HEAD: an FHE-based Privacy-preserving Cloud Computing Protocol with Compact Storage and Efficient Computation

Lijing Zhou†, Ziyu Wang†,Hongrui Cui‡, Xiao Zhang†, Xianggui Wang†, Yu Yu‡
†Huawei Technology, Shanghai, China, {zhoulijing,wangziyu13,zhangxiao81,wangxianggui1}@huawei.com,
‡Shanghai Jiao Tong University, Shanghai, China, {rickfreeman,yuyu}@cs.sjtu.edu.cn

***Abstract*—Fully homomorphic encryption (FHE) provides a natural solution for privacy-preserving cloud computing, but a straightforward FHE protocol may suffer from high computational overhead and a large ciphertext expansion rate, especially for computation-intensive tasks over large data, which are the main obstacles toward practical privacy-preserving cloud computing. In this paper, we present HEAD, a generic privacy-preserving cloud computing protocol that can be based on most mainstream (typically a BGV or GSW style scheme) FHE schemes with more compact storage and less computational costs than the straightforward FHE counterpart. In particular, our protocol enjoys a ciphertext/plaintext expansion rate of 1 (i.e., no expansion) in a cloud computing server, instead of a factor of hundreds of thousands. This is achieved by means of "pseudorandomly masked" ciphertexts, and the efficient transformations of them into FHE ciphertexts to facilitate privacy-preserving cloud computing. Depending on the underlying FHE in use, our HEAD protocol can be instantiated with the three masking techniques, namely modulo-subtraction-masking, modulo-division-masking, and XOR-masking, to support the decimal integer, real, or binary messages. Thanks to these masking techniques, various homomorphic computation tasks are made more efficient and less prone to noise accumulation. Furthermore, our multi-input masking and unmasking operations are more flexible than the FHE SIMD-batching, by supporting an on-demand configuration of FHE during each cloud computing request.**

We evaluate the performance of HEAD protocols on BFV, BGV, CKKS, and FHEW schemes based on the PALISADE and SEAL libraries, which confirms the theoretical analysis of the storage savings, the reduction in terms of computational complexity and noise accumulation. For example, in the BFV computation optimization, the sum or product of eight ciphertexts overhead is reduced from 336.3 ms to 6.3 ms, or from 1219.4 ms to 9.5 ms, respectively. We also embed HEAD into a mainstream database, PostgreSQL, in a client-server cloud storage and computing style. Compared with a straightforward FHE protocol, our experiments show that HEAD does not incur ciphertext expansion, and exhibits at least an order of magnitude saving in computing time at the server side for various tasks (on a hundred ciphertexts), by paying a reasonable price in client preprocessing time and communication. Our storage advantage not only gets around the database storage limitation but also reduces the I/O overhead.

I. Introduction

With the broad and rapidly growing need for big-data driven applications, the privacy protection of personal data is increasingly receiving social awareness and legal enforcement (e.g., see the policies and regulations in GDPR [48]). As an advanced form of encryption, fully homomorphic encryption (FHE) allows anyone not in possession of the decryption key to perform computations (typically by evaluating a Boolean/arithmetic circuit) on ciphertexts, and produces the desired computation result in an encrypted form. In other words, FHE can protect the user data without interrupting the original data-processing flow at the server side, offering a theoretically seamless integration with the current privacy-preserving cloud computing applications. Despite the promising features, FHE is not readily used in many real-life applications due to its heavy performance overhead in both storage and computation.

The storage issue lies in the high ciphertext expansion rate (CER) of FHE, which is especially relevant in the context of big-data storage where the plaintext data is already gigantic and a client has to outsource the data to a cloud computing server. In a typical setting[1], the CER of the CKKS scheme ranges from dozens to hundreds of thousands, so that 1TiB of real numbers costs a storage space of hundreds of PiB for CKKS ciphertexts. Even batching these messages, the ciphertext size is still dozens of TiB. For 1 TiB of binary numbers, an FHEW ciphertext spends the space of dozens of PiB.[2]

Another FHE disadvantage is the high computational cost which would be amplified in privacy-preserving cloud computing applications since the operations applied on plaintexts are typically complicated in the first place. The huge FHE ciphertexts also affect the computation since the I/O overhead cannot be omitted. For evaluating a circuit, an FHE computation is typically six orders of magnitude slower than the one in plaintext from our experiment experience.

Motivated by the performance bottleneck of applications with FHE, we aim to address the two problems by designing a protocol that combines the benefits of compact storage and homomorphic encryption.

A. Our Contribution

We present HEAD[3], an FHE-based privacy-preserving cloud computing protocol, which aims to address/mitigate the ciphertext expansion and computational overheads (due to the use of FHE) in a cloud computing server. In particular, instead of storing the FHE ciphertext directly at the server side, we instead store pseudo-randomly masked plaintexts, and then we can flexibly transform any part of these masked plaintexts into FHE ciphertexts at the query phase, via paying reasonable costs in computation and communication on demand.

[1]The ciphertext size comes from the PALISADE library experiments, in which the CKKS setting supports 3 computation levels, 128-bit security, and the ciphertext dimension is 16,384, and the FHEW setting is 128-bit security.

[2]Batching is a useful technique to mitigate these FHE problems to some extent but cannot solve them, especially not for terabyte data. Batching is not flexible enough when messages come at different times. Moreover, a GSW-style FHE scheme like FHEW does not support batching.

[3]The name is a portmanteau of HE and PAD.

Table I: HEAD contributions. (Ciphertext expansion rate abbreviated as CER)

HEAD protocols	**ModSub masking**	**ModDiv masking**	**XOR masking**	**XOR masking**
Input type	Decimal integer	Decimal integer	Real	Binary
Scheme	BFV/BGV	BFV/BGV	CKKS	FHEW
Storage optimization	CER = 1	CER = 1	CER = 1	CER = 1
Computation optimization	Many Adds or ScalarMult $\rightarrow$ one pt + ct	Many Mults $\rightarrow$ one ScalarMult	N.A.	Many XORs/XNORs $\rightarrow$ one NOT

Additionally, by utilizing the homomorphism at the masking level we achieve a computational efficiency boost for various computational operations on the ciphertexts.

Our protocol is more generic and supports the transformation from many typical mask-with-pseudorandom encryptions to mainstream FHE (e.g., BGV or GSW-style) ciphertexts with IND-CPA security (as expected from FHEs). The practical experiments, i.e., by integrating HEAD in PostgreSQL, demonstrate the advantage of these transformations. Concretely, we summarize the protocol properties as follows.

- **Generic.** The protocol can be instantiated with many FHE schemes. We apply the corresponding suitable masking methods leading to the modulo-subtraction-masking (ModSub), modulo-division-masking (ModDiv), and XOR-masking protocols, supporting different data types (integer, real or binary) of an underlying FHE.

- **Compact.** All FHE instantiations of the protocol achieve a ciphertext expansion rate of 1 for a long-term outsourced storage in a cloud computing server.

- **Efficient.** Our protocol optimizes various outsourced computing tasks (as summarised in Table I). These optimizations reduce a large amount of time and/or noise. Reducing noise for an FHE scheme eliminates the need for a heavy bootstrapping, potentially reducing computation time, too.
  - The ModSub protocol optimizes one/more ciphertext-wise additions (Adds), or a scalar multiplication (ScalarMult) to only a single plaintext-ciphertext addition, i.e., many Adds or ScalarMult $\rightarrow$ one pt + ct.
  - The ModDiv protocol optimizes one/more ciphertext-wise multiplications (Mults) to only a single scalar multiplication, i.e., many Mults $\rightarrow$ one ScalarMult.
  - The XOR-masking protocol optimizes one/more ciphertext-wise XOR and XNOR operations (XORs/XNORs) to a single plaintext-ciphertext XOR, which is further optimized to a single ciphertext-wise NOT, i.e., many XORs/XNORs $\rightarrow$ one NOT.

- **Flexible.** Our flexibility lies in two aspects.
  - It is flexible to configure the FHE encryption parameters of a transformed FHE ciphertext according to (and just before) a cloud computing query. That is, the parameter choices do not need to be decided prior to the masking operation, i.e., the storage phase.
  - When a cloud computing task involves several stored masked messages, the retrieval flexibly selects masked messages and transforms them into a single batched FHE ciphertext (if batching is supported by the underlying FHE scheme) for subsequent evaluations. This is quite useful when the data amount is huge, since the unrelated data does not need to be read.

Implementation. We implement our protocols with the BFV, BGV, CKKS, and FHEW FHE schemes in PALISADE and SEAL libraries and compare them with a straightforward FHE protocol (without our techniques) as a baseline. We show that a masked ciphertext with the expansion rate of 1 occupies much less space compared with a straightforward FHE ciphertext, and the computation optimizations are significant. For example, in a typical encryption parameter setting, we decrease the expansion rate of a BFV ciphertext from several hundreds of thousands to 1, and ModSub or ModDiv optimizes the sum or product of eight BFV ciphertexts from 336.3 ms to 6.3 ms, or from 1219.4 ms to 9.5 ms, respectively.

Database Integration. We integrate HEAD in a PostgreSQL database to evaluate the trade-offs and advantages of the HEAD protocols in a practical privacy-preserving cloud computing case. The 1,500 masked ciphertexts could be easily stored in a row of a database table, while the corresponding FHE ciphertexts should be split into several rows since one row could not bear the huge ciphertext size. Moreover, the I/O overhead saving is also exhibited in our texts. We evaluate the sum, average, inner product, and variance tasks for 100 numbers. By paying a reasonable client pre-processing time and communication overhead, all our tasks could be an order of magnitude faster than the ones in a straightforward FHE protocol at a cloud computing server, respectively.

We note that the above performance advantages are achieved at the cost of deviating from the straightforward FHE protocol. In particular, to transform the pseudorandomly masked messages to FHE encrypted ones, the client needs to send FHE encryption of the corresponding masks at the query phase, which is not required by a straightforward FHE protocol. In essence, we trade the query phase communication complexity for the ciphertext expansion rate at the server side. Nevertheless in the setting of cloud computing of big data storage, the trade-off is worthwhile.

B. Related Works

1) Homomorphic Encryption: An HE is called additively (resp., multiplicatively or fully) homomorphic if addition (resp., multiplication or both) is supported over ciphertexts. For instance, Paillier [41] is an additive HE that supports only homomorphic additions and scalar multiplications (but not multiplications between ciphertexts). A leveled FHE is a slightly weaker form of FHE that supports arbitrary computations up to a certain depth that must be specified in advance for parameter configurations.

In a leveled FHE scheme, a ciphertext would accumulate some noise during homomorphic computations, and may not be correctly decrypted when the computations are beyond the preset level, since the introduced noise is too large and conceals the plaintext. In order to avoid a decrypt-then-re-encrypt naive noise reduction, a mechanism named bootstrapping is proposed, which generally evaluates a decryption circuit homomorphically [26], [28]. However, the bootstrapping operation incurs considerable overhead. Currently, a learning-with-errors (LWE) based leveled FHE with a relatively efficient bootstrapping is the mainstream to achieve FHE when the system parameters are decoupled with the functions to be evaluated [35]. The rest of the paper views a leveled FHE scheme with bootstrapping as an FHE scheme.

Except for the FHE schemes based on the approximate greatest common division problem [15], the mainstream FHE construction relies on the LWE assumption. The FHE standardization procedure [7] and outstanding open-source libraries [3], [42], [34], [18], [1], [4], [2] mainly focus on the LWE path. Different LWE-based schemes can be categorized according to the supported input message types into two main classes, BGV style and GSW style. A BGV-style scheme, e.g., BGV [11] or BFV [24], [12], aims to handle the integer inputs (word-wise). CKKS [16] further extends an FHE to a real number input message. The GSW style copes with bit-wise binary messages, such as the GSW [32], FHEW [23], and TFHE [17] schemes. The homomorphic computation results in these schemes (except for CKKS) are precise, while CKKS may approximately evaluate a computation at a preset precision.

Although these mainstream schemes [11], [24], [16], [23] are implemented in open-source libraries [42], [3], [18], an FHE scheme is usually regarded as impractical in storage and computation. The storage problem lies in the notable FHE ciphertext expansion. An LWE-based FHE ciphertext, corresponding to an integer input, occupies several hundreds of kilobytes storage, since the hardness of the LWE problem requires a large enough lattice dimension. Hence, the ciphertext expansion rate, i.e., the ratio of the FHE ciphertext size to the input message size, is quite large in an FHE scheme, which vastly limits an FHE application in terms of communication and storage. Another focus for an FHE scheme is the inefficient computation concern. For a software implementation, a ciphertext-wise multiplication is usually the slowest operation in an integer FHE scheme, taking more than 100 ms. For a binary FHE scheme, the slowest computation unit is a ciphertext-wise XOR/XNOR, taking more than 1 second.

