SOCT: Secure Outsourcing Computation Toolkit using Threshold ElGamal Algorithm

Sen Hu, Shang Ci, Donghai Guan, and Çetin Kaya Koç *IEEE Fellow*

***Abstract*—Cloud computing offers inexpensive and scalable solutions for data processing, however privacy concerns often hinder the outsourcing of sensitive information. Homomorphic encryption provides a promising approach for secure computations over encrypted data. However, existing models often rely on restrictive assumptions, such as semi-honest adversaries and inaccessible public data.**

To address these limitations, we introduce the Secure Outsourcing Computation Toolkit (SOCT), which is a novel framework based on the threshold ElGamal cryptosystem. The toolkit employs a dual-server decryption architecture using a (2,2) threshold additively homomorphic ElGamal (TAHEG) algorithm. This architecture ensures that ciphertexts can be decrypted only with the cooperation of both servers, mitigating the risk of data breaches. The TAHEG algorithm requires the input of a secret key for every decryption operation, preventing unauthorized access to plaintext data. Moreover, the key generation process does not burden users with generating or distributing partial secret keys. We provide rigorous security proofs for our threshold ElGamal cryptosystem and associated secure computation functions.

Experimental results demonstrate that SOCT achieves significant efficiency gains compared to existing toolkits, making it a practical choice for privacy-preserving data outsourcing.

I. Introduction

Cloud servers offer computing power and storage space through outsourced computing for users who bring in their data, and cloud servers perform required computation tasks on the data. However, disclosing the sensitive data to servers has raised privacy concerns for users, which limits the application of outsourced computing [1].

To protect the privacy of sensitive data in outsourcing scenarios, users can choose homomorphic encryption (HE) algorithms to achieve high-level security [2]. Craig Gentry [3] proposed fully homomorphic encryption (FHE) to enable encrypted data to support multiplication and addition operations, and therefore through them, any algebraic function. Following Gentry's template, the CKKS algorithm [4] supports floating-point addition and multiplication operations, and the torus-based fully homomorphic encryption (TFHE) [5] efficiently supports homomorphic operations on binary data. However, these FHE schemes need to be used with the so-called *bootstrapping* operation to achieve full homomorphic properties, which results in low efficiency.

Sen Hu, Shang Ci, and Donghai Guan are with Nanjing University of Aeronautics and Astronautics.
E-mail: {hu_sen, cishang, dhguan}@nuaa.edu.cn

Çetin Kaya Koç is with Nanjing University of Aeronautics and Astronautics, Iğdır University, and University of California Santa Barbara.
E-mail: cetinkoc@ucsb.edu

FHE supports homomorphic multiplication and addition operations, and therefore any algebraic function through approximations. For example, comparison operation often appears in such scenarios for which Lee et al. [6] approximated the sign function using compositions of minimax approximation polynomials. Ong et al. [7] evaluated the efficiency of TFHE digital gates. Wang et al. [8] designed a comparison and sorting algorithm based on a self guided binary gate from the TFHE library. However, the approximation error and the multiplication usage limit the accuracy and efficiency of [6], and circuit depth and operand bit length limit the efficiency of [7], [8].

To improve the HE efficiency in applications, partially homomorphic encryption (PHE) has received attention [9], [10], which supports either homomorphic addition or homomorphic multiplication, but not both. In general, the PHE is more efficient than FHE, since it is constructed using algebraic structures with shorter operands. Unfortunately, additive or multiplicative homomorphism may not meet the outsourcing requirements. To enable PHE to support more homomorphic computations, a common approach is to combine the PHE with *multiple servers computing architectures* [11], [12]. These cloud servers interactively perform secure computing functions to achieve homomorphic computation. Data owners can be offline during the function execution, which reduces overhead.

Previous work on multiple decryption servers architecture are found in [12], [13], [14], [15], [16]. Among them the work performed by Zhao et al. [12], [16] needs particular attention due to its similarity to our proposal. They proposed a secure outsourced computation toolkit for integers, based on Paillier cryptosystem with threshold decryption (PaillierTD) [17], [18] and optimized Paillier with threshold decryption (FastPaiTD) [16]. The user generates key pairs, splits secret key and distributes the shares of secret key between 2 cloud servers. However, the user needs to send partial secret keys to cloud servers before the function execution, which requires the existence of secure channels between the user and the cloud servers.

II. Security Issues in Multiple Server Scenarios

For privacy concerns, servers participating in cloud computing are often assumed to be semi-honest (also called, honest-but-curious), that is, they will follow the system specifications, but attempt to learn user's private data from accessible materials.

On the other hand, adversaries sometimes are assumed to be semi-honest, that is, they would only receive the materials involved in a specific function. For example, if there are 3 ciphertexts $E(m_1), E(m_2), E(m_3)$, and the function uses only $E(m_1), E(m_2)$ as input, then it is assumed that the adversaries will not have access to $E(m_3)$. However, in reality, adversaries may easily obtain any ciphertexts, which is a general principle in almost all cryptographic settings. Therefore, we cannot assume that the adversary is semi-honest.

Existing studies [11], [12], [16] consider only untrusted servers as semi-honest adversaries, which is unrealistic. Although [12], [16] adds the definition of active semi-honest adversaries compared to [11], the partially decrypted ciphertext in [12], [16] still needs to be kept private. Common sense dictates that the server will keep the secret key and the random number selected in the encryption as private. But, there is not much motivation to protect data that "should be" secure by its very nature, such as ciphertext and partially decrypted ciphertext. This will make it easy for external parties to obtain ciphertext and partially decrypted ciphertext in the outsourcing process. An external organization can execute the threshold decryption function to obtain the plaintext with a sufficient number of partially decrypted ciphertexts in the PaillierTD cryptosystem.

In this paper, we consider internal adversaries (cloud server CS) and external adversaries (hacker $\mathcal{H}$) in outsourced computing. CS is considered as an active semi-honest adversary. He will obtain the input and output of any function and actively execute the selected function. $\mathcal{H}$ is considered an active non-malicious adversary. He can easily copy public data (e.g. public key, ciphertext, partially decrypted ciphertext) from CS's internal storage, but will not disrupt the outsourcing process.

~~Therefore, in reality, external hackers do not necessarily adhere to the semi-honest assumption, and there is a gap between the security assumptions and the real scenarios. As a result, the hackers have the ability to perform cryptographic operations in an attempt to gain access to private information. It is necessary to ensure that hackers do not learn private information from cloud computing.~~

~~In this paper, we consider the adversary as an untrusted server in cloud computing or an external organization or individual, which we denote as the hacker $\mathcal{H}$.~~

In our proposed multi-server scenario, the data owners encrypt their sensitive data and send ciphertext to the server, believing that the ciphertext will not leak their privacy. Meanwhile, the servers are allowed to have secret keys, and therefore, we need to take into account certain privacy leakage problems:

- Case 1: The privacy leaks if a ciphertext can be obtained by an adversary who holds the secret key.
- Case 2: The privacy leaks if the partial decryption function does not require any secret key.

The privacy leakage Case 1 may occurs in the single server decryption architecture such that the server holds the secret key. It is necessary to ensure the decryption server does not obtain a copy of the original ciphertext. For example, [11] assumes the decryption server cannot obtain the original ciphertext, however, this is difficult to achieve in reality. The multiple servers decryption architecture can solve the Case 1 privacy leakage. Multiple servers hold **partial decryption keys**, and the threshold setting ensures the minimum number of partial keys to decrypt ciphertexts. For example, a threshold setting of (t, n) requires at least t of the n partial keys to decrypt the ciphertext. This liberates the server from treating the ciphertexts as private data.

