\mathbf{w}$ and $\mathbf{w}'$. If the value stored in block j of $\mathbf{w}$ equals to the value in the corresponding block of $\mathbf{w}'$, set each slot of this block in $\tilde{\mathbf{c}}(\mathbf{a}')$ to 1. Otherwise, the value stored in this block remains unchanged.

$$\tilde{\mathbf{c}}(\mathbf{p}) \leftarrow \texttt{PI.equ}(\tilde{\mathbf{c}}(\mathbf{w}), \tilde{\mathbf{c}}(\mathbf{w}'));$$
$$\tilde{\mathbf{c}}(\mathbf{p}') \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{p}), \tilde{\mathbf{c}}(\eta^*{}_{0,\mu_3})); \tilde{\mathbf{c}}(\mathbf{p}_1) \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{p}), \mu_3 - 1);$$
$$\tilde{\mathbf{c}}(\mathbf{p}_2) \ \leftarrow \ \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{p}'), \mu_3 - 1); \tilde{\mathbf{c}}_1 \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{p}_2), \tilde{\mathbf{c}}(\mathbf{a}'));$$
$$\tilde{\mathbf{c}}_2 \ \leftarrow \ \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{p}_1), \tilde{\mathbf{c}}_{one});^4 \tilde{\mathbf{c}}(\mathbf{n}') \ \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}_1, \tilde{\mathbf{c}}_2);$$

4. $\tilde{\mathbf{c}}_{one}$; is the ciphertext which stores 1 in all the slots.

$$(\tilde{\mathbf{c}}(\mathbf{a}'_{\alpha_3 i+\alpha_3-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}'_{\alpha_3 i})) \leftarrow \texttt{IUpack}(\tilde{\mathbf{c}}(\mathbf{n}'), \mu_3).$$

Moreover, judge the class label of $\mathbf{I}_i$, i.e., letting $\mathbf{b}'_i = 1$, if $\mathbf{I} = \mathbf{I}_i$ then $(\mathbf{y}_i)_{\texttt{ten}} = c_1$; otherwise, let $(\mathbf{y}_i)_{\texttt{ten}} = 0$

$$\tilde{\mathbf{c}}_{py} \leftarrow \texttt{Ipack}(\tilde{\mathbf{c}}(\mathbf{y}_{\alpha_3 i+\alpha_3-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{y}_{\alpha i}) : \mu_3);$$
$$\tilde{\mathbf{c}}_3 \leftarrow \texttt{PI.equ}(\tilde{\mathbf{c}}_{py}, \tilde{\mathbf{c}}(\mathbf{y}^*)); \tilde{\mathbf{c}}''_i \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{p}), \tilde{\mathbf{c}}_3);$$
$$(\tilde{\mathbf{c}}(\mathbf{b}'_{\alpha_3 k+\alpha_3-1,j}), \ldots, \tilde{\mathbf{c}}(\mathbf{b}'_{\alpha_3 k,j})) \leftarrow \texttt{IUpack}(\tilde{\mathbf{c}}''_i, \mu_3),$$

where $\tilde{\mathbf{c}}(\mathbf{y}^*)$ stores integer c_1 in each block. After the calculation, the algorithm obtains $\tilde{\mathbf{c}}(\mathbf{a}'_\beta), \ldots, \tilde{\mathbf{c}}(\mathbf{a}'_0)$ and $\tilde{\mathbf{c}}(\mathbf{b}'_\beta), \ldots, \tilde{\mathbf{c}}(\mathbf{b}'_0)$.

(3) Refresh the value of $\mathbf{a}_{\beta-1}, \ldots, \mathbf{a}_0$ by replacing $\tilde{\mathbf{c}}(\mathbf{a}_{\beta-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}_0)$ with $\tilde{\mathbf{c}}(\mathbf{a}'_{\beta-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{a}'_0)$. Also, add $\mathbf{b}'_\beta, \ldots, \mathbf{b}'_0$ to a single element $\mathbf{b}'_{t_3}$ because only the element which $\mathbf{I} = \mathbf{I}_i$ stores information and all the other elements store 0. To perform the calculation, initialize $\tilde{\mathbf{c}}(\mathbf{b}_{t_3}) \leftarrow \tilde{\mathbf{c}}(\mathbf{b}'_0)$. Then, for $j = 1, \ldots, \beta - 1$, calculate $\tilde{\mathbf{c}}(\mathbf{b}_{t_3}) \leftarrow \texttt{I.add}(\tilde{\mathbf{c}}(\mathbf{b}_{t_3}, \mathbf{b}'_j)$.