2) Storage optimization: Packing [10] is a useful compression technique when the FHE encryption parameters satisfy some conditions. A bunch of input messages could be embedded in one ciphertext, rendering a relatively small ciphertext expansion rate via amortization. The state-of-the-art packing works achieve the minimum ciphertext expansion rate of around a dozen or a hundred for a BGV-style or a GSW-style FHE scheme [10], [38], respectively, which is not compact.

Gentry and Halevi [27] and Brakerski et al. [9] innovate some mechanisms to compress many GSW-style ciphertexts (occupying a large storage space) to a small-size ciphertext. The resulting ciphertext expansion rate for a compressed ciphertext is almost 1, at a price of losing some homomorphic functionalities in the compressed ciphertext.

The key-switching and modulus-switching are other two techniques related to decreasing the ciphertext size [31]. After multiplying two FHE ciphertexts, the dimension of the resulting ciphertext is increased, which requires a re-linearization operation, i.e., a key-switching operation [29], to recover the ciphertext dimension. However, the size of a key-switched ciphertext does not decrease compared with the original size. Modulus-switching, proposed by Brakerski and Vaikuntanathan [12], is a noise management technique intentionally. Generally, when the noise of a ciphertext is reduced via modulus-switching, the ciphertext size gets smaller. However, the switched ciphertext still keeps a large expansion rate and the computation depth it could support is decreased.

Chen et al. [14] present a unidirectional transformation from an LWE-based FHE ciphertext to a Ring LWE-based one, which supports packing and has an amortized ciphertext size. Still, the ciphertext expansion rate remains large.

3) Computation optimization: Based on packing, the work from Smart and Vercauteren [46] enhances the evaluation strength for one packed ciphertext, i.e., the SIMD (single instruction, multiple data) batching. Besides storage optimization, the evaluation on a batched FHE ciphertext is equally acting on each element in that batch. A heavy computation burden is amortized to each message, which is an acceleration in some sense. A batched FHE scheme already offers both the storage and computation optimization in a practical application like password-leakage searching [39]. However, some FHE schemes, such as TFHE [17], only support compress-used packing instead of SIMD-batching. Moreover, the packing or SIMD-batching solution incurs a trade-off between amortized storage and computation optimization.

Lu et al. [40] apply the batching technique to accelerate statistical analysis on encrypted categorical, ordinal, and numerical data as a privacy-preserving cloud computing case study. They propose a novel batch greater-than primitive and a layout consistent matrix primitive, and their protocol can evaluate statistical computation, e.g., histogram, on several thousands of data in several minutes.

Aubry et al. [8] propose a circuit re-writer to highly decrease the depth of a Boolean/arithmetic circuit, which optimizes the computation, since the computational overhead of an FHE scheme highly relies on the circuit depth.

Another technique for computation optimization of an FHE scheme is based on a hardware accelerator. There are a line of FPGA-based works [21], [20], [22], [45], [47] and some ASIC-based works [36], [44]. Especially, the hardware accelerator proposed by Feldmann et al. [25] outperforms state-of-the-art software implementations by up to 17,000 times.

4) HE transformation: Another way of FHE optimization is to process an HE ciphertext transformed from another "ciphertext", e.g., of secret sharing [43], [5] and AES [30], [31], to avoid heavy (fully) homomorphic ciphertext operations and obtain some storage advantages.

Rane et al. [43] and Akavia et al. [6], [5] [4] optimize the FHE storage in a one-client-two-server outsourcing scenario

[4]We were not aware of the compact storage solution for FHE via secret sharing until the event [6]. In some sense, our work could be seen as a variant of [5] with only one single cloud computing server.

using additive secret sharing, under the assumption that these two servers do not collude. In this case, a client first shares its confidential data (a secret) between the two servers. When the client makes a computation request, one server homomorphically encrypts its shares and sends the HE ciphertext to the other server. Then, the other server recovers the ciphertext of the secret to process the subsequent evaluation.

Gentry et al. [30], [31] transform an AES-128 ciphertext to a BGV FHE ciphertext, which could be utilized in a one-client-one-server outsourcing scenario. The AES ciphertext enjoys a compact storage, i.e., a ciphertext expansion rate of almost 1 when the input message size is a multiple of 128 bits. However, even with hardware acceleration optimizations, transforming a 120-block AES-128 ciphertext costs more than four minutes. In addition, the transformed FHE ciphertext corresponds to the elements in GF(2^d) which may not be an efficient representation for many practical applications.

Cho et al. [19] proposes a Real-to-Finite-field framework to avoid the high CER for the CKKS scheme when encrypting real numbers. They use the FV FHE scheme to encrypt the plain modulus and a stream cipher scheme is used to mask the original message. They accomplish a CER of 1.23 $\sim$ 1.54 (larger than ours) and the encryption throughput is optimized a lot at both client and server sides when the batch size is quite large (e.g., 65,535). However, to recover the target ciphertext, two times of transformation is involved with a heavy bootstrapping, which is not as efficient as our protocol.

Enlightened by the previous works [30], [31], we pursue a different way to optimize FHE storage and computation, without relying on non-colluding cloud computing servers or introducing high overheads of conversion. Moreover, the transformation is generic enough to support as many FHE schemes and input data types as possible.

Paper organization. Section II introduces the notations, pseudorandom function (PRF) and FHE definitions, and a straightforward FHE cloud computing scenario. Section III introduces our generic HEAD protocol while Section IV instantiates it by three different masking techniques, i.e., modulo-subtraction, modulo-division, and XOR. We show that these instantiated protocols not only achieve a ciphertext expansion rate of 1, but also optimize various computations in different FHE schemes. Section V evaluates the performance of HEAD protocols based on the BFV, BGV, CKKS, and FHEW FHE schemes. Section VI shows some practical cloud computing experiments when integrating HEAD protocols in a PostgreSQL database. Finally, Section VII concludes this paper.

II. Preliminaries

A. Notation

$x \leftarrow_{\$} \mathcal{X}$ or $x \leftarrow \mathcal{X}$ refers to drawing an element x from the space $\mathcal{X}$ uniformly at random, or according to the distribution of $\mathcal{X}$, respectively. $\{x_i\}$ represents a set having I messages, and each message in this set x_i has an index i. The m-bit message x could be represented in binary as $x^0, \cdots, x^{m-1}$. $\widetilde{x}$ represents the masked message of x, using a pseudorandom r. $\oplus$ or $\odot$ is an XOR or XNOR operation, respectively.

In an FHE scheme, let sk, pk, $\mathsf{ct}(\cdot)$, $\mathcal{M}$, and $\mathcal{CT}$ denote a secret key, a public key, and a ciphertext, a message space, and a ciphertext space, respectively. Γ is a circuit to be evaluated in an FHE-based privacy-preserving cloud computing. λ is the security parameter. n, t, and Q are FHE encryption parameters for the ciphertext ring dimension, the plaintext modulus, and the ciphertext modulus, respectively.

B. Pseudorandom Function

Let PRF: $\mathcal{K} \times \mathcal{X} \rightarrow \mathcal{Y}$ be an efficiently computable function. PRF is a pseudorandom function if for any probabilistic polynomial time (PPT) adversary $\mathcal{A}$ the advantage $\mathrm{Adv}^{\mathrm{PRF}}_{\mathcal{A}}(\lambda)$

$$\left| \Pr[b = 1 | \mathsf{key} \leftarrow_{\$} \mathcal{K}; b \leftarrow \mathcal{A}^{\mathrm{PRF}(key,\cdot)}] - \Pr[b = 1 | b \leftarrow \mathcal{A}^{\mathrm{RF}(\cdot)}] \right|$$

is negligible (negl), where $\mathsf{RF}(\cdot)$ is a random function RF : $\mathcal{X} \rightarrow \mathcal{Y}$. We use $r_i \leftarrow \mathsf{PRF}(\mathsf{key}, i)$ to denote the instantiation of the PRF function, which takes inputs on $\mathsf{key} \in \mathcal{K}$ and an index $i \in \mathcal{X}$, and outputs a pseudorandom number $r_i \in \mathcal{Y}$. We denote $\mathcal{Y}$ as same as the message space $\mathcal{M}$.

C. Fully Homomorphic Encryption

We summarize the key components of an FHE scheme including four PPT algorithms, i.e., FHE = (KeyGen, $\mathsf{Encrypt}$, $\mathsf{Decrypt}$, Eval) in a high level.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$: This algorithm inputs a security parameter λ and outputs a secret key sk and a public key pk. pk implicitly refers to a public encryption key, a public relinear key[5], and a public Galois key for rotating a SIMD-batched ciphertext [37].

- $\mathsf{ct}(x) \leftarrow \mathsf{Encrypt}(\mathsf{pk}/\mathsf{sk}, x)$ and $x \leftarrow \mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}(x))$: The encryption and decryption algorithms convert a message $x \in \mathcal{M}$ to a ciphertext $\mathsf{ct}(x) \in \mathcal{CT}$ and vice versa. Note that an FHE scheme may support a symmetric or an asymmetric encryption mode, so that the encryption algorithm would require a secret key or a public key, respectively.

- $\mathsf{ct}(\Gamma(x)) \leftarrow \mathsf{Eval}(\mathsf{pk}, \Gamma, \mathsf{ct}(x))$: A circuit Γ to be evaluated in an FHE scheme generally applies to the input $\mathsf{ct}(x)$ rendering $\mathsf{ct}(\Gamma(x))$.

Definition 1 (Correctness): A fully homormophic Encryption scheme FHE is correct if for all $x \in \mathcal{M}$, and $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^{\lambda})$, the following probability is overwhelming (close to 1 except for negligible error):

$$\Pr\left[\begin{array}{l} \mathsf{Decrypt}\big(\mathsf{sk}, \mathsf{Encrypt}(\mathsf{pk}/\mathsf{sk}, x)\big) = M; \\ \mathsf{Decrypt}\big(\mathsf{sk}, \mathsf{Eval}(\mathsf{pk}, \Gamma, \mathsf{Encrypt}(\mathsf{pk}/\mathsf{sk}, x))\big) = \Gamma(x) \end{array}\right]$$

Informally, an encryption scheme is indistinguishable secure under chosen plaintext attacks (IND-CPA, Appendix A) if no substantial information about the plaintext can be feasibly extracted from the ciphertext. In other words, for a given ciphertext, the distribution of the corresponding plaintext is still random in the view of any PPT adversary.

[5]After a ciphertext multiplication, the dimension of the ciphertext may increase. A relinearization is to recover the initial ciphertext dimension.

1) Encoding: Roughly speaking, the encryption algorithm of an LWE-based FHE scheme implicitly consists of an encoding process to convert a message to an element in a special plaintext space, and the decryption also includes a reversed decoding process. In the BFV scheme, a plaintext is defined by an element in a ring $R_t = \mathbb{Z}_t[x]/(x^n + 1)$, whose element is a polynomial having a degree smaller than n and the coefficients come from the field $\mathbb{Z}_t$. t is named as the plaintext modulus, which affects the correctness of an FHE scheme. A bigger t value may allow a larger message space to keep the computation correctness without data-type overflow. However, a too-big t value may introduce too much noise during the homomorphic computations. CKKS additionally requires an integer scale factor Δ to convert real inputs to integers. In the following paper, we do not specify the encoding algorithm and regard the encoding and decoding processes as the parts of the encryption and decryption algorithms, respectively.

2) Encryption parameters: A BGV-style FHE ciphertext typically contains two polynomials defined in a ring $R_Q = \mathbb{Z}_Q[x]/(x^n + 1)$, in which n is the ring dimension and Q is the ciphertext modulo. Selecting a big enough n value would guarantee the security level specified by the security parameter λ [7]. A small Q value cannot support too many computation levels in homomorphic encryption, while a too big ciphertext modulo Q value would introduce more noise. A too big n or Q value would add the storage and computational overhead. Hence, the encryption parameters should be carefully selected to fit a computation task without losing neither security, efficiency, nor correctness.

3) SIMD batching: An FHE encryption scheme has a large computational overhead and a large ciphertext storage size. SIMD is a typical trick used in parallel computing, which is widely used in FHE schemes [10], [46]. When N messages are embedded into a batched ciphertext, the computational overhead of an encryption/decryption or an evaluation, and the storage usage of the ciphertext, are all divided by N, i.e., an amortization operation. Hence, batching could be regarded as a computation acceleration or a ciphertext compression operation in some sense. Sect. V would compare the storage difference between batching and unbatching.

Roughly speaking, when the plaintext modulus t satisfies $t \equiv 1 \mod 2n$, BFV and BGV support an $N = n$ batching at maximum [37]. In CKKS, the maximum batching size is $\frac{n}{2}$. Popular FHE libraries, e.g., PALISADE [42] and SEAL [3], already widely follow this batching technique and highly recommend that the developers should batch their messages as much as possible, although batching does not fit all FHE applications as we discuss in Sect. I.