On the other hand, the privacy leakage Case 2 would occur in the implementation of Paillier cryptosystem with additively threshold decryption (PaillierTD) [12] in a multiple server decryption scenario. The decryption algorithm of the PaillierTD works as:

$$\begin{aligned} X_1 &\leftarrow \texttt{PDec}(E(m), sk_1) \\ X_2 &\leftarrow \texttt{PDec}(E(m), sk_2) \\ m &\leftarrow \texttt{TDec}(X_1, X_2) \end{aligned}$$

`PDec` function takes the ciphertext $E(m)$ and secret key sk_i as input, and outputs the partial decrypted ciphertext X_i for $i \in \{1, 2\}$. A single X_1 or X_2 cannot be obtained without the corresponding partial secret key. Thus, we can consider X_1, X_2 as publicly available data. However, `TDec` function can recover plaintext m without any secret key. Assuming that 2 servers execute the `PDec` function and obtain X_1, X_2 respectively, they might consider X_1, X_2 to be secure. However, a hacker $\mathcal{H}$ may easily obtain X_1, X_2, and perform `TDec` function to obtain the plaintext m. This leads to privacy leakage. Fortunately, the current PaillierTD based functions [12], [16] do not transmit X_1, X_2 together in a single function execution. If the servers do not leak X_1, X_2 beyond function transmission, it is secure in their assumption.

To compare, we consider the threshold ElGamal which contains **only** `PDec` function in decryption process. We have 2 secret keys sk_1, sk_2 such that their subsequent application to the `PDec` function produces the plaintext m, as follows:

$$\begin{aligned} X_1 &\leftarrow \texttt{PDec}(E(m), sk_1) \\ m &\leftarrow \texttt{PDec}(X_1, sk_2) \end{aligned}$$

The keys sk_1, sk_2 are given to 2 cloud servers, and therefore, the disclosure of partially decrypted ciphertexts $X_1, X_2, \ldots$ to the hacker does not cause any privacy leakage. However, this model still requires the cloud servers be **non-collusive**. If collusion is allowed in this multiple server architecture, the servers may share their secret keys, and they can decrypt the ciphertext.

III. Contributions

In this paper, we address the security problems in an HE-based outsourcing computation architecture in multiple server scenarios. We identified two cases of privacy leakages in Section 2 for the existing secure computing architectures. We propose the Secure Outsourcing Computation Toolkit (SOCT) based on the Threshold Additively Homomorphic ElGamal Algorithm (TAHEG) which brings robust solutions to the mentioned security problems. Compared with the existing schemes [12], [13], [14], [16], SOCT achieves higher efficiency, and optimized key generation and distribution processes. The security assumptions of the SOCT make it more suitable for real-world applications. We have 3 contributions in this paper:

- **Real-World Security Assumptions:**
  We identify the problems in the existing schemes, and define a novel security model that is closer to real-world scenarios. We take into account external adversaries who do not adhere the semi-honest assumptions. In real-world scenarios, hackers can obtain public keys, ciphertexts, and partially decrypted ciphertexts through transmission channels, function outputs, and even server internal storage. We propose the secure outsourcing computing toolkit (SOCT) based on this threat model.
- **Secure Outsourcing Architecture based on TAHEG:**
  We employ the $(2,2)$ TAHEG algorithm and built a novel outsourced computing architecture. We optimize partial key generation and distribution processes. The TAHEG key generation function can be executed in parallel by multiple participants. Our TAHEG is more secure than PaillierTD [12] algorithm in the partial decryption processes, since each decryption function in TAHEG requires the input of the secret key.
- **Secure Outsourcing Computation Toolkit:**
  SOCT contains all basic functions, but we also add 3 new functions: Secure multiplication (`SMUL`), secure comparison (`SCOM`), and secure sign and magnitude acquisition (`SSMA`). We provide the correctness and security proofs for these functions. Experimental results show that SOCT achieves higher efficiency compared to other advanced proposals [12], [13], [14], [16].

IV. Threshold Additively Homomorphic ElGamal

A (t,n) threshold cryptosystem [19], [20] shares a secret among n participants, so that the secret cannot be obtained unless at least t participants participate to compute it. Similarly a (t,n) threshold public-key encryption algorithm shares the private key among n participants, so that the private key can be recovered by t or more shares; therefore, a ciphertext can be decrypted only if at least t participants' shares are available.

The ElGamal cryptosystem has been described in [21], which has multiplicative homomorphic property. The additively homomorphic ElGamal algorithm [10] was developed from the ElGamal cryptosystem [21]. The threshold ElGamal cryptosystem has been implemented using threshold settings [19], [20] and the $(3,n)$ threshold ElGamal encryption algorithm was described in [20].

In this subsection, we present a $(2,2)$ threshold additively homomorphic ElGamal algorithm (TAHEG), which serves as the basic threshold PHE algorithm within our proposed secure computation toolkit. The TAHEG algorithm consists of the following functions:

- *Parameter Setup*, $G_q(p,q,g) \leftarrow \textit{ParSet}(\lambda)$.
  Take security public parameter λ, and output the cyclic subgroup parameters $G_q(p,q,g)$, where g is the group generator, q is the order of g, and p is the modulus. The p,q are large primes ($p > 2048$ bits and $q > 256$ bits) to ensure the hardness of discrete logarithm problem (DLP).
- *Key Generation*: $(pk, sk) \leftarrow \textit{KeyGen}(G_q)$.
  Select a random number $x \in \mathbb{Z}_q$ as the secret key, and compute $h = g^x \bmod p$. Denote the public key as $pk(p,q,g,h)$ and the secret key as $sk(x)$. Different key pairs can be generated under the same cyclic subgroup $G_q(p,q,g)$ by choosing different secret keys x. For example, two different key pairs can be represented as $\{pk_1(p,q,g,h_1), sk_1(x_1)\}$, $\{pk_2(p,q,g,h_2), sk_2(x_2)\}$, where $x_1 \neq x_2$.
- *Encryption*: $E(m) \leftarrow \textit{Enc}(m, pk_1, pk_2)$.
  The plaintext $m \in \mathbb{Z}_q$ can be encrypted with multiple public keys of pk_1, pk_2. The data owner select the random number $r \in \mathbb{Z}_q$, and compute the ciphertext as
  $$E(m) = (g^r,\ g^m \times h_1^r \times h_2^r \bmod p).$$
  The random number r should be different for each encryption. The plaintext m is selected from a smaller domain to ensure decryption efficiency, for example, $m \in \mathbb{Z}_{2^{32}}$.
- *Decryption*: $m \leftarrow \textit{Dec}(E(m), sk_1, sk_2)$.
  The ciphertext can be decrypted using all involved secret keys. Take the ciphertext $E(m)$ and secret keys sk_1, sk_2 as input, and compute
  $$g^m = (g^m \times h_1^r \times h_2^r) \times (g^r)^{-x_1} \times (g^r)^{-x_2} \bmod p,$$
  where the $(g^r)^{-x_1} = h_1^{-r}, (g^r)^{-x_2} = h_2^{-r}$. After g^m was obtained, the plaintext m is extracted from the exponent using the discrete logarithm function $m \leftarrow \mathrm{DLOG}(g^m)$.
- *Partial Decryption*: $P(m) \leftarrow \textit{PartDec}(E(m), sk_i)$.
  Since m is encrypted by both public keys pk_1 and pk_2, $E(m)$ can only be decrypted by both secret keys sk_1 and sk_2. We define the partial decryption function which performs a "half decryption" with one of the secret keys, to be completed by the other secret key in another step. Partial decryption is performed by CS1 or CS2 using the secret key sk_1 or sk_2, respectively. If partial decryption is performed by sk_1 to obtain $P(m)$, then $P(m)$ can be decrypted by sk_2 to obtain m, and vice versa.
  $$\begin{aligned} P(m) &= (g^r,\ g^m \cdot h_1^r \cdot h_2^r \times (g^r)^{-x_1}) \\ &= (g^r,\ g^m \cdot h_2^r) \\ g^m &= (g^r,\ g^m \cdot h_2^r \times (g^r)^{-x_2} \\ m &\leftarrow \mathrm{DLOG}(g^m) \end{aligned}$$
- *Homomorphic Addition with Ciphertext*:
  $E(m_1 + m_2) \leftarrow \textit{Add}(E(m_1), E(m_2))$.
  The homomorphic addition function takes two ciphertexts $E(m_1), E(m_2)$ and the public key $pk(p,q,g)$ as input, and outputs the ciphertext $E(m_1 + m_2)$.
  $$\begin{aligned} E(m_1) &= (g^r, g^{m_1} \times h_1^r \times h_2^r), \\ E(m_2) &= (g^u, g^{m_2} \times h_1^u \times h_2^u), \\ E(m_1 + m_2) &= (g^r \times g^u, g^{m_1} \times g^{m_2} \times h_1^{r+u} \times h_2^{r+u}). \end{aligned}$$
- *Homomorphic Addition with Scalar*:
  $E(m_1 + s) \leftarrow \textit{AddSca}(E(m_1), s)$.
  The homomorphic scalar addition function takes a ciphertext $E(m_1)$ and a scalar $s \in \mathbb{Z}_q$, and also both public keys $pk_1(p,q,g,h_1), pk_2(p,q,g,h_2)$ as input, and outputs the ciphertext $E(m_1 + s)$. It performs the encryption