Step-3. After obtaining $\tilde{\mathbf{c}}(\mathbf{b}_{k-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{b}_0)$, determine whether $\sum_{i=0}^{k-1} (\mathbf{b}_i)_{\texttt{ten}} \geq \lceil k/2 \rceil$. If $\sum_{i=0}^{k-1} (\mathbf{b}_i)_{\texttt{ten}} \geq \lceil k/2 \rceil$, set the class label of $\mathbf{x}_p$ to c_1; otherwise, the class label is c_2. (1) Add $\mathbf{b}_0, \ldots, \mathbf{b}_{k-1}$ with `GSum` and achieve secure comparison with `I.cmp`, i.e., $\tilde{\mathbf{c}}_l \leftarrow \texttt{GSum}(\tilde{\mathbf{c}}(\mathbf{b}_{k-1}), \ldots, \tilde{\mathbf{c}}(\mathbf{b}_0) : \mu_4); \tilde{\mathbf{c}}(\mathbf{s}) \leftarrow \texttt{I.cmp}(\tilde{\mathbf{c}}_l, \tilde{\mathbf{c}}(\mathbf{B})); \tilde{\mathbf{c}}(\mathbf{s}') \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{s}), \tilde{\mathbf{c}}(\pi^*_0))$.

(2) We need to pad the comparison result $\mathbf{s}$ and $\mathbf{s}'$ from slot 0 to all the slot in block 0, and use `H.add` and `H.mul` to select c_1 and c_2 as follows: $\tilde{\mathbf{c}}(\mathbf{o}_1) \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{s}'), \mu - 1); \tilde{\mathbf{c}}(\mathbf{o}_2) \leftarrow \texttt{Scpy}(\tilde{\mathbf{c}}(\mathbf{s}), \mu - 1); \tilde{\mathbf{c}}(\mathbf{y}_1) \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{o}_1), \tilde{\mathbf{c}}(\mathbf{k}_1)); \tilde{\mathbf{c}}(\mathbf{y}_2) \leftarrow \texttt{H.mul}(\tilde{\mathbf{c}}(\mathbf{o}_2), \tilde{\mathbf{c}}(\mathbf{k}_2))$; and $\tilde{\mathbf{c}}(\mathbf{y}_p) \leftarrow \texttt{H.add}(\tilde{\mathbf{c}}(\mathbf{y}_1), \tilde{\mathbf{c}}(\mathbf{y}_2))$, where $\tilde{\mathbf{c}}(\mathbf{k}_1)$ stores c_1 in block 0 and $\tilde{\mathbf{c}}(\mathbf{k}_2)$ stores c_2, $\tilde{\mathbf{c}}(\mathbf{B})$ stores integer $\lceil k/2 \rceil$ in block 0, and $\mu_4 \geq \mu + k$.

6.2 Secure Computation with Multiple Keys

All the secure circuits above can only be computed under the same key. If calculation across different domains is needed, Pockit cannot be applied directly. A straightforward solution is to use a multiple-key fully homomorphic encryption (MK-FHE) scheme to construct the circuits. However, existing MK-FHE schemes under the standard assumption [28] is still inefficient in terms of storage requirement and computational overhead. Another solution is to use `KeySwitch`—the transformation key is used to map encrypted domain of one user to that of another user, as in $\tilde{\mathbf{c}}^{\langle pk_j \rangle} \leftarrow \texttt{KeySwitch}(\tilde{\mathbf{c}}^{\langle pk_i \rangle}, W_{sk_i \rightarrow sk_j})$. That transforms user i's ciphertext $\tilde{\mathbf{c}}^{\langle pk_i \rangle}$ to user j's $\tilde{\mathbf{c}}^{\langle pk_j \rangle}$. In general, all the data can be transformed to the same public key domain, before applying Pockit for the computation. As the transformation key is the public key, `KeySwitch` can be stored and performed in CP without compromising the privacy of the DUs (see the efficiency of `KeySwitch` in Section 7.3).

6.3 Extension for Fix-Point Number Storage and Calculation

The above secure integer storage and calculation procedures can be easily extended to fix-point numbers. Indeed, integers can be considered a special case of fix-point number with the decimal point behind the least significant bit. (see supplemental material Section E, available online, for the construction).

7 Security & Performance Analysis

In this section, we prove the security of the circuits in Pockit based on the security model defined in Section 3.2,[5] analyze Pockit, and demonstrate its efficiency.

7.1 Security Analysis

Theorem 1. *The BGV scheme used in our paper is secure, assuming the semantic security of the underlying ring learning with errors (RLWE) hard problem.*

Proof. The security and hardness of the RLWE problem and security of the BGV scheme can be found in [19]. □

Theorem 2. *The secure unsigned/signed integer SIMD calculation circuits constructed in our paper securely computes integer calculation of plaintext over ciphertext in the presence of semi-honest adversaries $\mathcal{A}$ in Section 3.2.*

Proof. All calculations are performed on ciphertexts. Without the private key of the challenger DU, it is impossible for an adversary to decrypt the ciphertexts to obtain the plaintexts due to the semantic security of the BGV scheme (see Theorem 1). □

7.2 Analysis of Pockit

Why FHE scheme? Partial homomorphic encryption (PHE) can be used to achieve efficient secure integer computations on encrypted data, however, multiple (at least two) non-colluding servers are still required with executing some secure interactive protocols, and it hampers the large-scale application in the cloud. All the FHE based secure integer computation circuits can be performed on a single cloud server which reduces the risk of data leakage.