D. A Straightforward FHE Privacy-preserving Cloud Computing Protocol

Fig. 1 introduces a straightforward FHE protocol used in a privacy-preserving cloud computing scenario. At first, a client sets an FHE system by generating secret and public keys according to a set of encryption parameters. Then, the client encrypts its confidential messages by the FHE scheme, and sends several ciphertexts to a cloud computing server, together with the generated public key. The server stores the ciphertexts on behalf of the client. We name this the storage phase. When the client requests the server to conduct some computations on its encrypted data, the client proposes a computation task (generally a circuit Γ), i.e., a query phase. Next, the server computes this task without knowing the message. Finally, the server returns the target ciphertext to the client side, and the client decrypts it to obtain the results.

However, directly using an FHE scheme in this scenario faces the storage and computation obstacles. The FHE ciphertext expansion rate is significant. Even when batching is acceptable, the expansion rate of a ciphertext having the maximum batch size is still more than one.

In addition, the number of ciphertext-wise operations restrains this application from the computation time and introduced noise. Usually, the computation time for a ciphertext-wise multiplication is much higher than an addition or a scalar multiplication. A ciphertext-wise multiplication also introduces a large amount of noise compared with other computations. An addition or a scalar multiplication accumulates much less noise. Since bootstrapping is a heavy computational burden to eliminate noise, a noise optimization is implicit a computation time optimization.

III. The HEAD Protocol

In this section, we propose our generic HEAD protocol and delay to prove its security in a simulation-based model in Appendix A.

Definition 2: The generic HEAD protocol is depicted in Fig.2, consisting of six PPT algorithms, namely $\mathsf{HEAD} = (\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Eval}, \mathsf{NewEval}, \mathsf{Trans})$.

- The KeyGen, $\mathsf{Encrypt}$, $\mathsf{Decrypt}$, and Eval algorithms follow the original ones in an FHE scheme as we introduced in Sect. II. In addition, $\mathsf{ct}(r_i)$ is an encryption of a random mask r_i, i.e., $\forall i \in \mathcal{I}, r_i \leftarrow_{\$} \mathcal{M}, \mathsf{ct}(r_i) \leftarrow \mathsf{Encrypt}(\mathsf{pk}, r_i)$ for an index set I.
- The transformation algorithm $\mathsf{Trans}(x, r)$ returns a masked ciphertext $\widetilde{x}$ upon input of a message $x \in \mathcal{M}$ and a randomness $r \in \mathcal{M}$. $\{\widetilde{x}_i\} = \{\widetilde{x}_i \mid \widetilde{x}_i \leftarrow \mathsf{Trans}(x_i, r_i)\}$.
- The new evaluation algorithm defined in our HEAD protocols, $\mathsf{NewEval}(\mathsf{pk}, \Gamma, \mathsf{ct}(r), \widetilde{x})$, outputs a ciphertext $\mathsf{ct}(\Gamma(x)) \in \mathcal{CT}$ on input a public key pk, a circuit Γ, an FHE ciphertext $\mathsf{ct}(r)$ and a masked ciphertext $\widetilde{x}$.

Definition 3 (Correctness): The generic HEAD protocol is correct if $\forall x, r \in \mathcal{M}$ and $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$, the following probability is overwhelming (close to 1 except for negligible error):

$$\Pr\left[\begin{array}{l} \mathsf{Decrypt}\Big(\mathsf{sk}, \mathsf{Encrypt}\big(\mathsf{pk}, x\big)\Big) = x \wedge \\ \mathsf{Decrypt}\Big(\mathsf{sk}, \mathsf{Eval}\big(\mathsf{pk}, \Gamma, \mathsf{Encrypt}(\mathsf{pk}, x)\big)\Big) = \\ \mathsf{Decrypt}\Big(\mathsf{sk}, \mathsf{NewEval}\big(\mathsf{pk}, \Gamma, \mathsf{Encrypt}(\mathsf{pk}, r), \\ \mathsf{Trans}(x, r)\big)\Big) = \Gamma(x_i) \end{array}\right]$$

It is obvious that our generic HEAD protocol (Fig. 2) does not change the process of a straightforward FHE cloud computing protocol (Fig. 1), no matter whether an FHE scheme supports SIMD-batching or not. The masked ciphertexts spend

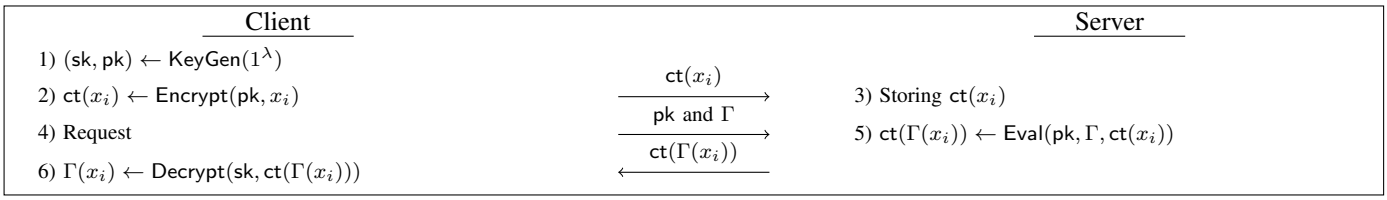

Figure 1: A straightforward FHE privacy-preserving cloud computing protocol

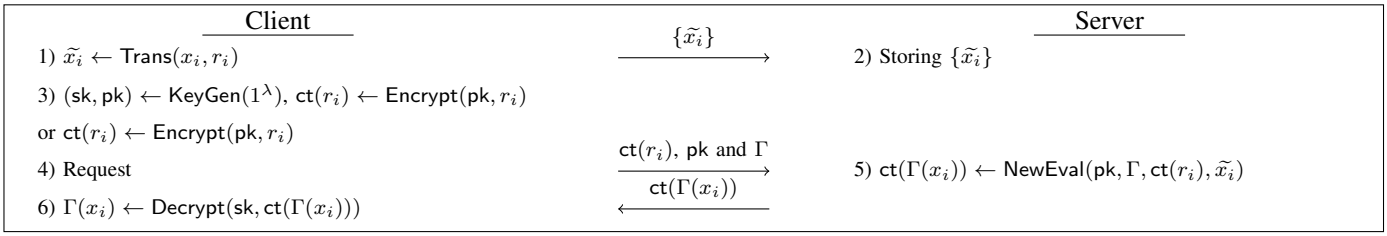

Figure 2: The generic HEAD protocol in the privacy-preserving cloud computing scenario

much less space achieving compact storage, i.e., the ciphertext expansion rate is 1. Moreover, our protocol offers some computation optimization and higher flexibility. We will describe how we achieve compact storage and efficient computation, and discuss the advantages (including the flexibility) and limitations in Sect. IV.

IV. Various HEAD instantiations

In this section, we instantiate our generic HEAD protocol with three protocols, i.e., a modulo-subtraction-masking protocol, a modulo-division-masking protocol, and an XOR-masking protocol. All instantiations achieve compact storage, i.e., decreasing the masked ciphertext expansion rate to 1, and optimize some computation types, in the FHE-based privacy-preserving cloud computing scenario (Sect. II-D). Our NewEval function includes two parts, i.e., an unmasking operation and an evaluation. The evaluation may point to many computation steps. Moreover, our optimization would be described in corresponding subsections and figures. The concrete computation time and noise reduction of the protocols would be exhibited in Sect. V.

A. Modulo Subtraction Masking

Fig. 3 instantiates Trans by modulo-subtraction-masking. In addition to a ciphertext expansion rate of 1, the computation optimization lies in a reduction from arbitrary times of ciphertext-wise additions or a scalar multiplication to a single ciphertext-plaintext addition, significantly reducing the computation time or noise, respectively. We describe this protocol for an integer input vector, which is suitable for an integer FHE scheme such as BFV [24] or BGV [11].

1) Storage optimization for an integer number FHE scheme: We consider an application based on a message space $\mathcal{M}$, whose length is a factor of the word length (e.g., 16 bits) and is slightly smaller than $\mathbb{Z}_t$. For Trans in step 1 in Fig. 3, a client firstly masks $\{x_i \in \mathcal{M}\}$ by element-wise modulo subtracting an integer pseudorandom vector $\{r_i\}$ (generated from $\mathsf{PRF}(\mathsf{key}, i)$), i.e., $\mathsf{Trans}(x_i, r_i) = x_i - r_i \bmod t,\ \forall i \in I$. Since the length of $\mathcal{M}$ is a factor of a word length, the size of the masking results $\{\widetilde{x_i}\}$ could be stored in the same size as the one of the original messages, i.e., $|\{\widetilde{x_i}\}| = |\{x_i\}|$ and the ciphertext expansion rate is 1. The client transfers the masked set $\{\widetilde{x_i}\}$ to a cloud computing server. Then, the client stores the small-size PRF key for a long time as needed, until a computation request is raised.

Next, we show how this modulo-subtraction-masking could be unmasked using the FHE homomorphic properties, i.e., step 5.1 in Fig. 3. When the client wants to launch a cloud computing query on its ciphertexts, the client re-generates the pseudorandom vector $\{r_i\}$ from the PRF using the same key, and generates the FHE scheme (e.g., BFV or BGV) secret and public keys. The FHE scheme should be configured by the same plaintext modulus t as the one in the masking process, while other configurations could be flexible according to the proposed computation circuit Γ. The masking set is encrypted into $\mathsf{ct}(r_i)$ in a SIMD-batching style, and $\mathsf{ct}(r_i)$ would be sent to the server along with the public key and Γ. After receiving the request, the server picks up the ciphertext $\widetilde{x_i}$ from its storage. The server would obtain $\mathsf{ct}(x_i)$ by computing $\mathsf{ct}(\widetilde{x_i}) \leftarrow \widetilde{x_i} + \mathsf{ct}(r_i) = \mathsf{ct}(\widetilde{x_i} + r_i)$. Note that the unmasking could be a SIMD-batched addition between a batched plaintext and a batched ciphertext.

The cloud server then computes the task according to the circuit description Γ, and returns a result ciphertext $\mathsf{ct}(\Gamma(x_i))$ to the client. Finally, the client decrypts it to have $\Gamma(x_i)$.

2) Computation optimization for one or more ciphertext-wise additions and a scalar multiplication: Since this modulo-subtraction-masking is linear, the masked message still keeps the linear homomorphic properties, which could move a linear FHE ciphertext evaluation to the masking side. The masking side operations are much easier than the ones in the FHE ciphertext side. In step 2 of Fig. 3, the client may preprocess $\mathsf{ct}(\sum_{i=1}^{k} r_i)$ or $\mathsf{ct}(c \cdot r_i)$ if the cloud computing task Γ requires a large number of additions or a scalar multiplication.

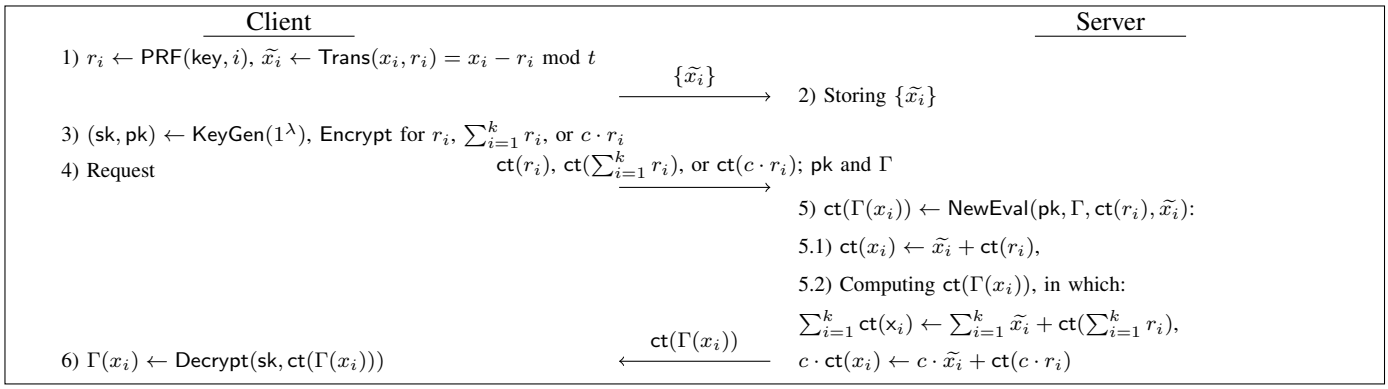

Figure 3: The modulo-subtraction-masking HEAD protocol for an integer number FHE scheme

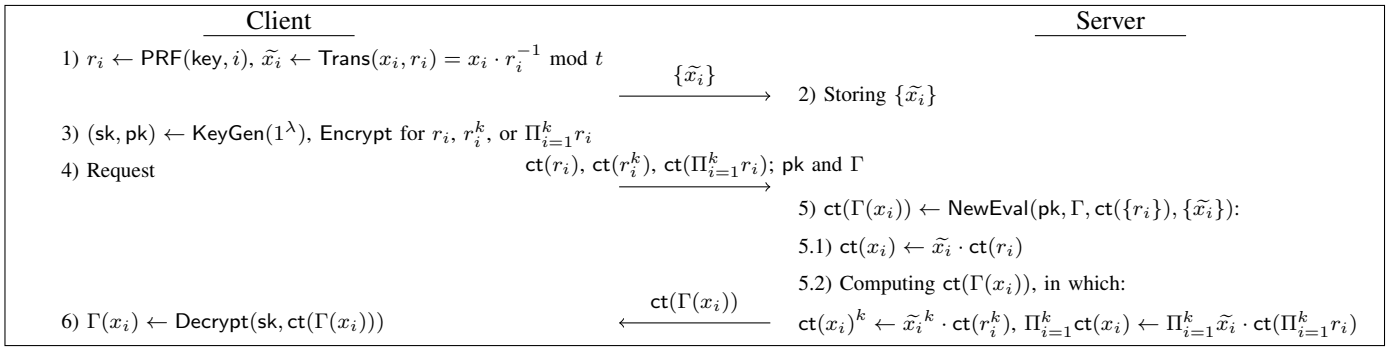

Figure 4: The modulo-division-masking HEAD protocol for an integer number FHE scheme

Step 5.2 in Fig. 3 reduces the sum of k FHE ciphertexts $\sum_{i=1}^k \mathsf{ct}(\mathsf{x}_i)$ to a single FHE ciphertext-plaintext addition $\sum_{i=1}^k \widetilde{x_i} + \mathsf{ct}(\sum_{i=1}^k r_i)$. This is a significant optimization since the computational overhead for the sum of k masked ciphertexts or k pseudorandoms is much small. Furthermore, when the inputs are batched in one FHE ciphertext, the straightforward FHE computation for the sum of k ciphertexts $(\sum_{i=1}^k \mathsf{ct}(x_i))$ should be accompanied with needed rotations or evaluations at indexes, while our technique does not need rotation. This transformation also decreases noise a little.