TABLE I
BASIC TAHEG FUNCTIONS

Name	Function	Participants
Parameter Setup	$G_q(p,q,g) \leftarrow \mathit{ParSet}(\lambda)$	DO
Key Generation	$(pk_1, sk_1) \leftarrow \mathit{KeyGen}(G_q)$	CS1
Key Generation	$(pk_2, sk_2) \leftarrow \mathit{KeyGen}(G_q)$	CS2
Encryption	$E(m) \leftarrow \mathit{Enc}(m, pk_1, pk_2)$	DO or CS1 or CS2
Decryption	$m \leftarrow \mathit{Dec}(E(m), sk_1, sk_2)$	CS1 and CS2
Partial Decryption	$P(m) \leftarrow \mathit{PartDec}(E(m), sk_i)$	CS1 or CS2, $i = 1$ or 2
Decryption after Partial Decryption	$m \leftarrow \mathit{Dec}(P(m), sk_j)$	CS2 or CS1, $j = 2$ or 1
Homomorphic Addition with Ciphertext	$E(m_1 + m_2) \leftarrow \mathit{Add}(E(m_1), E(m_2))$	CS1 or CS2
Homomorphic Addition with Scalar	$E(m_1 + s) \leftarrow \mathit{AddSca}(E(m_1), s)$	CS1 or CS2
Homomorphic Multiplication with Scalar	$E(m_1 \times s) \leftarrow \mathit{MulSca}(E(m_1), s)$	CS1 or CS2

function $E(m_1 + s) \leftarrow \mathit{Enc}(m, pk_1, pk_2)$ to encrypt the scalar number $s \in \mathbb{Z}_q^*$ with pk_1, pk_2. It then executes the following equations to compute $E(m_1 + s)$.

$$\begin{aligned} E(m_1) &= (g^r, g^{m_1} \times h_1^r \times h_2^r), \\ E(s) &= (g^u, g^s \times h_1^u \times h_2^u), \\ E(m_1 + s) &= (g^r \times g^u, g^{m_1} \times g^s \times h_1^{r+u} \times h_2^{r+u}). \end{aligned}$$

- *Homomorphic Multiplication with Scalar*:
  $E(m_1 \times s) \leftarrow \mathit{MulSca}(E(m_1), s)$.
  The homomorphic scalar multiplication function takes a ciphertext $E(m_1)$ and a scalar $s \in \mathbb{Z}_q$ as input, and outputs the ciphertext $E(m_1 \times s)$, computed as

$$\begin{aligned} E(m_1) &= (g^r, g^{m_1} \times h_1^r \times h_2^r), \\ E(m_1 \times s) &= [(g^r)^s, (g^{m_1} \times h_1^r \times h_2^r)^s]. \end{aligned}$$

  When $E(m_1 \times s)$ is obtained, it is necessary to refresh the related random number r. This operation can be performed by a scalar addition $E(m_1 \times s + 0) \leftarrow \mathit{AddSca}(E(m_1 \times s), 0)$

The set of functions given above are summarized in Table 1. However, they are not complete for implementing most common privacy-preserving applications, such as PPML (Privacy-Preserving Machine Learning). We need to add three new functions to the set that allow two cloud servers CS1 and CS2 collaboratively implement any privacy-preserving application. These functions are

- Secure Multiplication Function (`SMUL`)
- Secure Comparison Function (`SCOM`)
- Secure Sign and Magnitude Acquisition Function (`SSMA`)

These new functions are defined and described in Section 6. The executions of these functions require the participation of the data owner (DO) or both cloud servers (CS1, CS2). Since secure outsourcing computation requires interactions between the data owner and the cloud servers, a system architecture needs to be established. Several existing systems define either a single server which keeps the entire secret key or two servers each of which holds a part of the secret key. The dual-server decryption model is more complex but also more secure since it withstands to the single point of failure [12].

V. SYSTEM ARCHITECTURE AND THREAT MODEL

In our model, there are 4 participants: Data Owner (DO), Cloud Server 1 (CS1), Cloud Server 2 (CS2), and Hacker ($\mathcal{H}$).