Why RLWE-based BGV scheme? The RLWE-based BGV is appropriate for designing our Pockit in two aspects: 1) Storage consideration. The RLWE-based BGV can store $\phi(m)$ bits per ciphertext,[6] while the learning with errors (LWE) based BGV scheme can only store one bit. 2) SIMD calculation. The operations in RLWE ciphertext can be carried out on all the slots concurrently, which accelerates the computation by up to $\ell = \phi(m)/d$ times.

Why Bootstrapping? The RLWE-based BGV used in our system is a levelled homomorphic encryption scheme (which reduces noise by moving ciphertext from an upper level to a lower level). In order to support large circuits without bootstrapping, a very large L is required which makes the system inefficient and impractical. In other words, we would need a larger m and large ciphertext modular q_{L-1}, which results in significant ciphertext storage overhead because of Double-CRT representation (see the definition in [31]). The modulus-key-switching and bootstrapping techniques make Pockit more practical as we no longer require significant storage to store the ciphertexts, or significant RAM to load the ciphertexts during the computation.

5. The privacy model described in Section 6.2 can be employed to protect the content of the data (including the input data and final output). However, this model is unable to capture information leakage due to data access pattern. The latter can be solved using oblivious RAM, which is beyond the scope of this paper. We refer interested reader to [29], [30] for the construction.

6. To support integer calculation, we only use ℓ bits for storage.

Why use message space $\mathbb{Z}_2$? As binary circuits are time-consuming, Naehrig et al. [32] and Cheon et al. [16] provided a method to reduce the integer addition and multiplication computation overheads, using message space $\mathbb{Z}_t$. The idea is straightforward: break each message **a** into (at most μ) bits $\mathbf{a} = (\mathfrak{a}_{\mu-1}, \ldots, \mathfrak{a}_0) \in \mathbb{Z}_2^{\mu-1}$, code the message as $a^\star(X) = \sum_j \mathfrak{a}_j X^j \in \mathbb{A}_t$, and sum k of them to obtain $a^\star{}_{add}(X) = \sum_{i=1}^{k} a^\star{}_i(X)$. As long as $t > k$, this does not wrap around modulo t and upon decryption, and the final result is obtained by calculating $a^\star{}_{add}(2)$. For multiplication, if we encode **a** as polynomials of degree at most $\phi(m)/d'$, then the final result $a^\star{}_{mul}(X) = \prod_{i=1}^{d'} a^\star{}_i(X)$ do not wrap around modulo $\Phi_m(X)$ if it undergoes less than d' multiplication operations, and the final result is $a^\star{}_{mul}(2)$. However, we still adopt message space $\mathbb{Z}_2$ due to the following reasons:

- m and t cannot be too large. Large m and t values greatly affect the storage cost and computation overhead, especially for bootstrapping (see [33] for a detailed comparison).
- Inability to perform mixed operations. The $\mathbb{Z}_t$ method allows either integer additions or integer multiplications; it cannot support mixed operations (e.g., it does not permit integer addition after integer multiplications, since wraping around modulo will result in errors in the final result).
- Difficulty in designing integer arithmetic circuits. Designing integer arithmetic circuits are much more complicated than binary circuits.

7.3 Experiment Analysis

We evaluated the performance of our proposed secure circuits on a virtual machine (3.6 GHz single-core processor and 4 GB DDR3-1600 RAM memory). Due to the use of HElib [31], all programs are single-threaded safe with built in C++. To test the efficiency of our Pockit, Three types of metrics are considered, *runtime*, *communication cost*, and *security level*. The runtime refers to the outsourced secure circuits executing duration in our testbed (with the unit in second (s) or millisecond (ms)). As the data are outsourced before computation, *communication cost* (with the unit in MegaByte (MB)) refers to the total size of data (including encrypted data outsourced and retrieved) transmitted between DU and CP during one-time executing secure computation for secure circuits. Moreover, the security level is an indication of the security strength of a cryptographic primitive.

7.3.1 The Helib Parameter Initial

To select an appropriate $\Phi(m)$, the security level λ should be used, and in our context, $\Phi(m) \geq \frac{\log_2(Q/\sigma)(\lambda+110)}{7.2}$ with RLWE instance with modulus Q and noise σ [20]. Next, the modular chain are generated with p_i which p_0 is half the bit-size of the other p_i's. Thus, the odd indexed moduli in the chain are a product of the primes starting at p_0 ($q_i = \prod_{j=0}^{\lfloor i/2 \rfloor} p_j$) and the even indexed moduli are products that do not include p_0 ($q_i = \prod_{j=1}^{i/2} p_j$). For example, the length of q_0 has 23-25 bits, and the length of full-sized primes have 46-50 bits when $\lambda = 76$. The element in $\mathbb{A}_{q_t}$ uses (Double-CRT) representation, and the length of one ciphertext will not exceed $(\|q_0\| + \cdots + \|q_t\|) \times \phi(m)$ ($t \times \phi(m)$ matrix), where $\|q_0\|$