The optimization also works for an FHE scalar multiplication $c \cdot \mathsf{ct}(x_i)$ by computing the scaling on the masking side and the pseudorandom side, i.e., $c \cdot \widetilde{x_i}$ and $c \cdot r_i$, and then computing a plaintext-ciphertext addition $c \cdot \widetilde{x_i} + \mathsf{ct}(c \cdot r_i)$. Although this optimization decreases a little computation time, the reduction in noise volume is significant.

B. Modulo Division Masking

Fig. 4 introduces the modulo-division-masking HEAD protocol, i.e., instantiating Trans by a modulo division for integer number inputs. Still, this protocol not only achieves compact storage, but also decreases the FHE ciphertext multiplication consumptions in a cloud server. The computation optimization reduces arbitrary times of ciphertext multiplications to a single scalar multiplication, significantly decreasing both the computation time and noise.

1) Storage optimization for a non-zero integer number FHE scheme: This protocol has similar requirements for $\mathcal{M}$ and t to the ones in the modulo-subtraction-masking HEAD protocol (Sect. IV-A). For non-zero integer inputs x_i, Trans invokes $x_i \cdot r_i^{-1} \bmod t$, $\forall i \in I$, in which non-zero r_i comes from a PRF. The unmasking operation inside NewEval only consumes one scalar multiplication to unmask the maskings, i.e., $\mathsf{ct}(x_i) \leftarrow \widetilde{x_i} \cdot \mathsf{ct}(r_i)$. This unmasking also supports the SIMD-batching.

2) Computation optimization for one or more ciphertext-wise multiplications: Step 5.2 in Fig. 4 introduces the modulo-division-masking computation optimization. If a computation request Γ includes the k power of an FHE ciphertext, the client in HEAD would prepare r_i^k by computing the k power of each element r_i, then send the ciphertext $\mathsf{ct}(r_i^k)$ in a cloud computing query. The cloud server computes $\widetilde{x}^k$ and then only consumes one scalar multiplication to compute $\mathsf{ct}(x_i)^k$.

If Γ requires a k-degree cross term from k variables, i.e., $\Pi_{i=1}^k x_i$. The straightforward FHE protocol requires rotations or evaluations at different indexes to achieve the product of k ciphertexts $\Pi_{i=1}^k \mathsf{ct}(x_i)$. Our optimization also works in this case, by firstly computing $\Pi_{i=1}^k \widetilde{x_i}$ and $\Pi_{i=1}^k r_i$ then computing one scalar multiplication of $\Pi_{i=1}^k \widetilde{x_i} \cdot \mathsf{ct}(\Pi_{i=1}^k r_i)$.

These optimizations decrease both a large amount of computation time and noise, compared with the heavy homomorphic operations in the straightforward FHE usage.

C. XOR Masking

The XOR-masking HEAD protocols (in Fig. 5 and Fig. 6) work for a real or binary number input. Both protocols achieve compact storage.

1) Storage optimization for a real number FHE scheme: We describe the positive case here to simplify the expression, and extend it to the general case in Appendix C when keeping one bit to indicate a positive or negative number. We consider a fixed-point positive real number x occupying m bits in Fig. 5, and x^j stands for the j-th bit for $j \in [0, m-1]$. Specifically, the first m_{frac} bits represent the fraction part of this message, i.e., $x^0, \cdots, x^{m_{\mathsf{frac}}-1}$, while the remaining $m - m_{\mathsf{frac}}$ bits indicate the integer part, i.e., $x^{m_{\mathsf{frac}}}, \cdots, x^{m-1}$. Hence, the binary representation of x satisfies $x = \sum_{j=0}^{m_{\mathsf{frac}}-1} x^j \cdot 2^{j-m_{\mathsf{frac}}} + \sum_{j=m_{\mathsf{frac}}}^{m-1} x^j \cdot 2^{j-m_{\mathsf{frac}}} = \sum_{j=0}^{m-1} x^j \cdot 2^{j-m_{\mathsf{frac}}}$.

The Trans operation masks each bit of the input x_i^j by an XOR operation with another random bit r_i generated from a PRF, i.e., $\widetilde{x_i^j} \leftarrow x_i^j \oplus r_i^j,\ \forall j \in [0, m-1],\ \forall i \in I$.

Since a real number FHE scheme like CKKS supports SIMD-batching, $\mathsf{NewEval}$ may execute the unmasking operations in a batching style. For each bit of each element x_i^j, the binary XOR could be converted to a decimal relation, i.e., $x_i^j = \widetilde{x_i^j} \oplus r_i^j = \widetilde{x_i^j} + r_i^j - 2 \cdot \widetilde{x_i^j} \cdot r_i^j$. For an m-bit fixed-point decimal real number, we have

$$
\begin{aligned}
x_i &= \sum_{j=0}^{m-1} \left(\widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}} + r_i^j \cdot 2^{j-m_{\mathsf{frac}}} - 2 \cdot \widetilde{x_i^j} \cdot r_i^j \cdot 2^{j-m_{\mathsf{frac}}} \right) \\
&= \sum_{j=0}^{m-1} \left(\widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}} + (1 - 2 \cdot \widetilde{x_i^j}) \cdot r_i^j \cdot 2^{j-m_{\mathsf{frac}}} \right)
\end{aligned}
$$

Hence, the relation in ciphertexts keeps,

$$
\begin{aligned}
\mathsf{ct}(x_i) &= \sum_{j=0}^{m-1} \left(\widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}} + (1 - 2 \cdot \widetilde{x_i^j}) \cdot \mathsf{ct}(r_i^j \cdot 2^{j-m_{\mathsf{frac}}}) \right), \\
\mathsf{ct}(x_i) &= \sum_{j=0}^{m-1} \widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}} + \sum_{j=0}^{m-1} (1 - 2 \cdot \widetilde{x_i^j}) \cdot \mathsf{ct}(r_i^j \cdot 2^{j-m_{\mathsf{frac}}}).
\end{aligned}
$$

Before a request, the client prepares $\mathsf{ct}(r_i^j \cdot 2^{j-m_{\mathsf{frac}}})$ for $\forall j \in [0, m-1],\ \forall i \in I$, then transfers these ciphertexts (or one single batched ciphertext) to the cloud server in a cloud computing query. After unmasking and computations, the CKKS ciphertext $\mathsf{ct}(\Gamma(x_i))$ is transferred to the client. Note that the client could embed all encrypted bits in one batched ciphertext. With necessary rotations, the unmasking could be efficiently processed.

2) Storage optimization for a binary input FHE scheme: The XOR-masking HEAD protocol (Fig. 6) works for a binary GSW-style FHE scheme, in which the message space is $\mathcal{M} = \{0, 1\}$. For masking an binary x_i, the client runs a bit-wise XOR by another pseudorandom bit r_i in Trans, rendering the same size bit $\widetilde{x_i}$, which also prevents the ciphertext expansion. For unmasking, this XOR HEAD protocol transfers $\mathsf{ct}(r_i)$ to the cloud server, and the unmasking process computes $\widetilde{x_i} \oplus \mathsf{ct}(r_i)$ to recover $\mathsf{ct}(x_i)$.

3) Computation optimization for one or more ciphertext-wise XOR or XNOR operations for a binary FHE scheme: Step 5.2 in Fig. 6 introduces our computation optimization for this protocol. If a computation request Γ requires the XOR or XNOR sum of k ciphertexts (i.e., $\oplus_{i=1}^k \mathsf{ct}(x_i)$ or $\odot_{i=1}^k \mathsf{ct}(x_i)$), the client would prepare and send the ciphertext $\mathsf{ct}(\oplus_{i=1}^k r_i)$ or $\mathsf{ct}(\odot_{i=1}^k r_i)$ along with a cloud computing query, respectively. Since the XOR operation is transitive, we have $\oplus_{i=1}^k \mathsf{ct}(x_i) \leftarrow (\oplus_{i=1}^k \widetilde{x_i}) \oplus \mathsf{ct}(\oplus_{i=1}^k r_i)$, which is a single ciphertext-wise XOR operation. One XOR operation based on two FHE ciphertexts is very slow as the exhibition in Sect. V. Hence, our optimization is significant since we reduce the heavy burden of computing $k-1$ times of XOR for $\oplus_{i=1}^k \mathsf{ct}(x_i)$ to a single XOR.

The optimization also works for XNOR. From $a \odot b = \neg(a \oplus b)$ and $a \odot b \odot c = a \oplus b \oplus c$, we could induce that $\odot_{i=1}^k r_i$ equals to $\oplus_{i=1}^k r_i$ or $\neg(\oplus_{i=1}^k r_i)$ if k is odd or even, respectively. Hence, the XNOR sum of k ciphertexts $\odot_{i=1}^k \mathsf{ct}(x_i)$ could be optimized to $(\odot_{i=1}^k \widetilde{x_i}) \oplus \mathsf{ct}(\odot_{i=1}^k r_i)$ or $(\odot_{i=1}^k \widetilde{x_i}) \oplus \neg\mathsf{ct}(\odot_{i=1}^k r_i)$ for an even k or odd k, respectively. Similarly, the ciphertext-wise XNOR operation is heavy, so that our optimization is also significant.

4) Supporting a symmetric binary FHE scheme: From our implementation experience, two famous FHE open-source libraries PALISADE [42] and TFHE [18] only support the symmetric version of FHE schemes, which do not support a plaintext-ciphertext XOR or an encryption before a ciphertext-wise XOR at the server side in an FHE-based privacy-preserving cloud computing scenario.

In order to apply this XOR-masking protocol in practice, we design another method at the server side (step * in Fig. 6). For computing $\widetilde{x_i} \oplus \mathsf{ct}(r_i)$, the operation is to reverse $\mathsf{ct}(r_i)$ (an NOT ciphertext operation) in essence, according to the bit value of the $\widetilde{x_i}$. Since $x_i \leftarrow \widetilde{x_i} \oplus r_i$, $x_i \leftarrow 0 \oplus r_i = r_i$ if $\widetilde{x_i} = 0$, and $x_i \leftarrow 1 \oplus r_i = \neg r_i$ if $\widetilde{x_i} = 1$. Hence, the server could set $\mathsf{ct}(x_i) \leftarrow \mathsf{ct}(r_i)$ if $\widetilde{x_i} = 0$, or $\mathsf{ct}(x_i) \leftarrow \neg\mathsf{ct}(r_i)$ if $\widetilde{x_i} = 1$. An NOT ciphertext operation takes a much less time (the level of microseconds) than other ciphertext-wise Boolean operations (the level from hundreds of milliseconds or seconds).

D. Discussion

1) Compact storage: the masked ciphertext expansion rate of one: Similar to Gentry et al.'s work [30], [31], a server in our HEAD protocols stores compact ciphertexts in an FHE-based privacy-preserving cloud computing scenario. When the client requests a computation, the AES ciphertexts in [30], [31] and the masked ciphertexts are transferred to high-expansion FHE ciphertexts. Both [30], [31] and ours have an expansion rate of one in the storage phase. However, our protocol is more generic and extra offers computation optimizations than [30], [31], by paying a little communication overhead.