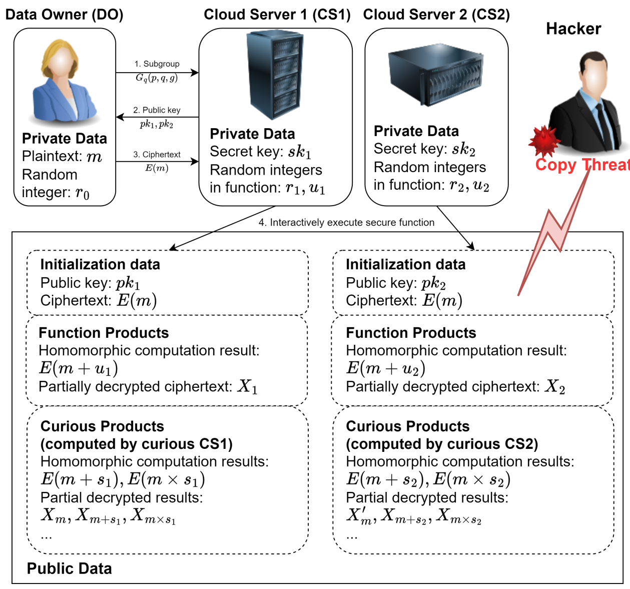
Data Owner (DO)
Cloud Server 1 (CS1)
Cloud Server 2 (CS2)
Hacker
Private Data
Plaintext: m
Random integer: r_0
1. Subgroup $G_q(p,q,g)$
2. Public key pk_1, pk_2
3. Ciphertext $E(m)$
Private Data
Secret key: sk_1
Random integers in function: r_1, u_1
Private Data
Secret key: sk_2
Random integers in function: r_2, u_2
Copy Threat
4. Interactively execute secure function
Initialization data
Public key: pk_1
Ciphertext: $E(m)$
Initialization data
Public key: pk_2
Ciphertext: $E(m)$
Function Products
Homomorphic computation result: $E(m+u_1)$
Partially decrypted ciphertext: X_1
Function Products
Homomorphic computation result: $E(m+u_2)$
Partially decrypted ciphertext: X_2
Curious Products (computed by curious CS1)
Homomorphic computation results: $E(m+s_1), E(m \times s_1)$
Partial decrypted results: $X_m, X_{m+s_1}, X_{m \times s_1}$ ...
Curious Products (computed by curious CS2)
Homomorphic computation results: $E(m+s_2), E(m \times s_2)$
Partial decrypted results: $X'_m, X_{m+s_2}, X_{m \times s_2}$ ...
Public Data

Fig. 1. System Architecture and Threat Model

- **DO** generates and distributes the cyclic subgroup parameters $G_q(p,q,g)$. After receiving cloud servers' public keys (pk_1, pk_2), DO encrypts plaintext data with multiple public keys. Then, DO sends the encrypted data to the cloud servers CS1 and CS2, as shown in Figure 1. DO can be offline during the outsourced computing process. When the outsourcing computation is completed, DO receives the encrypted results and severs' secret keys (sk_1, sk_2) from the cloud servers.
- **CS1** generates his own key pairs (pk_1, sk_1), and broadcasts the public key pk_1. CS1 interactively performs functions with CS2.
- **CS2** generates his own key pairs (pk_2, sk_2), and broadcasts the public key pk_2. CS2 interactively performs functions with CS1.
- $\boldsymbol{\mathcal{H}}$ can easily copy public data (e.g. public key, ciphertext, partially decrypted ciphertext) from cloud servers's internal storage, and he will not disrupt the outsourcing process.

Since the public key and ciphertext are secure in almost all encryption settings, the servers CS1, CS2 will treat them as public data. Although the cloud servers do not broadcast the encrypted intermediates and results during the function execution, it remains relatively easy for the hacker $\mathcal{H}$ to obtain them. We define the external individual or organization as the hacker $\mathcal{H}$, who is considered a threat to all of the public data in system model, as shown in Figure 1.

Participants DO, CS1, CS2 hold private data. When this data is disclosed by any untrusted party, security would be breached. Specifically, DO's plaintext data contains private information, and she hopes to outsource the computation without compromising privacy. CS1, CS2 generate secret keys and treat them as private data. Also, the CS1, CS2 carefully select random numbers in the function to counter hacker threats. Private data held by the three parties are as follows:

- DO: The plaintext m, and random number r_0 used during in the encryption.
- CS1: The secret key sk_1; the plaintext m and the random numbers r_1, u_1 used in the function.
- CS2: The secret key sk_2; the plaintext m and the random numbers r_2, u_2 used in the function.

In our threat model, the DO is considered as honest, the servers CS1, CS2 are considered as semi-honest (also called *honest-but-curious*), and the hacker $\mathcal{H}$ is considered a threat to all of the public data in system model.

- The cloud servers CS1, CS2 obtain data through the secure function passively. Curious cloud servers CS1, CS2 can also actively select plaintext and ciphertext, perform homomorphic operations and secure functions to obtain private data.
- The hacker $\mathcal{H}$ eavesdrops on the communication link between CS1 and CS2 to obtain data. The hacker $\mathcal{H}$ can disguise himself as a server and execute the secure function with either CS1 or CS2. The hacker $\mathcal{H}$ can copy public data from cloud servers' internal storage.

Moreover, an important assumption we make is that CS1 and CS2 are **non-collusive**; they will not exchange data in any way other than as specified in the functions. Also, the servers will not collude with hacker $\mathcal{H}$. We provide Figure 1 that includes all participants and the data they hold. In our security outsourcing computing model, public keys, ciphertexts, and partially decrypted ciphertexts are considered public data, which conforms to the general principles of cryptographic systems. The preparation and execution phases of the secure computation function generate public data, as shown in the "initialization data" and "function products" in Figure 1. More importantly, we consider the curious cloud server to be an active semi-honest adversary, who can perform homomorphic operations and secure functions to obtain private data.

VI. Three New Homomorphic Functions

Our proposed secure computation toolkit contains 3 new homomorphic functions which are needed for implementing privacy-preserving machine learning applications. These functions are Secure Multiplication Function (`SMUL`), Secure Comparison Function (`SCOM`), and Secure Sign and Magnitude Acquisition Function (`SSMA`).

We define the positive plaintexts as integers in $[1, q/2]$, and the negative plaintexts as integers in $[q/2, q-1]$ for the cyclic subgroup G_q. For instance, if $m > 0$ implies $m \in [1, q/2]$; otherwise, $m < 0$ then $m \in [q/2, q-1]$.

A. *Secure Multiplication Function (`SMUL`)*

The cloud servers CS1 and CS2 execute the secure multiplication function (`SMUL`) to compute $E(m_1 \times m_2)$ using the inputs $E(m_1)$ and $E(m_2)$. Even though CS1 and CS2 have access to sk_1 and sk_2 respectively, `SMUL` function given below computes $E(m_1 \times m_2)$ collaboratively with CS1 and CS2, and the plaintexts m_1 and m_2 are not exposed to the servers CS1, CS2 or the hacker.

Function 1 Secure Multiplication Function (`SMUL`)

```
Input: $E(m_1), E(m_2)$
Output: $E(m_1 \times m_2)$
Step 1. CS1 executes
    Generate random number $u \in \mathbb{Z}_q$
    $E(m_1 + u) \leftarrow \textit{AddSca}(E(m_1), u)$
    $P(m_1 + u) \leftarrow \textit{PartDec}(E(m_1 + u), sk_1)$
    Send $P(m_1 + u)$ to CS2
Step 2. CS2 executes
    $(m_1 + u) \leftarrow \textit{Dec}(P(m_1 + u), sk_2)$
    $E(m_2 \times (m_1 + u)) \leftarrow \textit{MulSca}(E(m_2), (m_1 + u))$
    Send $E(m_2 \times (m_1 + u))$ to CS1
Step 3. CS1 executes
    $E(m_2 \times u) \leftarrow \textit{MulSca}(E(m_2), u)$
    $E(m_1 \times m_2) \leftarrow E(m_2 \times (m_1 + u)) \times E(m_2 \times u)^{-1}$
```

The correctness of this function is based on the identity

$$m_2 \times (m_1 + u) - u \times m_2 = m_1 \times m_2 \ .$$

In the beginning of the algorithm the ciphertexts $E(m_1)$ and $E(m_2)$ are available for CS1 and CS2. Recall that $E(m_1)$ and $E(m_2)$ are obtained using both public keys pk_1 and pk_2, and cannot be decrypted without the corresponding secret keys sk_1 and sk_2. CS1 homomorphically computes $E(m_1 + u)$ using a random scalar u, and applies partial decryption function to obtain $P(m_1+u)$, and sends it to CS2 which can decrypt using its private key sk_2 to obtain $m_1 + u$, which is not equal to m_1, therefore, m_1 remains unknown. CS2 multiplies the scalar $m_1 + u$ with $E(m_2)$ using $MulSca$ function, and obtains $E(m_2 \times (m_1 + u))$, and sends it to CS1. Finally, CS1 performs the operation $E(m_2 \times (m_1 + u)) \times E(m_2 \times u)^{-1}$, and obtains

$$E(m_2 \times m_1 + m_2 \times u - m_2 \times u) = E(m_1 \times m_2)$$

This computation is accomplished without the non-colluding servers CS1, CS2 exposing the plaintexts $m_1 + u$, random number u and secret keys sk_1, sk_2.