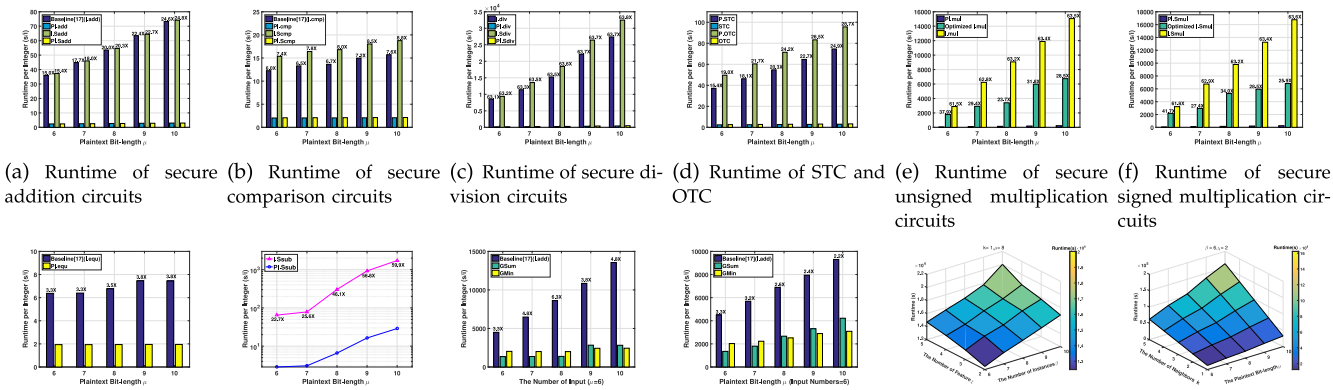
TABLE 2
Performance of the Basic Operations

Security	$\lvert\Phi(m)\rvert$	Slots	d	Enc	Dec	KS	MS	H.mul	H.cmul	H.rotate	Bootstrap	CL (B/A)
76	16,384	1,024	16	0.135	0.051	0.199	0.129	0.381	0.012	0.426/0.517	196.36	22/10
123	23,040	960	24	0.147	0.064	0.235	0.153	0.448	0.013	0.625/0.957	247.81	24/11
145	46,080	1,920	24	0.304	0.128	0.480	0.315	0.918	0.028	1.29/1.95	672.12	40/25

– In the table, the unit of runtime is seconds, KS represents KeySwitch, MS represents ModSwitch, H.rotate has two values ($k \bmod d = 0 / k \bmod d \neq 0$), where k is the rotated number. CL (B/A) represents the ciphertext level (before/after bootstrapping)

TABLE 3
Performance of the Secure Unsigned Integer Circuits ($\mu = 6$, $\mu' = 16$)

Circuits	Runtime for Single Key (s/i)			Runtime for Multiple Keys (s/i)		
	Unpacked	Packed	Improvement	Unpacked	Packed	Improvement
I.add	36.02	2.40	15X	37.00	2.41	15.4X
I.cmp	12.26	2.03	6.0X	13.24	2.04	6.5X
I.equ	6.37	1.94	3.3X	7.35	1.95	3.8X
I.mul	2883	46.88	61.5X	2,883.98	46.90	61.5X
OptimizedI.mul	1,775.37	29.58	60X	1,776.35	29.6	60X
I.div	8,563.01	135.63	63.1X	8564	135.65	63.1X



(a) Runtime of secure addition circuits (b) Runtime of secure comparison circuits (c) Runtime of secure division circuits (d) Runtime of STC and OTC (e) Runtime of secure unsigned multiplication circuits (f) Runtime of secure signed multiplication circuits

(g) Runtime of secure equality circuits (h) Runtime of secure subtraction circuits (i) Runtime of I. add, GSum and GMin (vary with input numbers) (j) Runtime of I. add, GSum and GMin (vary with μ) (k) Runtime of secure k-NN classifier (vary with β and χ) (l) Runtime of secure k-NN classifier (vary with μ and k)

Fig. 4. Evaluations.

denotes the bit-length of q_0. Thus, the level-t ciphertext needs two elements in $\mathbb{A}_{q_t}$, which requires $2 \times (\|q_0\| + \cdots + \|q_t\|) \times \phi(m)$.

7.3.2 Performance of Basic Operations

With the above parameters, we choose three security levels ($\lambda = 76, 123, 145$) to test the basic operations. For integer and ring element transformation (for $\lambda = 76/123/145$), it takes 0.857/1.19/2.49 ms to encode an 16-bit unsigned integer into an $\mathbb{A}_2$ element, while it takes 89.12/118.41/480.61 ms to encode a 16-bit signed integer into an $\mathbb{A}_2$ element. Also, it takes 1.15/1.45/2.76 ms to decode an $\mathbb{A}_2$ element into a 16-bit unsigned integer, and 89.61/119.36/481.92 ms to decode an $\mathbb{A}_2$ element into a 16-bit signed integer. Moreover, it cost 0.1/0.12/0.39 ms for H.add, 22.704/50.383/218.959 s for Ipack, and 28.467/58.893/255.836 s for IUpack (with block size is 32-bit). For other parameters and homomorphic operations, we list the performance of basic operations in Table 1. Note that multiple generators are used to construct the hypercube structure to perform the H.rotate in Helib.