We regard an FHE scheme as a black box in our protocols, which offers us more generality without relying on a specific FHE scheme. The works [30], [31] only could store the AES ciphertext and transfer it to a BGV FHE ciphertext. Our protocols support various masking techniques and FHE schemes depending on the data type computing in a cloud server. In addition, a masked ciphertext could be transformed faster and could support different representations other than $GF(2^d)$ or

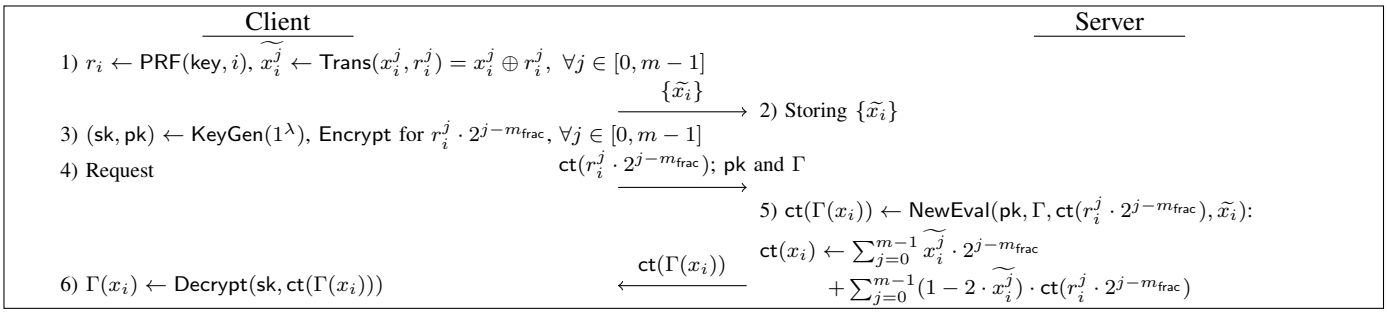

Figure 5: The XOR-masking HEAD protocol for a real number FHE scheme

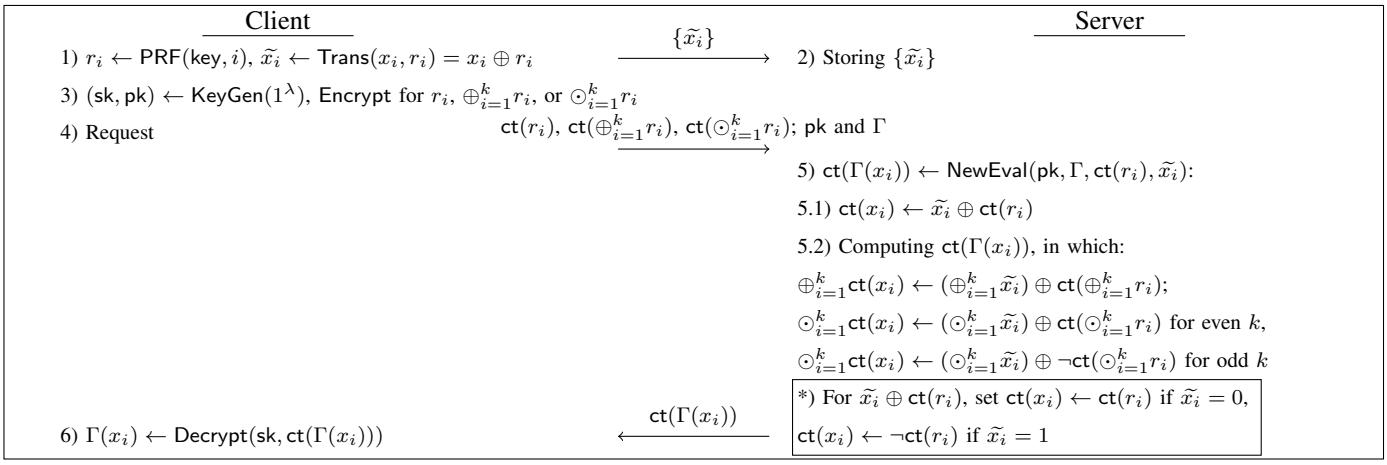

Figure 6: The XOR-masking HEAD protocol for a binary FHE scheme

even optimize some computation, compared with an AES ciphertext. While the computation of an AES ciphertext is limited in the field $GF(2^d)$.

2) Supporting different computation types: The three HEAD protocols optimize different computations depending on the masking technique used in a storage phase. Since the masked ciphertext expansion rate is 1 in all the three instantiations, a client may outsource long-term storage for all the three types (modulo-subsection, modulo-division, and XOR) of masked messages, using three different pseudorandom keys, without occupying too much space. No matter what kind of computation task, the corresponding maskings are picked up and used in the following computation.

3) The flexibility of HEAD protocols: It is worth noticing that the server may store the masked ciphertexts during the life-cycle of the outsourced cloud storage and computing application. Our protocol is not of one-time use. After one-time outsourced storage, the client may propose multiple cloud computing queries on its outsourced ciphertexts without changing the pseudorandom maskings.

A client may upload its I_1 masked messages at the beginning, while each following requested computation may only require I_2 messages and $I_2 << I_1$, i.e., a shorter pseudorandom array as needed, which would further save the communication overhead for the corresponding FHE ciphertext(s). Also, the indexes included in I_2 are not limited. The retrieved messages could be transformed to one or more FHE ciphertexts as required if the selected FHE scheme supports SIMD-batching.

What's more, the encryption parameters of a transformed FHE ciphertext could be configured just before a computation task is proposed. The client configures an FHE scheme flexibly according to a cloud computing task.

4) The value of a plaintext modulus t: As we introduced in Sect. II, an FHE evaluation result is t-modulo after several computations. Different applications have different requirements of the t value. However, no matter what the t value is, our ModSub and ModDiv protocols keep compact storage in an FHE-based privacy-preserving cloud computing scenario.

5) The extra costs of HEAD: As we depicted in Fig. 1 and Fig. 2, the HEAD protocols do not change the procedure of a straightforward FHE privacy-preserving cloud computing protocol too much. In exchange for compact storage and computation optimization, the HEAD protocols may sacrifice some communication overhead to transfer some FHE ciphertexts of (the variants of) the pseudorandom maskings. Since, BFV, BGV, and CKKS support SIMD-batching, the ModDiv and ModSub communication overheads are small, only transferring a batched ciphertext. The XOR-masking protocol for CKKS could also transfer one $n \cdot m$ batched ciphertext when the computation task requires n real numbers and each number has

m bits. Since FHEW (or other GSW-style FHE schemes) does not support batching, both our XOR-masking protocol and a straightforward FHEW privacy-preserving cloud computing protocol would transfer unbatched FHEW ciphertexts.

For the unmasking operations, the computational overhead is small or reasonable in different HEAD protocols, as we discussed in the previous subsections.

6) The trade-off between compact storage and computation optimization for CKKS: Our HEAD XOR protocol for CKKS (Fig. 5) achieves the extremely compact ciphertext storage. However, the ModSub protocol also supports CKKS, and similarly optimizes additions and scalar multiplications, with a small ciphertext expansion.

For a real number x, a scale-round-modulo operation renders $(\lfloor x \cdot \Delta \rceil \mod Q) \in \mathcal{M}$. The scale factor Δ would be the same as the one used in the following CKKS scheme. Implementing Trans by $\widetilde{x_i} = \lfloor x_i \cdot \Delta \rceil - r_i \mod Q$ and selecting $r_i \leftarrow_{\$} \mathbb{Z}_Q$ render the masking size expansion, since usually the value of Q is not small. Considering a double-type input x and $\log_2 Q = 150$, the expansion rate is $\frac{150}{64} \approx 3$.

For the unmasking operation inside $\mathsf{NewEval}$, each element in $\{\widetilde{x_i}\}$ would be multiplied by the scale factor inverse, i.e., $\widetilde{x_i} \cdot \Delta^{-1}$. The client prepares $\mathsf{ct}(r_i \cdot \Delta^{-1})$ and then the server computes $\widetilde{x_i} \cdot \Delta^{-1} + \mathsf{ct}(r_i \cdot \Delta^{-1})$. The masked messages in this way still keep the similar computation optimizations as the ones in the integer case, i.e., step 5.2 in Fig. 3.

7) The open question whether a real input could be division masked to enhance the ciphertext multiplication strength: According to our security proof for the HEAD protocol in Sect. III, a masking method is secure when a mask and the resulting masked message are indistinguishable. The ratio $\widetilde{x} = \frac{x}{r}$ would be too small or too large if the double-precision type variables x and r come from the same space, which is beyond the precision of an FHE scheme like CKKS. Hence, how to use the HEAD protocol to mask a real number input in a division form would is left as future work.

V. Evaluating the HEAD Implementation Performance

This section evaluates the performance of our three HEAD protocols in storage and computation aspects, including the BFV, BGV, CKKS, and FHEW schemes. The straightforward usage of the FHE schemes without our protocols is also exhibited as a baseline. We mainly evaluate the performance using the PALISADE library [42][6], while also obtaining the introduced noise volume from the SEAL library [3]. The testing environment is based on Intel Xeon Gold 6266C CPU @ 3.00GHz and 64 GB memory, with Ubuntu 18.04.2 operating system and GCC 7.5.0 compiler.

A. ModSub for BFV and BGV

We first examine the capabilities of the modulo-subtraction-masking (ModSub) protocol in Fig. 3. We choose the batch size as $8,192$ and sets the ring dimension accordingly. The plaintext and ciphertext modulus are chosen as $t = 65,537$ and $\log_2 Q = 180$, i.e., we use 16-bit unsigned integer to represent input messages. We denote this setting as Setting I and summarize the ciphertext size ($\mathsf{ct_{size}}$), the ciphertext expansion rate (CER), and the transformation cost for BGV and BFV in Table II.

ModSub generates $\{r_i\}$ and computes $\{x_i - r_i \mod t\}$ for 510 μs while a later re-generation of $\{r_i\}$ from the PRF costs 279 μs, which is a very small computation overhead. If a cloud computing query requires all stored messages, one batched unmasking computation for all masked ciphertexts costs only 9.4 ms for necessary encodings combined with one plaintext-ciphertext addition (Step 5.1 in Fig. 3). One straightforward unbatched BFV ciphertext takes 386.97 KiB, and the CER is around $\frac{396{,}261\ \mathrm{B}}{2\ \mathrm{B}} \approx 198,130.5$. Then, 8,192 unbatched BFV ciphertexts occupy a considerable space of 3.02 GiB. The 8,192-batched BFV CER is $\frac{396{,}261\ \mathrm{B}}{2\ \mathrm{B} \cdot 8{,}192} \approx 24.2$. The BGV experiment keeps the same order of magnitude.

Table III exhibits the ModSub computation optimization, which reduces one or more ($k-1$ times of) ciphertext-wise additions ($\mathsf{ct}+\mathsf{ct}$) or a scalar multiplication ($\mathsf{ct}\cdot\mathsf{pt}$) to a single plaintext-ciphertext addition ($\mathsf{ct}+\mathsf{pt}$). We take a sum case to reflect the computation optimization. For a straightforward FHE protocol, when computing the sum of $k = 8$ ciphertexts, it is obvious that equally computing $\mathsf{pt}(8) \cdot \mathsf{ct}(x)$ is the fastest way if the eight ciphertexts are identical, consuming a scalar multiplication for 6.4 ms. When the accumulated items are dispersed in one ciphertext batch, there are extra 7 operations for "evaluation at index" and 7 additions, leading to around 336.3 ms to compute $\sum_{i=1}^{8} \mathsf{ct}(x_i)$ in total. For our ModSub protocol, both the tasks of $\mathsf{pt}(8)\cdot\mathsf{ct}(x)$ and $\sum_{i=1}^{8} \mathsf{ct}(x_i)$, could be computed for 6.3ms. No matter how many numbers of additions ($k-1$ additions takes $(k-1)\cdot$6.3 ms) could be reduce to a single plaintext-ciphertext addition (taking 6.3 ms).

For BGV, the PALISADE library release 1.11.5 extra offers the rotation functionality. The sum could be accomplished in a straightforward FHE protocol by rotations if $x_1, \cdots, x_8$ are arranged in order in a batch[7] for 183.8 ms, which is denoted by $\sum_{i=1}^{8} \mathsf{ct}(x_i)^*$ in Table III. However, ModSub only takes 3.2 ms for the sum of eight ciphertexts, faster than both an evaluation-at-index sum and a rotation sum for the case of batched items in order or not respectively.

Table V exhibits our BFV noise optimization from the SEAL library [3]. Since a ciphertext-wise addition introduces not much noise to the ciphertext, the optimization is slight for several bits when the sum size k is not large[8]. However, the noise of a scalar multiplication is 21 bits in our setting. It makes our optimization noticeable since a plaintext-ciphertext addition introduces tiny noise.