From the hacker's perspective, the ciphertext of $E(m_1), E(m_2), E(m_1 + u), P(m_1 + u), E(m_2 \times (m_1 + u)), E(m_2 \times u), E(m_1 \times m_2)$ are accessible. He cannot learn any plaintext information from these ciphertexts due to the security of $(2, 2)$ TAHEG.

B. *Secure Comparison Function (`SCOM`)*

The cloud servers CS1 and CS2 execute secure comparison function (`SCOM`) with the inputs $E(m_1)$ and $E(m_2)$. The output of `SCOM` is either the encryption of 0 or the encryption of 1. If $m_1 > m_2$ the output is the pair $(E(0), E(1))$, otherwise the output is the pair $(E(1), E(0))$.

Function 2 Secure Comparison Function (SCOM)

```
Input: $E(m_1), E(m_2)$.
Output: $\mathbf{cr} = (E(0), E(1))$ if $m_1 > m_2$.
        $\mathbf{cr} = (E(1), E(0))$ if $m_1 \leq m_2$.
Step 1. CS1 executes
    Generate the random bit $b \in \{0, 1\}$
    If $b = 0$, then $E(d) \leftarrow \mathit{Add}(E(m_1), E(m_2)^{-1})$
    If $b = 1$, then $E(d) \leftarrow \mathit{Add}(E(m_2), E(m_1)^{-1})$
    $E(d \cdot v) \leftarrow \mathit{MulSca}(E(d), v)$
    $E(d \cdot v + e) \leftarrow \mathit{AddSca}(E(d \cdot v), e)$
    $P(d \cdot v + e) \leftarrow \mathit{PartDec}(E(d \cdot v + e), sk_1)$
    Send $P(d \cdot v + e)$ to CS2
Step 2. CS2 executes
    $(d \cdot v + e) \leftarrow \mathit{Dec}(P(d \cdot v + e), sk_2)$
    $E(0) \leftarrow \mathit{Enc}(0, pk_1, pk_2)$
    $E(1) \leftarrow \mathit{Enc}(1, pk_1, pk_2)$
    If $d \cdot v + e > 0$, then $\mathbf{cr} = (E(0), E(1))$
    If $d \cdot v + e \leq 0$, then $\mathbf{cr} = (E(1), E(0))$
    Send $\mathbf{cr}$ to CS1.
Step 3. CS1 executes
    If $b = 0$, then the output is $\mathbf{cr}$
    If $b = 1$, then the output is Rotate($\mathbf{cr}$)
```

In Step 1, CS1 takes $E(m_1)$ and $E(m_2)$ as input, and homomorphically computes $E(m_1 - m_2)$ or $E(m_2 - m_1)$ depending on the random bit $b = 0$ or $b = 1$. The sign of $d = m_1 - m_2$ or $d = m_2 - m_1$ gives whether $m_1 > m_2$ or $m_1 \leq m_2$. However, CS1 cannot discover the unencrypted value d since it has access to sk_1 but not sk_2. Both are needed for correct decryption of $E(d)$.

On the other hand, if CS1 sends $\mathit{PartDec}(E(d), sk_1)$ to CS2, then d can be computed by CS2, which is not acceptable by our security assumption that neither CS1, nor CS2 should discover the value of d. Instead, CS1 applies a **sign-preserving homomorphic transformation** to $E(d)$ and computes $E(d \cdot v + e)$, and sends $\mathit{PartDec}(E(d \cdot v + e), sk_1)$ to CS2, where v and e are random values such that $v > e$. Notice that since $v > e$, if d is negative then $d \cdot v + e$ is negative, and likewise, if d is positive then $d \cdot v + e$ is positive.

In Step 2, CS2 decrypts $P(d \cdot v + e)$, and obtains $d \cdot v + e$. Depending on the sign of $d \cdot v + e$, CS2 forms either the pair $\mathbf{cr} = (E(0), E(1))$ or the pair $\mathbf{cr} = (E(1), E(0))$, and sends $\mathbf{cr}$ to CS1.

In Step 3, CS1 rotates the pair $\mathbf{cr}$ if the random bit $b = 1$, otherwise it keeps it as received. Now both CS1 and CS2 have access to $\mathbf{cr}$ which is equal to $(E(0), E(1))$ if $m_1 > m_2$ or $(E(1), E(0))$ if $m_1 \leq m_2$. The exact value of $\mathbf{cr}$ can be discovered only if CS1 and CS2 collude and decrypt $\mathbf{cr}$ together. Our security assumption forbids that.

From the hacker's perspective, the ciphertext of $E(m_1), E(m_2), E(d), E(d \cdot v), E(d \cdot v + e), P(d \cdot v + e), \mathbf{cr}$ are accessible. He cannot learn any plaintext information from these ciphertexts due to the security of $(2, 2)$ TAHEG.

C. Secure Sign & Magnitude Acquisition Function (SSMA)

The cloud servers CS1 and CS2 execute the secure sign and magnitude acquisition function (SSMA) to obtain the sign and magnitude of m in the ciphertext $E(m)$. The positive sign is represented with the pair $\mathbf{cr} = (E(0), E(1))$, and the negative sign is represented with the pair $\mathbf{cr} = (E(1), E(0))$, which is computed by SCOM function. We denote the magnitude by $E(|m|)$ which can be computed using SCOM and SMUL functions as shown below.

Function 3 Secure Sign & Magnitude Acquisition (SSMA)

```
Input: $E(m)$
Output: $\mathbf{cr}$, $E(|m|)$
Step 1. CS1 and CS2
    $\mathbf{cr} \leftarrow \mathtt{SCOM}(E(m), E(0))$
Step 2. CS1 and CS2
    $A_1 \leftarrow \mathtt{SMUL}(E(m)^{-1}, \mathbf{cr}[1])$
    $A_2 \leftarrow \mathtt{SMUL}(E(m), \mathbf{cr}[2])$
    $E(|m|) \leftarrow \mathit{Add}(A_1, A_2)$
```

The sign of m in $E(m)$ is obtained using SCOM function in Step 1. The comparison result is the vector $(\mathbf{cr}[1], \mathbf{cr}[2])$. We now show that the magnitude $E(|m|)$ can be computed by the servers CS1 and CS2 by performing an inner-product operation with the vectors $(E(m)^{-1}, E(m))$ and $\mathbf{cr}$. Assume $m > 0$, then we have $\mathbf{cr} = (E(0), E(1))$ and $|m| = m$. The result of the inner-product operation would be

$$\begin{aligned} \mathtt{SMUL}(E(m)^{-1}, \mathbf{cr}[1]) &= E(-m \cdot 0) = E(0) \\ \mathtt{SMUL}(E(m), \mathbf{cr}[2]) &= E(m \cdot 1) = E(m) \\ \mathit{Add}(E(0), E(m)) &= E(m) \end{aligned}$$

as required. On the other hand, if $m < 0$, we have $\mathbf{cr} = (E(1), E(0))$ and $|m| = -m$. In this case, the result of the inner-product operation would be

$$\begin{aligned} \mathtt{SMUL}(E(m)^{-1}, \mathbf{cr}[1]) &= E(-m \cdot 1) = E(-m) \\ \mathtt{SMUL}(E(m), \mathbf{cr}[2]) &= E(m \cdot 0) = E(0) \\ \mathit{Add}(E(-m), E(0)) &= E(-m) \end{aligned}$$

In this function, the perspective of the hacker is consistent with SMUL and SCOM. $\mathcal{H}$ cannot learn any plaintext information from these ciphertexts due to the security of SMUL and SCOM functions.