7.3.3 Performance of Secure Unsigned Integer Circuit

Here, we compared our circuit with the existing circuits (I.add, I.cmp, and I.equ) in [16], and use *second per integer* (short for s/i) to measure performance. For illustration, we choose security level $\lambda = 76$ and let plaintext size $\mu = 8$ and Ipack parameter $\mu' = 16$. Note that the running time of the packed unsigned integer circuit includes the running time of Ipack and IUpack. For evaluate the efficiency of the basic unsigned computation circuits and their packed version, we use our testbed to test the runtime and communication cost of the circuits for both single key setting and multiple keys settings in Table 3. Moreover, we evaluate two factors (bit-length and the number of input) which affects the performance of secure circuits. Also, the bit-length affects the performance of the circuits. From Figs. 4a, 4b, and 4c, we observe that the runtime of the circuits with single key and multiple keys setting increase with the plaintext bit-length, and packed secure unsigned integer circuits are much more efficient than their unpacked counterparts. The runtime of our packed circuit only needs $t_{pv} = 1/\alpha' t_{up} + (\alpha' - 1)(2t_{rt} + t_{ad}) + \alpha' t_{cml}$ for single key

TABLE 4
Performance of Improved Secure Circuits ($\mu = 6$, $\mu' = 16$)

Runtime of Single Key (s/i)			Runtime of Multiple Keys (s/i)		
`I.add`	`GSum`	Improve	`I.add`	`GSum`	Improve
4,499.5	1,351.2	3.3X	4,500.5	1,352.2	3.3X

TABLE 5
Performance of the Secure Signed Integer Circuits ($\mu = 6$, $\mu' = 16$)

Circuits	Runtime of Single Key (s/i)			Runtime of Multiple Keys (s/i)		
	Unpacked	Packed	Improvement	Unpacked	Packed	Improvement
`STC`	37.15	2.41	15.4X	38.13	2.43	15.7X
`OTC`	49.67	2.61	19.0X	50.65	2.63	19.3X
`I.Sadd`	37.3	2.42	15.4X	38.28	2.43	15.8X
`I.Scmp`	15.37	2.08	7.4X	16.35	2.09	7.8X
`I.Smul`	3,227.72	52.27	61.8X	3,328.7	53.85	61.8X
Optimized`I.Smul`	2,179.09	35.88	60.7X	2,180.1	35.9	60.7X
`I.Sdiv`	9,396.91	148.66	63.2X	9,397.9	148.68	63.2X

TABLE 6
Comparative Summary

Function/Algorithm	[18]	[13]	[15]	[14]	[32]	[16]	Proposed
No additional server	×	×	×	×	✓	✓	✓
Multiple Keys	✓	×	×	✓	×	×	✓
Communication round (between servers)	Multiple	Multiple	Multiple	Multiple	0	0	0
Communication round (user & server)	Multiple	One	One	One	One	One	One
Solve Overflow Problem	×	×	✓	×	×	×	✓
Numbers of integer Calculation	Arbitrary	Arbitrary	Arbitrary	Arbitrary	Limit	Limit	Arbitrary
Crypto Homomorphic Type	Additive	Additive	Additive	Additive	Somewhat	Somewhat	Fully
Process Signed Integer	✓	✓	✓	✓	×	×	✓
Semi-honest Model	✓	✓	✓	✓	✓	✓	✓

setting and $t_{pv} = 1/\alpha' t_{up} + (\alpha' - 1)(2t_{rt} + t_{ad}) + \alpha' t_{cml} + 3t_{MS} + 3t_{KS}$ for multiple keys setting, where t_{up} is runtime of its corresponding unpack circuit and $\alpha' = \lfloor \ell/\mu' \rfloor$.[7] Note our packed version can be accelerated if and only if $t_{up} \geq \alpha'(2t_{rt} + t_{ad}) + \frac{(\alpha')^2}{\alpha'-1} t_{cml}$. Moreover, the communication cost of our packed circuits (i.e., sending two packed ciphertexts to CP to obtain the encrypted result costs 6.509 MB) is α' times smaller than the unpacked version for all the secure unsigned circuits of both single/multiple-key setting. Besides plaintext bit-length, `GSum` is also affected by the number of inputs. From Figs. 4a, 4b, and 4c, we observe that the running time of `GSum/GMin` increases with the number of inputs for both single and multiple key settings, and `GSum/GSum` is more efficient than using the corresponding `I.add/I.cmp` in [16] under the same μ, as our packing technique are adopted (see Table 4).