B. ModDiv for BFV and BGV

In order to support a deeper circuit in the experiments, we set the encyption parameters by $n = 8,192, t = 65,537, \log_2 Q = 180$ for BFV and $n = 16,384, t =$

[6]Currently, the PALISADE library supports most FHE schemes.

[7]At first, left-rotate the batch once, then compute an addition to get $\mathsf{ct}(x_1)+\mathsf{ct}(x_2)$. Next, left-rotate the result by 2 slots, and an addition renders $\mathsf{ct}(x_1) + \cdots + \mathsf{ct}(x_4)$. Finally, a 4-slot left-rotation and an addition lead to $\sum_{i=1}^{8} \mathsf{ct}(x_i)$. The PALISADE library release 1.11.5 only supports ciphertext rotations for BGV and CKKS.

[8]The SEAL library release 3.7 does not offer the evaluation-at-index functionality, and only supports the noise value evaluation for BFV.

Table II: Storage optimization for ModSub HEAD (Ciphertext expansion rate abbreviated as CER)

Scheme	Straightforward FHE (Unbatch)		Straightforward FHE (Batch)		ModSub HEAD			
	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{Trans}_{\mathsf{time}}$	$\mathsf{Unmask}_{\mathsf{time}}$
BFV	3,246,170,112 B (3.02 GiB)	198,130.5	396,261B (386.97 KiB)	24.2	16 KiB	1	510 μs	9.4 ms
BGV	4,320,976,896 B (4.02 GiB)	363,731.5	527,463 (515.10 KiB)	32.2	16 KiB	1		6.3 ms

Table III: Computation optimization for ModSub and ModDiv HEAD

Scheme	Setting I			Setting II		
	Operation	Straightforward FHE time	ModSub time	Operation	Straightforward FHE time	ModDiv time
BFV	$\mathsf{ct}+\mathsf{pt}$	6.3 ms	6.3 ms	$(k-1)\cdot(\mathsf{ct}\times\mathsf{ct})$	$(k-1)$·131.5 ms	9.5 ms
	$(k-1)\cdot(\mathsf{ct}+\mathsf{ct})$	$(k-1)$·6.3 ms		$(k-1)\cdot(\mathsf{ct}\times\mathsf{ct})^*$	$(k-1)$·132.1 ms	
	$\mathsf{ct}\cdot\mathsf{pt}$	6.4 ms		$\mathsf{ct}(x)^8$	398.1 ms	
	$\sum_{i=1}^{8}\mathsf{ct}(x_i)$	336.3 ms		$\Pi_{i=1}^{8}\mathsf{ct}(x_i)$	1,219.4 ms	
BGV	$\mathsf{ct}+\mathsf{pt}$	3.2 ms	3.2 ms	$(k-1)\cdot(\mathsf{ct}\times\mathsf{ct})$	$(k-1)$·93.8 ms	9.8 ms
	$(k-1)\cdot(\mathsf{ct}+\mathsf{ct})$	$(k-1)$·6.4 ms		$(k-1)\cdot(\mathsf{ct}\times\mathsf{ct})^*$	$(k-1)$·100.8 ms	
	$\mathsf{ct}\cdot\mathsf{pt}$	9.7 ms		$\mathsf{ct}(x)^8$	357.8 ms	
	$\sum_{i=1}^{8}\mathsf{ct}(x_i)$	423.6 ms		$\Pi_{i=1}^{8}\mathsf{ct}(x_i)$	1,311.4 ms	
	$\sum_{i=1}^{8}\mathsf{ct}(x_i)^*$	183.8 ms		$\Pi_{i=1}^{8}\mathsf{ct}(x_i)^*$	585.0 ms	

Table IV: Storage optimization for XOR HEAD (Ciphertext expansion rate abbreviated as CER)

Scheme	Straightforward FHE (Unbatch)		Straightforward FHE (Batch)		XOR-masking HEAD			
	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{ct}_{\mathsf{size}}$	CER	$\mathsf{Trans}_{\mathsf{time}}$	$\mathsf{Unmask}_{\mathsf{time}}$
CKKS	8,615,895,040 B (8.02 GiB)	525,872.5	1,051,745 B (1,027.09 KiB)	64.2	16 KiB	1	253 μs	289.3 ms
FHEW	33,923,072 B (32.35 MiB)	33,128	N.A.	N.A.	8,192×1 bit	1		8,192×2 μs

Table V: Noise Introduction for ModSub and ModDiv HEAD (BFV parameters: $n = 8,192,\ t = 65,537,\ \log_2 Q = 180$)

Scheme	Operation	Straightforward FHE noise	ModSub or ModDiv noise
BFV	$\mathsf{ct}+\mathsf{pt}$	$<$ 1 bit	$<$ 1 bit
	$\mathsf{ct}+\mathsf{ct}$	$<$ 1 bit	$<$ 1 bit
	$\mathsf{pt}\cdot\mathsf{ct}$	21 bit	
	$\sum_{i=1}^{8}\mathsf{ct}(x)$	3 bits	
	$\sum_{i=1}^{8}\mathsf{ct}(x_i)^*$	5 bits	
	$\mathsf{ct}(x)^8$	86 bits	21 bits
	$\Pi_{i=1}^{8}\mathsf{ct}(x_i)^*$	87 bits	

Table VI: Computation optimization for XOR-masking HEAD protocol (128 bits security for FHEW; XORs or XNORs: the computations for $\oplus_{i=1}^{k}\mathsf{ct}(x_i)$ or $\odot_{i=1}^{k}\mathsf{ct}(x_i)$, respectively)

Scheme	XORs	XNORs
Straightforward FHEW	$(k-1)$×1.71 s	$(k-1)$×1.70 s
XOR-masking HEAD	2 μs	2 μs

$65,537, \log_2 Q = 225$ for BGV, respectively, which is denoted as Setting II in Table III.

The modulo-division-masking (ModDiv) protocol also offers a ciphertext expansion rate of 1. The difference lies in the masking and unmasking overhead. The masking takes 4.1 ms for 8,192 integers, and a scalar-multiplication-based unmasking costs 9.5 ms or 9.8 ms for BFV or BGV, respectively.

Table III exhibits the multiplication saving owing to ModDiv in BFV and BGV. For k-1 times of ciphertext-wise multiplications (or multiplication then relinearization, denoted by $(k-1)\cdot(\mathsf{ct}\times\mathsf{ct})^*$ in Table III), a straightforward BFV protocol takes $(k-1)$·131.5 ms (or $(k-1)$·132.1 ms, respectively), which is reduced to 9.5 ms by ModDiv.

In addition, we compute the eighth power of a ciphertext $\mathsf{ct}(x)^8$ and the product of eight ciphertexts $\Pi_{i=1}^{8}\mathsf{ct}(x_i)$. These two tasks are common in any one or k variables, k degree polynomials in an FHE-based cloud computing query. In the straightforward BFV protocol, invoking $\log_2(k) = 3$ times of multiplication renders the 8 power of the ciphertext, which takes 398.1 ms in total. When $x_1, \ldots, x_k$ are stored in a size-8 batch, 7 times of evaluation at indexes and 7 multiplications cost 1219.4 ms in total. The straightforward BGV protocol could take the rotation way to compute $\Pi_{i=1}^{8}\mathsf{ct}(x_i)$ if the messages are arranged in order in a batch, denoted by $\Pi_{i=1}^{8}\mathsf{ct}(x_i)^*$ in Table III. The multiplication production tasks in the straightforward BGV protocol take several hundreds of milliseconds or seconds.

ModDiv prepares masked $\widetilde{x^k}$ or $\Pi_{i=1}^{k}\widetilde{x^i}$ outside an FHE

scheme. The optimized computation (a single plaintext-ciphertext multiplication) overhead is 9.5 ms or 9.8 ms for computing both $\mathsf{ct}(x)^8$ and $\Pi_{i=1}^{8}\mathsf{ct}(x_i)$ in BFV or BGV, respectively. The time reduction is considerable.

Similarly, Table V shows the noise level introduction of the straightforward FHE operations and our protocol. A scalar multiplication requires 21 bits noise in our protocol, which is much smaller than the one to compute $\mathsf{ct}(x)^8$ (86 bits) and $\Pi_{i=1}^{8}\mathsf{ct}(x_i)$ (87 bits) in BFV. For a larger k value, the noise reduction by our protocol would also increase.

C. XOR-masking for CKKS and FHEW

We implement our XOR-masking HEAD protocol for CKKS and FHEW and select an 8,192-size batch and 8,192 unbatched ciphertexts as in the previous two cases.

Table IV shows our storage optimization compared with the straightforward CKKS usage. The CKKS encryption parameters are configured by $n = 16,384, \log_2 Q = 150$, and we consider 16-bit fixed-point numbers with 8 bits representing the fraction part. When the batch size is 8,192, Trans (Step 1 in Fig. 5), i.e., generating $\{r_i\}$ and computing $x_i^j \oplus r_i^j$, costs 253 μs in total. The straightforward CKKS unbatched or batched ciphertext expansion rates are $\frac{1{,}051{,}745\ \text{B}}{2\ \text{B}} \approx 525,872.5$ and $\frac{1{,}051{,}745\ \text{B}}{2\ \text{B}\cdot 8{,}192} \approx 64.2$, respectively. While XOR-masking for CKKS still keeps a ciphertext expansion rate of 1. The computational overhead to unmask the 8,192 masked messages is relatively higher than the operations in other protocols, but is still reasonable, which consumes around 289.3 ms in total[9].

The implemented FHEW scheme in PALISADE only supports unpacked computations and symmetric encryption[10]. We configure the 128 bits security and leave other parameters as automatic recommendations. Table IV shows that our protocol decreases the ciphertext expansion rate from $\frac{4141\ \text{B}}{1\ \text{bit}} = 33,128$ to 1. The unmasking operation in the symmetric version of the XOR-masking protocol (step * in Fig. 6) reverses the value of $\mathsf{ct}(r_i)$ according to the value of the masking $\widetilde{x_i}$. Each ciphertext NOT operation takes 2 μs. If an 8,192-size masked vector $\{\widetilde{x_i}\}$ are all value 1, Table IV shows the worst case that the unmasking operations take 8,192$\times$2 μs. Our XOR-masking protocol also has a significant computation optimization for one or more ciphertext-wise XOR and XNOR operations. For the tasks of $\oplus_{i=1}^{k}\mathsf{ct}(x_i)$ and $\odot_{i=1}^{k}\mathsf{ct}(x_i)$, the overhead reduces from $(k-1)\cdot 1.7$ s to 2 μs.

VI. HEAD Practical Experiments in PostgreSQL

In this section, we embed the SEAL FHE library into a mainstream database system, PostgreSQL, and compare our HEAD protocols with straightforward FHE usages in practice. PostgreSQL is deployed in a localhost network as a server, and a client is to outsource its storage and computation with or without HEAD. We focus on the BFV ModSub and CKKS XOR-masking protocol. The encryption parameters are $n = 8,192,\ \log_2 Q = 130,\ t = 65535$ and $n = 8,192,\ \log_2 Q = 180,\ \Delta = 2^{30}$ for BFV and CKKS, respectively.

Before launching a cloud computing query, the client in all HEAD protocols needs to run a pre-process and then send a ciphertext (under different encryption schemes) to the server which is not required in the straightforward FHE protocol. We thus view our protocol as trading the ciphertext expansion rate for a bit more computation (at the client side) and communication overhead. Considering the case of a big-data storage setting, each query request only queries a small portion of the large database. Hence, we argue that the storage (i.e., ciphertext expansion rate) constitutes a significant performance bottleneck rather than a pre-processing time and some communication accompanied with a request. Moreover, our practical experiment results (Tab. VII and VIII) show that these extra overhead is reasonable.

A. Cloud Storage in a database

We select 1,500 numbers (16 bits for an unsigned integer or a fixed point real number) as a typical example, since the default limit of the column number in PostgreSQL is 1,600. In our HEAD ModSub and XOR-masking protocols, the 1,500 masked ciphertexts could be easily stored in a row in a database table. If these numbers are encrypted by a straightforward FHE scheme (BFV or CKKS), the size of these 1,500 ciphertexts (276MiB or 487MiB, resp.) is larger than the row size limitation in PostgreSQL[11]. Therefore, the data should be divided into 4 rows and each row consumes 375 columns. Batching is another choice and 1,500 numbers could be encrypted in only one straightforward ciphertext. However, batching is not flexible when the client would like to select part of outsourced numbers to do a cloud computing query. The encrypted numbers in a batched ciphertext have to be selected all or nothing.

We also evaluate the pre-processing time and the storing time observed at the client side in Tab. VII. Masking 1,500 unsigned integers or real numbers only spends several milliseconds. The pre-process includes generating masks and masking the messages in HEAD ModSub and XOR-masking protocols, or encrypting the messages by a straightforward FHE protocol. The storing time is recorded as the interval of a storage SQL instruction. As the experiment results show, the HEAD protocols not only decrease the database storage overhead, but also significantly reduce the writing overhead since the masked ciphertext size is much smaller than the straightforward ones.