VII. Experimental Evaluation

To evaluate the performance of SOCT, we implemented the toolkit using **gmpy2** version 2.1.5 in Python 3.11 on a PC with an Intel(R) Core(TM) i5-1155G7 @ 2.50GHz CPU and 8GB RAM. We also implemented the advanced works of SOCI [12] and SOCI$^+$ [16] in the same environment for a comparative efficiency analysis.

The parameter p in the cyclic subgroup is set to 2048 bits, following the parameter settings in SOCI and SOCI$^+$, where $N = 2048$ bits and the modulus $N^2 = 4096$ bits. This ensures that the ciphertexts have the same bit length across the SOCI, SOCI$^+$, and SOCT toolkits. The secret key in SOCI and SOCI$^+$ is 128 bits, while our secret key is 256 bits.

The TAHEG algorithm defines the plaintext space as $\mathbb{Z}_{2^\sigma}$ and prepares a look-up table $\{g^m : m\}$ for $m \in \mathbb{Z}_{2^{\sigma+1}}$ to reduce the DLOG(g^m) computation during the decryption

TABLE II
TIME COMPARISON OF BASIC CRYPTOGRAPHIC OPERATIONS OF SOCI, SOCI$^+$ AND OUR TOOLKIT WITH THE SAME CIPHERTEXT SIZE (2048 BITS)

HE algorithms	Toolkit	Enc	Dec	PDec(sk_1)[1]	PDec(sk_2)[1]	Addition	ScalarMul	Subtraction
PaillierTD	SOCI	9.250	8.986	17.868	17.646	0.039	0.075	0.076
FastPai	SOCI$^+$	**0.421**	2.050	0.690	11.087	0.006	0.068	0.067
BCP	SOCHE	4.750	2.420	2.420	2.430	2.430	4.410	-
BGV-Circuits	Pockit	147.0	51.0	-	-	0.120	-	-
MK-CKKS	-	-	-	-	-	10.0	120.0	-
TAHEG	SOCT	1.020	**0.744**	**0.374**	**0.375**	**0.004**	**0.012**	**0.031**

[1] PDec(sk_1) and PDec(sk_2) perform with sk_1, sk_2 respectively.
[2] The time unit is ms.

TABLE III
COMPARISON OF SECURE FUNCTIONS OF SOCI, SOCI$^+$ AND OUR TOOLKIT WITH THE SAME CIPHERTEXT SIZE (2048 BITS)

Schemes	Runtime (ms)			Communication cost (KB)		
	SMUL	SCOM	SSMA	SMUL	SCOM	SSMA
SOCI	107.788	54.208	175.386	2.498	1.498	3.996
SOCI$^+$	15.210	19.185	36.878	1.498	**1.497**	**2.997**
Pockit	29580	2030	-	-	-	-
SOCT	**4.256**	**4.868**	**14.545**	**1.000**	1.500	3.500

* The time unit is ms.

process ($\sigma = 16$). We repeat the operations 1000 times and compute the average as the final results. Additionally, we compare our experimental results with those from Pockit [14] and SOCHE [13].

We conducted experiments to evaluate the performance of basic operations in the HE algorithms: PaillierTD [18], FastPai [22], BGV-Circuits [14], BCP [23], MK-CKKS [24] and TAHEG. The SOCI and SOCI$^+$ implementations are based on the PaillierTD and FastPai, with SOCI$^+$ outperforming SOCI due to optimizations in the FastPai algorithm. The SOCHE utilizes the BCP [23] algorithm, as proposed by Bresson, Catalano, and Pointcheval. Meanwhile, Pockit [14] designs homomorphic circuits based on the BGV algorithm [25].

Table II presents a runtime comparison of the HE algorithms. The BCP is a partially homomorphic encryption (PHE) algorithm, where the ciphertext consists of a tuple containing two large integers modulo $N^2 = 4096$ bits. The larger ciphertext size results in lower efficiency. The multi-key CKKS (MK-CKKS) [24] algorithm supports homomorphic operations on ciphertexts encrypted with different keys. It is developed using the CKKS algorithm, and it optimizes the multiplication efficiency of [26]. Due to the conceptual differences between PHE and FHE algorithms, we only selected the homomorphic operation times of addition and multiplication for comparison.

The BGV-Circuits [14] construct homomorphic circuits, requiring integers to be encrypted as strings of bits, which reduces efficiency in both encryption and decryption. Our toolkit, based on the ElGamal cryptosystem, features a more efficient decryption function compared to the Paillier cryptosystem. In the partial decryption function of SOCI$^+$, the cloud servers' secret keys sk_1 and sk_2 are additively shared, with $sk_1 = 128$ bits, resulting in sk_2 having a bit length of 2495 bits. The larger operand leads to increased processing time, resulting in a significant performance difference between PDec(sk_1) and PDec(sk_2).

For basic homomorphic operations such as addition, scalar multiplication, and subtraction, the modulus in the ElGamal cryptosystem is 2048 bits, while the modulus in the Paillier cryptosystem is nearly 4096 bits. Consequently, operating on two 2048-bit numbers is faster than operating on a single 4096-bit number, making the TAHEG algorithm more efficient.

We conducted experiments to evaluate the consumption of secure computation functions SMUL, SCOM, and SSMA. As shown in Table III, the runtime measures the time consumed by two servers executing the function, while the communication cost calculates the size of data transmitted over the channel.

Pockit designs the SMUL and SCOM functions using homomorphic circuits that minimize interaction between servers. The SOCI$^+$ demonstrates significant efficiency improvements over SOCI, as FastPaiTD [16] is more efficient than PaillierTD.

Our SOCT exhibits greater efficiency in function steps compared to the functions in SOCI$^+$. Specifically, we reduce 3 encryption and 4 addition operations in the SMUL function, and 3 encryption and 1 addition operation in the SCOM functions. Consequently, our toolkit achieves the best performance in these experiments.

In SOCI$^+$, the SSMA function employs 1 SMUL and 1 SCOM, while our SOCT employs 2 SMUL and 1 SCOM, aligning with the experimental results. The communication cost for these comparison toolkits has reached the "KB" level. The modulus in SOCI and SOCI$^+$ is computed with N^2, where N is 2048 bits. Thus, the N^2 in our experiments may be less than 4096 bits (e.g., 4094 bits = 0.496 KB).