7.3.4 Performance of Secure Signed Integer Circuit

Similar to the experiments for unsigned integer circuits, we set the plaintext size $\mu = 8$ and security level $\lambda = 76$. The difference is to choose the `Ipack` parameter $\mu' = 17$ to avoid block overflow/underflow when preforming `I.Smul` (see the discussion in Section 5.3). Before testing signed integer circuits, we first evaluated the performance of both single-key and multiple-key setting for `STC`, `OTC`, `I.Sadd`, `I.Smul` and `I.Sdiv` and its packed version, and list the results in Table 5. Also, we observe that the running time of all the above secure signed integer circuits (single key setting from Figs. 4a, 4b, and 4c, and multiple-key setting from Figs. 4a, 4b, and 4c,) increase with the plaintext bit-length, and our packed signed integer circuits are much more efficient than their corresponding unpacked counterparts. Similar to the unpacked secure circuits, the communication cost of our packed circuits are α' times smaller than the unpacked version for all the secure signed circuits of both single/multiple key setting. The computation complexity of the unpacked/packed secure signed circuits are the same to the analysis in Section 7.3.3.

7.3.5 Performance of Secure k-NN Classifier

There are four factors that affect the performance of the secure k-NN classifier, namely: plaintext bit length μ, number of instances β, number of features χ and parameter k. In Figs. 4d and 4e, we observe that the running time of the secure k-NN classifier increases with μ, β, χ, k, but only slightly with α. This is because only fixed levels (only 10 levels for $\lambda = 76$) are left after bootstrapping, and more bootstrappings are required when μ and k are large, which has a more pronounced impact by k. Fortunately, k cannot be too large in practice (see the discussion in Section 6).

7. The $t_{rt}, t_{ad}, t_{ml}, t_{cml}, t_{MS}, t_{KS}$ denote the runtime of `H.rotate`, `H.add`, `H.mul`, `H.cmul`, `ModSwitch`, `KeySwitch` in BGV, respectively. (see Table 1 for efficiency).

7.4 Comparative Analysis

Here, we compare related schemes reported in [13], [14], [15], [16], [17], [18]. In [18], Liu et al. used a secure computation protocol to construct a Naïve Bayesian classifier. The system requires a data user to communicate with the cloud server over multiple rounds. Different from [18], Peter et al. [13] introduced an additional server to hold a strong private key in order to decrypt the ciphertexts in the system and execute the secure computation protocols, in order to avoid multiple rounds of communication between the user and the cloud. However, any leakage or (insider) abuse would compromise the system. To mitigate this limitation, Liu et al. [15] proposed a framework for outsourced integer calculation. The authors provided calculation toolkits for multiple keys setting [14], which separate strong trapdoor into two shares to reduce key leaking risk. Although these frameworks are efficient, two non-colluding servers are still required which limits the applicability of the frameworks in practice. In order to avoid needing an additional server during computation, Naehrig et al. [32] used the somewhat homomorphic encryption scheme [34] to perform unsigned integer calculation on encrypted data. More recently, Cheon et al. [16] used the somewhat homomorphic encryption scheme to perform search and computation over encrypted unsigned integer data. However, the computation circuit in these two schemes cannot be too large (i.e., they only support a limited number of calculations) and the schemes work only some simple circuits (unsigned integer addition, comparison, and equality circuits) in the single key setting. A comparative summary among these schemes is shown in Table 6.

8 Related Work

Partial homomorphic encryption (PHE) (including additive [35] and multiplicative [36]) is often considered viable solutions to address privacy computation challenges (see Table 6). However, an additional server is generally required to assist the computation server to execute the underlying secure protocols. Somewhat homomorphic encryption (SWHE) [34] can support both addition and multiplication at the same time. Unfortunately, only finite steps of homomorphic operations can be executed which limits its application. Fully homomorphic encryption (FHE) allows the cloud server to manipulate over ciphertexts to effect operations on the corresponding plaintexts, and is widely considered to be the ultimate solution for secure computation. The first generation of FHE was proposed by Gentry [37], who constructed a SWHE scheme and made it bootstrappable, i.e., capable of evaluating its own decryption circuit and then at least one more operation. Although several optimizations and refinements were proposed [38], [39], these schemes were impractical as the ciphertext size and computation time significantly with the security level.

Brakerski and Vaikuntanathan [40] proposed a scheme based on the hardness of the Learning with errors (LWE) problem. In a separate work, Brakerski et al. [19] built a new efficient tool to reduce ciphertext noise. Gentry et al. [41] also presented the approximate eigenvector method, designed to make homomorphic addition and multiplication more efficient. Smart and Vercauteren [22] outlined a technique to enable packing of many plaintext values in a single ciphertext and operate on all the plaintext values in a SIMD fashion. Halevi and Shoup [31] built the HElib library to implement the BGV cryptosystem and bootstropping method [33]. Gentry et al. [20] implemented the AES-encryption circuit under HElib. Although a number of FHE schemes have been proposed in the literature, all homomorphic operations are build over the ring and cannot be directly deployed in a real-world environment, such as constructing efficient integer operations. That is the gap that this paper tries to bridge.

9 Conclusion and Discussion

In this paper, we proposed Pockit, a novel privacy-preserving outsourced calculation toolkit for cloud computing environment. Pockit is designed to achieve secure outsourced computation. We proposed a new methodology for secure outsourced unsigned/signed integer number calculation over packed ciphertext in the SIMD fashion. We then demonstrated the security and practicality of our proposed solutions. There are a number of extensions for this work,

1) How to design a more efficient FHE scheme? The FHE scheme that we use in Pockit is not as efficient as the framework using partial homomorphic encryption, particularly during ciphertext noise reduction such as bootstrapping which negatively impacts the performance of the overall system.
2) How to design secure circuits to perform calculations on large integers? The running time of the circuits in Pockit grows significantly as the plaintext size increases; thus, it is highly desirable to propose new techniques for constructing secure circuits that supports efficient large integer calculations.