B. Cloud Computing from a database

For examining the HEAD protocol performance in a practical cloud computing scenario, we implement the tasks of `sum`, `avarage`, `innerProduction`, and `variance` on 100 numbers. We consider a practical case in which all numbers may be stored at a different time so that a straightforward FHE protocol has to be used in an unbatched way. These 100 numbers would firstly be read from the database and

[9]The computations include sixteen times of encodings, sixteen times of scalar multiplications, and fifteen times of additions.

[10]Another Boolean FHE library, tfhe [18] also only supports unpacked computations and symmetric encryption in the 1.0 version (https://tfhe.github.io/tfhe/releases.html).

[11]The row size limitation in PostgreSQL is 8,160 KiB. Each serialized FHE ciphertext could be stored in a TOAST data structure while the size of a TOAST pointer costs 18Byte. Hence, each row could store around 453 ciphertexts at maximum.

Table VII: HEAD Storage Experiment in PostgreSQL (1500 numbers to be stored)

Scheme	Setting	Arrangement	Storage	Pre. Time	Storing Time
BFV	**HEAD ModSub**	**1 row× 1,500 col.**	**5.9KiB**	**3.7ms**	**15.3ms**
	Straightforward Unbatch	4 rows× 375 col.	276MiB	29.7s	15.0s
	Straightforward Batch	1 row× 1 col.	189KiB	21.8ms	14.3ms
CKKS	**HEAD XOR**	**1 row× 1,500 col.**	**5.9KiB**	**6.6ms**	**15.8ms**
	Straightforward Unbatch	4 rows× 375 col.	487MiB	54.4s	28.2s
	Straightforward Batch	1 row× 1 col.	330KiB	44.7ms	23.1ms

Table VIII: HEAD Computation Experiment in PostgreSQL (100 numbers)

Scheme	Task	Protocol	Pre. Time	SQL Payload	Reading Time	Processing Time
BFV	`sum`	**HEAD ModSub**	0.02s	0.18MiB	**3ms**	**0.06s**
		Straightforward		N.A.	0.2s	2.2s
	`innerProduct`	**HEAD ModSub**	0.02s	0.18MiB	**3ms**	**0.1s**
		Straightforward		N.A.	0.2s	2.8s
CKKS	`sum`	**HEAD XOR**	0.04s	0.32MiB	**3ms**	**0.3s**
		Straightforward		N.A.	0.4s	4s
	`innerProduct`	**HEAD XOR**	0.04s	0.32MiB	**3ms**	**0.3s**
		Straightforward		N.A.	0.4s	4s
	`avarage`	**HEAD XOR**	0.04s	0.32MiB	**3ms**	**0.3s**
		Straightforward		N.A.	0.4s	4s
	`variance`	**HEAD XOR**	0.04s	0.32MiB	**3ms**	**0.4s**
		Straightforward		N.A.	0.4s	5s

input into the computation. We regard the pre-processing time recorded at the client side and the payload size of a SQL instruction (communication overhead) as the trade-offs of our HEAD protocols, while the time savings of reading data from the database and the following computing in a cloud server is our advantages. We choose a practical setting in which the same FHE encryption setting and keys are used several times so that the relinearization and Galois keys are sent only once, whose generation time and communication overhead are not counted.

In the HEAD BFV ModSub experiments, the SQL instruction in the `sum` task includes an FHE ciphertext, $\mathsf{ct}(\sum r_i)$, so that the server only spends one ciphertext-plaintext addition (0.06 s) to unmasking the stored masked ciphertexts (Fig. 3). While the SQL instruction in the `innerProduct` task has a batched FHE ciphertext for $\{r_i\}$. By unmasking, multiplications, rotations, and additions, the cloud server could easily computer the inner product of two 50-element vectors for only 0.1 s. Correspondingly, a straightforward FHE protocol would cost 2.2 s or 2.8 s in these two tasks respectively.

In the HEAD CKKS XOR-masking experiments, we arrange the ciphertexts of all bits of messages in one batch, i.e., the batch size is 1,600 for 100 16-bit-fixed-point real numbers. Similarly, the cloud server would read the masked ciphertexts $\{\widetilde{x_i^j}\}$ from the database and unmask them using the payload in the SQL instruction (the ciphertext for the batch $\{r_i^j \cdot 2^{j-m\mathsf{frac}}\}$, Fig. 5). Then, the instructed task is processed to compute the result. The tasks of `sum`, `avarage`, `innerProduction`, and `variance` are particularly designed to combine multiplications, rotations, and additions FHE operations for better performance. The computing time in the cloud server is reduced from 4 s, 4 s, 4 s, and 5 s to 0.3 s, 0.3 s, 0.3 s and 0.4 s, respectively. Appendix B details the `variance` task.

Tab. VIII shows the trade-off and advantage in reality. The client would pay 20 ms (or 40 ms) computation and 0.18 MiB (or 0.32 MiB) communication overhead (only transferring one FHE ciphertext) in HEAD BFV Modsub (or CKKS XOR-masking, respectively) protocol. However, these costs could trade a lot of storage and computation advantages at the server side. Both the reading and the processing computation time are saved to an order magnitude by our protocols.

VII. Conclusion

In this paper, we present HEAD, a generic FHE-based privacy-preserving cloud computing protocol with compact storage and efficient computation. We provide three instantiations of HEAD for different types of input messages. Our protocols accomplish a ciphertext expansion rate of 1 for storage and our experiments show the significant computation time and/or noise optimization for several evaluation functions. To demonstrate its advantage and utility, we incorporate the HEAD protocol into the PostgreSQL database system and conduct experiments compared to straightforward FHE-based privacy-preserving cloud computing protocols. The results show the compact storage with at least an order of magnitude saving in computation time, at a slight cost of query phase communication and preprocessing. Our design and practical experiment show that the FHE becomes feasible to be applied in privacy-preserving cloud computing.

REFERENCES

[1] Fv-NFLlib. https://github.com/CryptoExperts/FV-NFLlib, May 2016. CryptoExperts.

[2] HEAAN. Online: https://github.com/snucrypto/HEAAN, September 2018. snucrypto.

[3] Microsoft SEAL (release 3.7). https://github.com/Microsoft/SEAL, September 2021. Microsoft Research, Redmond, WA.

[4] Lattigo v2.4.0. Online: https://github.com/ldsec/lattigo, January 2022. EPFL-LDS.

[5] Adi Akavia, Neta Oren, Boaz Sapir, and Margarita Vald. Compact storage for homomorphic encryption. Cryptology ePrint Archive, Report 2022/273, 2022. https://ia.cr/2022/273.

[6] Adi Akavia, Neta Oren, Boaz Sapir, and Margarita Vald. Compact storage for homomorphic encryption. *The 2nd joint EUCN-Haifa Workshop on Cryptography*, November 17th, 2021.

[7] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, Toronto, Canada, November 2018.

[8] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. Faster homomorphic encryption is not enough: Improved heuristic for multiplicative depth minimization of boolean circuits. In *CT-RSA 2020*, pages 345–363.

[9] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In *TCC 2019*, pages 407–437.

[10] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *PKC 2013*, pages 1–13.

[11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[12] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM J. Comput.*, 43(2):831–871, 2014.

[13] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptol.*, 13(1):143–202, 2000.

[14] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *ACNS 2021*, pages 460–479.

[15] Jung Hee Cheon, Jean-Sébastien Coron, Jinsu Kim, Moon Sung Lee, Tancrède Lepoint, Mehdi Tibouchi, and Aaram Yun. Batch fully homomorphic encryption over the integers. In *EUROCRYPT 2013*, pages 315–335.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT 2017*, volume 10624, pages 409–437.

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

[18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption library, August 2016. https://tfhe.github.io/tfhe/.

[19] Jihoon Cho, Jincheol Ha, Seongkwang Kim, ByeongHak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering framework for approximate homomorphic encryption. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021*, volume 13092, pages 640–669.

[20] David Bruce Cousins, John Golusky, Kurt Rohloff, and Daniel Sumorok. An FPGA co-processor implementation of homomorphic encryption. In *IEEE HPEC 2014*, pages 1–6.

[21] David Bruce Cousins, Kurt Rohloff, and Daniel Sumorok. Designing an fpga-accelerated homomorphic encryption co-processor. *IEEE Trans. Emerg. Top. Comput.*, 5(2):193–206, 2017.

[22] Yarkin Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating fully homomorphic encryption in hardware. *IEEE Trans. Computers*, 64(6):1509–1521, 2015.

[23] Léo Ducas and Daniele Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT 2015*, pages 617–640.

[24] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. https://ia.cr/2012/144.

[25] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ronald G. Dreslinski, Karim Eldefrawy, Nicholas Genise, Chris Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption (extended version). *arXiv*, 2109.05371, 2021.

[26] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC 2009*, pages 169–178.

[27] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019*, pages 438–464.

[28] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT 2011*, pages 129–148.

[29] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Field switching in bgv-style homomorphic encryption. *J. Comput. Secur.*, 21(5):663–684, 2013.

[30] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO 2012*, volume 7417, pages 850–867.

[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the aes circuit (updated implementation). Cryptology ePrint Archive, Report 2012/099, 2012. https://ia.cr/2012/099.

[32] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO 2013*, pages 75–92.

[33] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[34] Shai Halevi and Victor Shoup. Algorithms in helib. In *CRYPTO 2014*, pages 554–571.

[35] Shai Halevi and Victor Shoup. Bootstrapping for helib. *J. Cryptol.*, 34(1):7, 2021.

[36] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha P. Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1651–1669.

[37] Kim Laine. Simple encrypted arithmetic library 2.3.1. https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf, November 2017. Microsoft Research, Redmond, WA.

[38] KangHoon Lee and Ji Won Yoon. Efficient adaptation of TFHE for high end-to-end throughput. In *WISA 2021*, pages 144–156.

[39] Jie Li, Yamin Liu, and Shuang Wu. Pipa: Privacy-preserving password checkup via homomorphic encryption. In *ASIA CCS 2021*, pages 242–251.

[40] Wenjie Lu, Shohei Kawasaki, and Jun Sakuma. Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. In *NDSS 2017*.

[41] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT 1999*, pages 223–238. Springer.

[42] PALISADE (release 1.11.5). https://gitlab.com/palisade/palisade-release, September 2021.

[43] Shantanu Rane, Wei Sun, and Anthony Vetro. Secure function evaluation based on secret sharing and homomorphic encryption. In *Annual Allerton Conference on Communication, Control, and Computing (Allerton) 2009*, pages 827–834, 2009.

[44] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. Cheetah: Optimizing and accelerating homomorphic encryption for private inference. In *IEEE HPCA 2021*, pages 26–39.

[45] Sujoy Sinha Roy, Furkan Turan, Kimmo Järvinen, Frederik Vercauteren, and Ingrid Verbauwhede. Fpga-based high-performance parallel architecture for homomorphic computing on encrypted data. In *IEEE HPCA 2019*, pages 387–398.

[46] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.

[47] Furkan Turan, Sujoy Sinha Roy, and Ingrid Verbauwhede. HEAWS: an accelerator for homomorphic encryption on the amazon AWS FPGA. *IEEE Trans. Computers*, 69(8):1185–1196, 2020.

[48] Paul Voigt and Axel von dem Bussche. *The EU general data protection regulation (GDPR) A Practical Guide*. Springer International Publishing, 2017.

APPENDIX

A. HEAD Security Analysis

We consider the HEAD security under a semi-honest adversary who has corrupted Server, which means the adversary will honestly execute the protocol with Client but still could not extract any substantial information of Client's messages.

The proof is according to the standard simulation paradigm with the real/ideal model [13], [33]. Informally speaking, for a semi-honest adversary $\mathcal{A}$, we would construct a simulator $\mathcal{S}$ to simulate Client and execute the protocol with $\mathcal{A}$. However, the distribution in the view of $\mathcal{A}$ is indistinguishable from the one in a real execution, which means no leakage about the information of Client's messages.

*Definition 4 (*IND-CPA *security):* A fully homomorphic encryption scheme FHE is indistinguishable secure under chosen plaintext attacks, if for any PPT adversary $\mathcal{A}$, the following probability holds:

$$\left|\Pr\left[\mathsf{Expt}^{\text{IND-CPA}}_{\mathsf{FHE}} = 1\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda),$$

where the security game $\mathsf{Expt}^{\text{IND-CPA}}_{\mathsf{FHE}}$ is defined in Fig. 7.

Theorem 1: Assume that the underlying fully homomorphic scheme FHE is IND-CPA secure, then our generic HEAD protocol is secure in the presence of a semi-honest adversary.

Proof: Let $\mathcal{A}$ be an adversary who has corrupted the Server; we construct a simulator $\mathcal{S}$ to simulate the Client in the HEAD protocol as follows.