TABLE IV
USER RUNTIME IN PARAMETER GENERATION AND ENCRYPTION (SEC)

Schemes	Group or Key Generation	Thousands Enc	Millions Enc
SOCI	0.069	9.462	9488.892
SOCI$^+$	0.003	0.437	439.227
SOCT	74.839	1.029	1047.615

We evaluate the user overhead in our SOCT. Before executing the secure computation functions, users are required to generate cyclic subgroup parameters $G_q(p, q, g)$ and broadcast them to the cloud servers. Subsequently, CS1 and CS2 select their secret keys sk_1 and sk_2 and upload $h_1 = g^{sk_1}$ and $h_2 = g^{sk_2}$ to construct the public key $pk(p, q, g, h_1, h_2)$.

Users must encrypt their plaintext data and send the ciphertexts to CS1 and CS2. This means users must complete two tasks: generating the group parameters $G_q(p,q,g)$ and encrypting plaintexts before function execution, as shown in Table IV. The user's time consumption of SOCI and SOCI$^+$ includes key generation, key splitting, and data encryption. The key generation efficiency of Paillier cryptosystem is better than that of ElGamal cryptosystem. The user's time consumption in the encryption phase is mainly related to the consumption of Enc function. More importantly, the SOCI and SOCI$^+$ additionally require users to distribute the partial secret keys, which requires the secure channel before the outsourcing process. Our SOCT key pairs are generated by servers, and user only need to broadcast group parameters $G_q(p,q,g)$.

During our parameter generation process, the function $G_q(p,q,g) \leftarrow \mathit{ParSet}()$ randomly selects large primes p and q such that $q|(p-1)$, with $p \geq 2048$ bits and $q \geq 256$ bits. This can result in variable time costs. In practical applications, users can choose to utilize pre-computed $G_q(p,q,g)$ to reduce their overhead.

VIII. Related Work

Homomorphic algorithms with partial decryption can be applied in dual-server decryption architecture. The existing partial decryption is mainly achieved by threshold homomorphic encryption [19], [20] and multi-key encryption [27]. Threshold homomorphic encryption requires a trusted party to split the secret key and distribute it to the servers. Multi-key encryption supports each participant to generate their own key pairs, and then the ciphertext is encrypted by multiple public keys. However, additional operations are required to ensure homomorphic operations between ciphertexts encrypted with different keys. In these methods, a single untrusted party cannot decrypt the ciphertext with his secret key, and decryption of the ciphertext requires a partial decryption process with the input of multiple parties' secret key.

Existing multi-key homomorphic encryption is mainly implemented with FHE algorithms. Chen et al. [28] proposed a multi-key implementation of the TFHE [29] algorithm. Li et al. [30] propose a DGHV-type MKHE scheme. Che et al. [31] proposed RLWE-based multi-key FHE scheme. Kim et al. [24] proposed multi-key implementation of the BFV [32], [33] and CKKS [4].

In these schemes, additional operations are employed at the expense of efficiency to support homomorphic operations on ciphertexts encrypted with different keys. In MK-TFHE [28], the ciphertexts encrypted with different keys are stored in the extended ciphertext. Therefore, as the number of participants increases, the computational complexity increases by $\mathcal{O}(n^2)$. The decryption process requires all parties to partially decrypt the ciphertext. The DGHV-type MKHE [30] requires an extended key to convert ciphertexts encrypted with different keys to the same key to support homomorphic operations. In partial decryption process, each participant partially decrypts the ciphertext and uploads it to the cloud server, which completes the final decryption. Che et al. [31] expands the ciphertext of each participant into a $(k+1)\times(k+1)$ matrix, where k is the number of participants. Partial decryption process requires the secret keys of all participants. Kim et al. [24] introduces the homomorphic gadget decomposition to allow arithmetic operations on the decomposed vectors, which reduces the complexity of multi-key multiplication to $\mathcal{O}(n)$. Ciphertexts encrypted with different keys can be directly operated homomorphically, and the associated keys are recorded using a reference set. The partial decryption process requires secret key of the corresponding participants, which are recorded in the reference set.

Threshold homomorphic encryption is mainly implemented based on the PHE algorithm. Limited by the need for trusted parties to distribute secret keys, research on threshold homomorphic encryption has progressed slowly. The most representative ones are PaillierTD [17], [18] and threshold ElGamal [19], [20]. PaillierTD is developed from Paillier [34] algorithm. After generating the traditional Paillier key pair, the trusted party needs to split the secret key and distribute it to other parties. Although the PaillierTD implementation in [12], [16] claims to eliminate the need for trusted parties to generate and distribute keys compared to [35], [36], they instead leave this task to the user (data owner) of homomorphic outsourcing computation. This still requires users to consider secure channels to transfer secret keys before outsourcing operations begin. For the partial decryption process of PaillierTD, it relies on threshold setting and requires multiple secret keys to jointly decrypt the ciphertext.

In the above MKHE scheme, the ciphertext requires multiple related secret keys to be decrypted. Since the secret key is usually private data of the participants, in actual implementation, it is necessary for multiple participants to perform partial decryption separately, and then collect all the partial decryption results to obtain the plaintext. However, such a partial decryption setting allows the plaintext to be computed from multiple partially decrypted results without requiring any private data. This will cause the channel eavesdropper to easily obtain partial decryption results from multiple participants and decrypt the ciphertext.

In this paper, we employ the threshold homomorphic encryption of threshold additive ElGamal and let semi-honest parties generate key pairs. All ciphertexts are encrypted with the same public keys, making it support traditional ElGamal homomorphic operations. Each function in the partial decryption process requires a partial secret key to prevent an external hackers from obtaining plaintext through partial decryption results.

IX. Conclusion

In this paper, we introduced the Secure Outsourcing Computation Toolkit based on the threshold ElGamal cryptosystem. Our security assumptions are closer to real-world scenarios. We describe a $(2,2)$ threshold additively homomorphic ElGamal algorithm derived from the existing threshold ElGamal cryptosystem. Experiments show that the efficiency of our algorithms are better than existing toolkits [12], [13], [14], [16]. The SOCT supports the construction of computation functions for floating-point numbers [35] and rational numbers [36], and can be applied to more complex outsourced computing scenarios. A limitation of our SOCT architecture is that external hackers do not collude with internal servers, which will be studied in future work. The SOCT library is available for public access, and it can be downloaded from the GitHub site: https://github.com/KocLab2023/SOCT.

ACKNOWLEDGEMENTS

This research is supported in part by the Jiangsu Province 100 Foreign Experts Introduction Plan BX2022012. This research is also supported in part by TÜBİTAK Projects 2232-118C332 and 1001-121F348.

REFERENCES

[1] P. Li, J. Li, Z. Huang, C.-Z. Gao, W.-B. Chen, and K. Chen, "Privacy-preserving outsourced classification in cloud computing," Cluster Computing, vol. 21, no. 1, pp. 277–286, Mar. 2018.

[2] R. Xu, N. Baracaldo, and J. Joshi, "Privacy-Preserving Machine Learning: Methods, Challenges and Directions," Sep. 2021.

[3] C. Gentry, "Fully homomorphic encryption using ideal lattices," in Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing. Bethesda, USA: ACM, May 2009, pp. 169–178.

[4] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in Advances in Cryptology – ASIACRYPT 2017, T. Takagi and T. Peyrin, Eds. Cham, Switzerland: Springer, 2017, vol. 10624, pp. 409–437.

[5] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," Journal of Cryptology, vol. 33, no. 1, pp. 34–91, Jan. 2020.