Acknowledgments

The authors thank the associate editor and reviewers for their constructive and generous feedback. This research is supported in part by the AXA Research Fund, National Natural Science Foundation of China under No. 61702105 and No. 61402112.

References

[1] Cisco, "Cisco global cloud index: Forecast and methodology, 2014–2019 white paper," (2018, Feb.). [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.html
[2] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann, "An evaluation of multi-resolution storage for sensor networks," in *Proc. 1st Int. Conf. Embedded Netw. Sensor Syst.*, 2003, pp. 89–102.
[3] V. Kundra, "Federal cloud computing strategy." (2011). [Online]. Available: http://www.whitehouse.gov/sites/default/files/omb/assets/egov_docs/federal-cloud-computing-strategy.pdf
[4] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Comput.*, vol. 7, no. 1, pp. 76–80, Jan./Feb. 2003.
[5] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[6] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, 2015.
[7] A. A. Cruz-Roa, J. E. A. Ovalle, A. Madabhushi, and F. A. G. Osorio, "A deep learning architecture for image representation, visual interpretability and automated basal-cell carcinoma cancer detection," in *Proc. Int. Conf. Med. Image Comput. Comput.-Assisted Intervention*, 2013, pp. 403–410.

[8] R. Agrawal and R. Srikant, "Privacy-preserving data mining," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 439–450, 2000.
[9] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding, "Data mining with big data," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 1, pp. 97–107, Jan. 2014.
[10] D. He, N. Kumar, H. Wang, L. Wang, K. R. Choo, and A. Vinel, "A provably-secure cross-domain handshake scheme with symptoms-matching for mobile healthcare social network," *IEEE Trans. Depend. Secure Comput.*, 2016, doi: 10.1109/TDSC.2016.2596286.
[11] Q. Alam, S. U. R. Malik, A. Akhunzada, K. R. Choo, S. Tabbasum, and M. Alam, "A cross tenant access control (CTAC) model for cloud computing: Formal specification and verification," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 6, pp. 1259–1268, Jun. 2017.
[12] S. Kamara and K. Lauter, "Cryptographic cloud storage," in *Proc. Int. Conf. Financial Cryptography Data Secur.*, 2010, pp. 136–149.
[13] A. Peter, E. Tews, and S. Katzenbeisser, "Efficiently outsourcing multiparty computation under multiple keys," *IEEE Trans. Inf. Forensics Secur.*, vol. 8, no. 12, pp. 2046–2058, Dec. 2013.
[14] X. Liu, R. H. Deng, K. R. Choo, and J. Weng, "An efficient privacy-preserving outsourced calculation toolkit with multiple keys," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 11, pp. 2401–2414, Nov. 2016.
[15] X. Liu, K. R. Choo, R. H. Deng, R. Lu, and J. Weng, "Efficient and privacy-preserving outsourced calculation of rational numbers," *IEEE Trans. Depend. Secure Comput.*, vol. 15, no. 1, pp. 27–39, Jan./Feb. 2016.
[16] J. H. Cheon, M. Kim, and M. Kim, "Optimized search-and-jcompute circuits and their application to query evaluation on encrypted data," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 1, pp. 188–199, Jan. 2016.
[17] Y. Rahulamathavan, S. Veluru, R. C.-W. Phan, J. A. Chambers, and M. Rajarajan, "Privacy-preserving clinical decision support system using Gaussian kernel-based classification," *IEEE J. Biomed. Health Informat.*, vol. 18, no. 1, pp. 56–66, Jan. 2014.
[18] X. Liu, R. Lu, J. Ma, L. Chen, and B. Qin, "Privacy-preserving patient-centric clinical decision support system on naive Bayesian classification," *IEEE J. Biomed. Health Informat.*, vol. 20, no. 2, pp. 655–668, Mar. 2016.
[19] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *Proc. 3rd Innovations Theoretical Comput. Sci. Conf.*, 2012, pp. 309–325.
[20] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *Proc. Annu. Cryptology Conf. Advances Cryptology*, 2012, pp. 850–867.
[21] C. Gentry, S. Halevi, and N. P. Smart, "Better bootstrapping in fully homomorphic encryption," in *Proc. Int. Workshop Public Key Cryptography*, 2012, pp. 1–16.
[22] N. P. Smart and F. Vercauteren, "Fully homomorphic SIMD operations," *Des. Codes Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.
[23] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2012, pp. 465–482.
[24] T. Denoeux, "A k-nearest neighbor classification rule based on Dempster-Shafer theory," *IEEE Trans. Syst. Man Cybern.*, vol. 25, no. 5, pp. 804–813, May 1995.
[25] M. Mohandes, M. Deriche, and J. Liu, "Image-based and sensor-based approaches to arabic sign language recognition," *IEEE Trans. Human-Mach. Syst.*, vol. 44, no. 4, pp. 551–557, Aug. 2014.
[26] P. B. Callahan and S. R. Kosaraju, "A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields," *J. ACM*, vol. 42, no. 1, pp. 67–90, 1995.
[27] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios, "k-nearest neighbors in uncertain graphs," *Proc. VLDB Endowment*, vol. 3, no. 1/2, pp. 997–1008, 2010.
[28] M. Clear and C. McGoldrick, "Multi-identity and multi-key leveled FHE from learning with errors," in *Proc. Annu. Cryptology Conf.*, 2015, pp. 630–656.
[29] B. Pinkas and T. Reinman, "Oblivious RAM revisited," in *Proc. Annu. Cryptology Conf.*, 2010, pp. 502–519.
[30] M. T. Goodrich and M. Mitzenmacher, "Privacy-preserving access of outsourced data via oblivious RAM simulation," in *Proc. 38th Int. Conf. Automata Languages Programm.*, 2011, pp. 576–587.
[31] S. Halevi and V. Shoup, "HElib-an implementation of homomorphic encryption," 2014. [Online]. Available: https://github.com/shaih/HElib
[32] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proc. 3rd ACM Workshop Cloud Comput. Secur. Workshop*, 2011, pp. 113–124.
[33] S. Halevi and V. Shoup, "Bootstrapping for HElib," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2015, pp. 641–670.
[34] J. Loftus, A. May, N. P. Smart, and F. Vercauteren, "On CCA-secure somewhat homomorphic encryption," in *Proc. Int. Workshop Sel. Areas Cryptography*, 2011, pp. 55–72.
[35] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 1999, pp. 223–238.
[36] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," in *Proc. Workshop Theory Appl. Cryptographic Techn.*, 1984, pp. 10–18.
[37] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, USA, 2009.
[38] D. Stehlé and R. Steinfeld, "Faster fully homomorphic encryption," in *Proc. Int. Conf. Theory Appl. Cryptology Inf. Secur.*, 2010, pp. 377–394.
[39] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Proc. Int. Workshop Public Key Cryptography*, 2010, pp. 420–443.
[40] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," *SIAM J. Comput.*, vol. 43, no. 2, pp. 831–871, 2014.
[41] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Proc. Advances Cryptology–CRYPTO*, 2013, pp. 75–92.
[42] J. Gallian, *Contemporary Abstract Algebra*. Boston, MA, USA: Cengage Learning, 2016.
[43] I. D. V. Pastro, N. Smart, S. Zakarias, et al., "Multiparty computation from somewhat homomorphic encryption," in *Proc. Annu. Cryptology Conf. Advances Cryptology*, 2012, pp. 643–662.