- Upon input of a public security parameter λ, a simulator $\mathcal{S}$ invokes $\mathsf{sk}, \mathsf{pk} \leftarrow \mathsf{KeyGen}(1^\lambda)$ and receives back $(\mathsf{sk}, \mathsf{pk})$.
- $\mathcal{S}$ invokes $\mathcal{A}$ upon input of pk.
- $\mathcal{S}$ randomly chooses $x_i, r_i \leftarrow_\$ \mathcal{M},\ \forall i \in I$, generates $\widetilde{x_i} \leftarrow \mathsf{Trans}(x_i, r_i)$, and computes $\mathsf{ct}(r_i) \leftarrow \mathsf{Encrypt}(\mathsf{pk}, r_i)$.
- $\mathcal{S}$ ramdomly generates a circuit Γ and invokes $\mathcal{A}$ upon input of $(\Gamma, \mathsf{ct}(r_i), \widetilde{x_i})$.
- $\mathcal{S}$ receives $\mathsf{ct}(\Gamma(x_i))$ and outputs the plaintext $\Gamma(x_i) \leftarrow \mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct}(\Gamma(x_i)))$.

We prove that the distribution of $\mathcal{A}$'s view in the simulation is indistinguishable from the distribution in a real protocol execution. The main difference between the simulation by $\mathcal{A}$ and a real execution with an honest Client is the way that $\widetilde{x_i}$ is generated: the Client internally decides its own message x'_i in $\mathcal{M}$, chooses random r'_i and computes $\widetilde{x'_i} \leftarrow \mathsf{Trans}(x'_i, r'_i)$; whereas $\mathcal{S}$ chooses random x_i and computes $\widetilde{x_i} \leftarrow \mathsf{Trans}(x_i, r_i)$. Since the underlying fully homormophic encyption scheme FHE is IND-CPA secure, and r_i, r'_i are chosen randomly, the distributions over the outputs of $\mathsf{Trans}(x_i, r_i)$ and $\mathsf{Trans}(x'_i, r'_i)$ are indistinguishable in $\mathcal{A}$'s view on receiving $\mathsf{ct}(r_i)$. Otherwise, we could construct a distinguisher D for the IND-CPA security experiment $\mathsf{Expt}^{\text{IND-CCA}}_{\mathsf{FHE}}$ naturally. Thus, we conclude that the distribution of the semi-honest adversary $\mathcal{A}$'s view is indistinguishable from the distribution in a real execution, and finish the proof. ■

$\underline{\mathsf{Expt}^{\text{IND-CPA}}_{\mathsf{FHE}}(1^\lambda):}$ $b \leftarrow_\$ \{0,1\}$, $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ $(x_0, x_1) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}}}$ $\mathsf{ct}^* \leftarrow \mathsf{Encrypt}(\mathsf{pk}/\mathsf{sk}, x_b)$ $b' \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{Enc}}(\cdot)}(\mathsf{ct}^*)$ Return$(b' = b)$	$\underline{\mathcal{O}_{\mathsf{Enc}}(\cdot):}$ Return $\perp$ if $x \in \{x_0, x_1\}$ Otherwise return $\mathsf{Encrypt}(\mathsf{pk}/\mathsf{sk}, x)$.

Figure 7: Security game $\mathsf{Expt}^{\text{IND-CPA}}_{\mathsf{HE}}$

B. HEAD CKKS XOR `variance` Protocol in PostgreSQL

The HEAD CKKS XOR `variance` protocol (Alg. 1) in PostgreSQL aims to pursue less communication overhead in a SQL instruction and decrease the computation overhead in a cloud computing server. Since the variance computation is

$$\begin{aligned}\frac{1}{n-1}\cdot\sum_{i=1}^{n}(x_i-\bar{x})^2 &= \sum_{i=1}^{n}\frac{x_i^2}{n-1} - \frac{n}{n-1}\cdot\bar{x}^2,\\ &= \sum_{i=1}^{n}\left(\frac{x_i}{\sqrt{n-1}}\right)^2 - \left(\frac{\sum_{i=1}^{n}x_i}{\sqrt{n(n-1)}}\right)^2,\end{aligned}$$

the server would construct two ciphertexts for $\frac{x_i}{\sqrt{n-1}}$ and $\frac{x_i}{\sqrt{n(n-1)}}$. Then, the square sum minus the sum square leading to the result.

The client firstly recovers the $n \cdot m$ masking bits (r_i^j) using the same PRF seed as the one it uses to store the data. These bits would multiply corresponding scales $2^{j-m_{\mathsf{frac}}}$ in each slot. Especially, the client would extra arrange $n \cdot m$ zero numbers after that $n \cdot m$ slots. Finally, the client encrypts this batch leading to a ciphertext ct_r having $2 \cdot n \cdot m$ messages. The client sends only one ciphertext ct_r in a `variance` SQL instruction.

After receiving the `variance` SQL instruction, the server reads the corresponding $n{\cdot}m$ masked ciphertexts $\{\widetilde{x_i^j}\}$ from the database. The server constructs four vector as the description in the step 3 of Alg. 1. After two encoding (step 4), two scalar multiplication (step 5), two rotate-then-sum (step 6-10), and two addition (step 11) operations, the ciphertexts ct_0 and ct_1 are corresponding to the batches of $\frac{x_i}{\sqrt{n-1}}$ and $\frac{x_i}{\sqrt{n(n-1)}}$, respectively. Next, the step 12&13-17 and the step 13-17& 18 put the $\sum_{i=1}^{n}\left(\frac{x_i}{\sqrt{n-1}}\right)^2$ and $\left(\frac{\sum_{i=1}^{n}x_i}{\sqrt{n(n-1)}}\right)^2$ in the first slot of ct_3 and ct_4, respectively. Note that the final $n \cdot m$ zeros in ct_r ensures the correctness of the step 13-17.

In total, our HEAD CKKS XOR `variance` protocol (Alg. 1) in PostgreSQL consumes two encoding, two multiplication, two scalar multiplication, $2 \cdot (\log_2(m) + \log_2(n))$ left rotation and addition, one substraction, and two ciphertext-plaintext addition operations.

Algorithm 1 `variance` task for n m-bit real numbers in PostgreSQL HEAD CKKS XOR protocol.

// Client recovers r_i^j using the same PRF seed.

Construct vector $\{r_i^j \cdot 2^{j-m_{\mathsf{frac}}}\}$ from r_i^j, $\forall i \in n,\ j \in [0, m-1]$.

Append mn zeros in the end of $\{r_i^j \cdot 2^{j-m_{\mathsf{frac}}}\}$, then encrypt it to ct_r).

// Server receives ct_r in a SQL instruction and reads $\{\widetilde{x_i^j}\}$ from the database.

Construct four $2mn$-element vector $\{a_i^j\}, \{b_i^j\}, \{c_i^j\}, \{d_i^j\}$ as follows. For $\forall i \in n,\ j \in [0, m-1]$,

the $(i \cdot m + j)$-th element is $\frac{(1-2\widetilde{x_i^j})}{\sqrt{n(n-1)}}$ in $\{a_i^j\}$,

the $(i \cdot m + j)$-th element is $\frac{(1-2\widetilde{x_i^j})}{\sqrt{n-1}}$ in $\{b_i^j\}$,

the $(i \cdot m)$-th element is $\frac{\widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}}}{\sqrt{n(n-1)}}$ in $\{c_i^j\}$,

the $(i \cdot m)$-th element is $\frac{\widetilde{x_i^j} \cdot 2^{j-m_{\mathsf{frac}}}}{\sqrt{n-1}}$ in $\{c_i^j\}$.

The remained elements in $\{a_i^j\}, \{b_i^j\}, \{c_i^j\}, \{d_i^j\}$ are zero.

Encode $\{a_i^j\}, \{b_i^j\}, \{c_i^j\}, \{d_i^j\}$ to $\mathsf{pt}_a, \mathsf{pt}_b, \mathsf{pt}_c, \mathsf{pt}_d$, respectively.

Set $\mathsf{ct}_0 \leftarrow \mathsf{ct}(\{r_i^j\}) \times \mathsf{pt}_a$, $\mathsf{ct}_1 \leftarrow \mathsf{ct}(\{r_i^j\}) \times \mathsf{pt}_b$, then do two rescalizations.

for k in $[1, \lceil \log_2(m) \rceil]$ **do**

  $\mathsf{ct}_0' \leftarrow \text{LeftRotate}(2^{k-1}, \mathsf{ct}_0)$,

  $\mathsf{ct}_0 \leftarrow \mathsf{ct}_0' + \mathsf{ct}_0$,

  $\mathsf{ct}_1' \leftarrow \text{LeftRotate}(2^{k-1}, \mathsf{ct}_1)$,

  $\mathsf{ct}_1 \leftarrow \mathsf{ct}_0' + \mathsf{ct}_1$.

Set $\mathsf{ct}_2 \leftarrow \mathsf{ct}_0 + \mathsf{pt}_c$, $\mathsf{ct}_3 \leftarrow \mathsf{ct}_1 + \mathsf{pt}_d$.

Set $\mathsf{ct}_3 \leftarrow \mathsf{ct}_1 \times \mathsf{ct}_1$, do a relinearization and a rescalization.

for k in $[1, \lceil \log_2(n) \rceil]$ **do**

  $\mathsf{ct}_2' \leftarrow \text{LeftRotate}(m \times 2^{k-1}, \mathsf{ct}_2)$,

  $\mathsf{ct}_2 \leftarrow \mathsf{ct}_2' + \mathsf{ct}_2$,

  $\mathsf{ct}_3' \leftarrow \text{LeftRotate}(m \times 2^{k-1}, \mathsf{ct}_3)$,

  $\mathsf{ct}_3 \leftarrow \mathsf{ct}_1' + \mathsf{ct}_3$.

Set $\mathsf{ct}_4 \leftarrow \mathsf{ct}_2 \times \mathsf{ct}_2$, do a relinearization and a rescalization.

Set $\mathsf{ct}_{\mathsf{res}} \leftarrow \mathsf{ct}_3 - \mathsf{ct}_4$, and return $\mathsf{ct}_{\mathsf{res}}$.

Client receives $\mathsf{ct}_{\mathsf{res}}$.

Decrypt then decode $\mathsf{ct}_{\mathsf{res}}$, output the first element.

C. Expressing Both Positive and Negative Numbers in HEAD CKKS XOR Protocol

Sect. IV-C describe how to express a positive fixed-point real number in the HEAD CKKS XOR-masking protocol. Here, we extend the protocol to support both positive and negative fixed-point real numbers, i.e., using a complement to express a negative fixed-point real number. Therefore, a positive number follows the original binary expression, while the binary expression of a negative number takes complements of all bits. For an m-bit real number, we take the first bit as the sign symbol. 0 or 1 means a positive or a negative number, respectively. Next, the second to the $(m_{\mathsf{frac}}+1)$-th bits represent the fraction part. The remained $m - m_{\mathsf{frac}} - 1$ bits are the integer part. We use x^j to denote the j-th bit of a fixed-point real number x. Hence, for $j \in [0, m-1]$, $x = x^0 \cdot (2^{-m_{\mathsf{frac}}} - 2^{m-m_{\mathsf{frac}}-1}) + \sum_{j=1}^{m-1} x^j \cdot 2^{j-m_{\mathsf{frac}}-1}$.

The masking way is similar to the positive case as we describe in Sect. IV-C, i.e., $\widetilde{x^j} \leftarrow \mathsf{Trans}(x^j, r^j) = x^j \oplus r^j$. The server would follow a different way to unmask it, by extra considering the sign symbol. Since, $x^j = \widetilde{x^j} \oplus r^j = \widetilde{x^j} + r^j - 2 \cdot \widetilde{x^j} \cdot r^j$, we have

$$x = (\widetilde{x^0} + (1 - 2 \cdot \widetilde{x^0}) \cdot r^0) \cdot (2^{-m_{\mathsf{frac}}} - 2^{m-m_{\mathsf{frac}}-1}) + \sum_{j=1}^{m-1} \left(\widetilde{x^j} \cdot 2^{j-m_{\mathsf{frac}}-1} + (1 - 2 \cdot \widetilde{x^j}) \cdot r^j \cdot 2^{j-m_{\mathsf{frac}}-1} \right).$$

The server computes the following steps to do unmasking,

$$\mathsf{ct}(x) = \left(\widetilde{x^0} + (1 - 2 \cdot \widetilde{x^0}) \cdot \mathsf{ct}(r^0) \right) \cdot (2^{-m_{\mathsf{frac}}} - 2^{m-m_{\mathsf{frac}}-1}) + \sum_{j=1}^{m-1} \left(\widetilde{x^j} \cdot 2^{j-m_{\mathsf{frac}}-1} + (1 - 2 \cdot \widetilde{x^j}) \cdot \mathsf{ct}(r^j \cdot 2^{j-m_{\mathsf{frac}}-1}) \right).$$