[6] E. Lee, J.-W. Lee, J.-S. No, and Y.-S. Kim, "Minimax Approximation of Sign Function by Composite Polynomial for Homomorphic Comparison," IEEE Transactions on Dependable and Secure Computing, vol. 19, no. 6, pp. 3711–3727, Nov. 2022.

[7] D. Ong, R. Yap, and C. N. Yap, "Comparison of cuFHE vs TFHE on Arithmetic Circuit Computation," in 2023 8th International Conference on Computer and Communication Systems (ICCCS). Guangzhou, China: IEEE, Apr. 2023, pp. 612–616.

[8] C. Wang, J. Chen, X. Zhang, and H. Cheng, "An Efficient Fully Homomorphic Encryption Sorting Algorithm Using Addition Over TFHE," in 2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS). Nanjing, China: IEEE, Jan. 2023, pp. 226–233.

[9] Z. Shan, K. Ren, M. Blanton, and C. Wang, "Practical Secure Computation Outsourcing: A Survey," ACM Computing Surveys, vol. 51, no. 2, pp. 1–40, Mar. 2019.

[10] Ç. K. Koç, F. Özdemir, and Z. Ödemiş Özger, Partially Homomorphic Encryption. Cham, Switzerland: Springer, 2021.

[11] W. Wu, J. Liu, H. Wang, J. Hao, and M. Xian, "Secure and Efficient Outsourced k-Means Clustering using Fully Homomorphic Encryption With Ciphertext Packing Technique," IEEE Transactions on Knowledge and Data Engineering, vol. 33, no. 10, pp. 3424–3437, Oct. 2021.

[12] B. Zhao, J. Yuan, X. Liu, Y. Wu, H. H. Pang, and R. H. Deng, "SOCI: A Toolkit for Secure Outsourced Computation on Integers," IEEE Transactions on Information Forensics and Security, vol. 17, pp. 3637–3648, 2022.

[13] Q. Wang, D. Zhou, and Y. Li, "Secure Outsourced Calculations with Homomorphic Encryption," Advanced Computing: An International Journal, vol. 9, no. 6, pp. 01–14, Nov. 2018.

[14] X. Liu, R. H. Deng, K.-K. R. Choo, Y. Yang, and H. Pang, "Privacy-Preserving Outsourced Calculation Toolkit in the Cloud," IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 5, pp. 898–911, Sep. 2020.

[15] K. Zhao, X. A. Wang, B. Yang, Y. Tian, and J. Zhang, "A privacy preserving homomorphic computing toolkit for predictive computation," Information Processing & Management, vol. 59, no. 2, p. 102880, Mar. 2022.

[16] B. Zhao, W. Deng, X. Li, X. Liu, Q. Pei, and R. H. Deng, "SOCI+: An Enhanced Toolkit for Secure Outsourced Computation on Integers," IEEE Transactions on Information Forensics and Security, vol. 19, pp. 5607–5619, 2024.

[17] P.-A. Fouque, G. Poupard, and J. Stern, "Sharing Decryption in the Context of Voting or Lotteries," in Financial Cryptography, Y. Frankel, Ed. Berlin, Germany: Springer, 2001, vol. 1962, pp. 90–104.

[18] C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft, and A. A. Nicolosi, "Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting," Journal of Cryptology, vol. 32, no. 2, pp. 265–323, Apr. 2019.

[19] T. P. Pedersen, "A Threshold Cryptosystem without a Trusted Party," in Advances in Cryptology — EUROCRYPT '91, D. W. Davies, Ed. Berlin, Germany: Springer, 1991, vol. 547, pp. 522–526.

[20] Y. Desmedt, "Threshold cryptosystems," in Advances in Cryptology — AUSCRYPT '92, G. Goos, J. Hartmanis, J. Seberry, and Y. Zheng, Eds. Berlin, Germany: Springer, 1993, vol. 718, pp. 1–14.

[21] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," IEEE Transactions on Information Theory, vol. 31, no. 4, pp. 469–472, Jul. 1985.

[22] H. Ma, S. Han, and H. Lei, "Optimized Paillier's Cryptosystem with Fast Encryption and Decryption," in Annual Computer Security Applications Conference. Virtual Event, USA: ACM, Dec. 2021, pp. 106–118.

[23] E. Bresson, D. Catalano, and D. Pointcheval, "A Simple Public-Key Cryptosystem with a Double Trapdoor Decryption Mechanism and Its Applications," in Advances in Cryptology - ASIACRYPT 2003, G. Goos, J. Hartmanis, J. Van Leeuwen, and C.-S. Laih, Eds. Berlin, Germany: Springer, 2003, vol. 2894, pp. 37–54.

[24] T. Kim, H. Kwak, D. Lee, J. Seo, and Y. Song, "Asymptotically Faster Multi-Key Homomorphic Encryption from Homomorphic Gadget Decomposition," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. Copenhagen, Denmark: ACM, Nov. 2023, pp. 726–740.

[25] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. Cambridge, USA: ACM, Jan. 2012, pp. 309–325.

[26] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London, UK: ACM, Nov. 2019, pp. 395–412.

[27] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption," in Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing. New York, USA: ACM, May 2012, pp. 1219–1234.

[28] H. Chen, I. Chillotti, and Y. Song, "Multi-Key Homomorphic Encryption from TFHE," in Advances in Cryptology – ASIACRYPT 2019, S. D. Galbraith and S. Moriai, Eds. Cham, Switzerland: Springer, 2019, vol. 11922, pp. 446–472.

[29] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds," in Advances in Cryptology – ASIACRYPT 2016, J. H. Cheon and T. Takagi, Eds. Berlin, Germany: Springer, 2016, vol. 10031, pp. 3–33.

[30] X. Li, H. Li, J. Gao, and R. Wang, "Privacy preserving via multi-key homomorphic encryption in cloud computing," Journal of Information Security and Applications, vol. 74, p. 103463, May 2023.

[31] X. Che, L. Liu, B. Wang, Y. Han, X. A. Wang, X. Yang, and T. Zhou, "Multi-key homomorphic encryption with tightened RGSW ciphertexts without relinearization for ciphertexts product," Journal of King Saud University - Computer and Information Sciences, vol. 35, no. 10, p. 101794, Dec. 2023.

[32] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," in Advances in Cryptology – CRYPTO 2012, R. Safavi-Naini and R. Canetti, Eds. Berlin, Germany: Springer, 2012, vol. 7417, pp. 868–886.

[33] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012. [Online]. Available: https://eprint.iacr.org/2012/144

[34] P. Paillier and D. Pointcheval, "Efficient Public-Key Cryptosystems Provably Secure Against Active Adversaries," in Advances in Cryptology - ASIACRYPT'99, G. Goos, J. Hartmanis, J. Van Leeuwen, K.-Y. Lam, E. Okamoto, and C. Xing, Eds. Berlin, Germany: Springer, 1999, vol. 1716, pp. 165–179.

[35] X. Liu, R. H. Deng, W. Ding, R. Lu, and B. Qin, "Privacy-Preserving Outsourced Calculation on Floating Point Numbers," IEEE Transactions on Information Forensics and Security, vol. 11, no. 11, pp. 2513–2527, Nov. 2016.

[36] X. Liu, K.-K. R. Choo, R. H. Deng, R. Lu, and J. Weng, "Efficient and Privacy-Preserving Outsourced Calculation of Rational Numbers," IEEE

Transactions on Dependable and Secure Computing, vol. 15, no. 1, pp. 27–39, Jan. 2018.