Ximeng Liu (S'13-M'16) received the BSc degree in electronic engineering from Xidian University, Xian, China, in 2010 and the PhD degrees in cryptography from Xidian University, China, in 2015. Now, he is a research fellow with the School of Information System, Singapore Management University, Singapore. He has published more than 80 research articles include the *IEEE Transactions on Information Forensics and Security*, the *IEEE Transactions on Dependable and Secure Computing*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Services Computing*, the *IEEE Transactions on Industrial Informatics*, and the *IEEE Transactions on Cloud Computing*. His research interests include cloud security, applied cryptography, and big data security. He is a member of the IEEE.

Robert H. Deng (F'16) is AXA chair professor of cybersecurity and professor of Information Systems with the School of Information Systems, Singapore Management University since 2004. His research interests include data security and privacy, multimedia security, network, and system security. He served/is serving on the editorial boards of many international journals, including the *IEEE Transactions on Information Forensics and Security* and the *IEEE Transactions on Dependable and Secure Computing*. He is a fellow of the IEEE.

Kim-Kwang Raymond Choo (SM'16) received the PhD degree in information security from the Queensland University of Technology, Australia, in 2006. He currently holds the Cloud Technology Endowed professorship with the University of Texas at San Antonio. He is the recipient of various awards including ESORICS 2015 Best Paper Award, Winning Team of the Germany's University of Erlangen-Nuremberg (FAU) Digital Forensics Research Challenge 2015, and British Computer Society's Wilkes Award in 2008. He is also a fellow of the Australian Computer Society, and a senior member of the IEEE.

Yang Yang (M'17) received the BSc degree from Xidian University, Xi'an, China, in 2006 and the PhD degrees from Xidian University, China, in 2012. She is a research fellow (postdoctor) under supervisor Robert H. Deng with the School of Information System, Singapore Management University. She is also an associate professor with the College of Mathematics and Computer Science, Fuzhou University. Her research interests include the area of information security and privacy protection. She is a member of the IEEE.

HweeHwa Pang received the BSc (first class honors) and MS degrees from the National University of Singapore, in 1989 and 1991, respectively, and the PhD degree from the University of Wisconsin-Madison, in 1994, all in computer science. He is a professor with the School of Information Systems, Singapore Management University. His current research interests include database management systems, data security, and information retrieval.