Lightweight FHE-Based Protocols Achieving Results Consistency for Data Encrypted Under Different Keys

Marina Checri, Jean-Paul Bultel, Renaud Sirdey and Aymen Boudguiga
Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Keywords: Homomorphic Encryption, Multi-Key Frameworks, Cloud Computing, Data Consistency.

Abstract: Over the last few years, the improved performances of FHE has paved the way for new multi-user approaches which go beyond performing encrypted-domain calculation for a single user. In this context, this paper proposes several simplified multi-user setups resulting in new FHE-based building blocks and protocols. By *simplified multi-user* setting we mean that, in order to process a user request, the FHE server is able to select only data encrypted under the proper key in an oblivious way. In doing so, information like the distribution of data per user remains private without losing the consistency of the obtained homomorphic results. We conclude the paper with experiments illustrating that these simplified setups, although not universally applicable, can lead to practical performances for moderate-size databases.

1 INTRODUCTION

Homomorphic encryption (HE) allows blind computing on encrypted data, thus providing data confidentiality-by-design, not only during storage or transfer but also during processing. In many situations, institutions hold pieces of knowledge which gain value when put together. Still, for some reasons, such data are often very difficult to share, at least when sharing implies their disclosure to others. In this context, multi-user FHE can serve as a basis to design protocols where participants can encrypt their data under their own keys, and calculations are performed blindly by a server on several contributions.

This paper proposes a *simplified multi-user* HE setting that allows calculations to be performed by a server on data encrypted by several contributors under different keys. It guarantees data and results confidentiality, results' *consistency* and untraceability between a contributor's data and the user authorized to use that data (encrypted under that user's key). By *simplified multi-user* setting, we mean that only one user at a time (the requester) can request other users (the contributors) to encrypt their data under his own public key and send it to the server, then instruct the server to perform calculations on the relevant information stored in its database (that is, contributors' messages encrypted under the requester's key).

The paper investigates four setups or *scenarios* of increasing practical relevance and usefulness. The first setup defines a baseline toy scenario in which the requester can detect whether or not a result has been produced by means of only data encrypted under his key. Intuitively, this setup corresponds to the case where the requester wishes to detect situations in which some data *not* encrypted under his key have been wrongly taken into account to produce the result. The three other setups allow to produce correct results even in presence of computations in between data *not* encrypted under the same key with different trade-offs in terms of communication and storage (two of them analyze a trade-off between transmission and storage of evaluation keys while the last one provides results' compactness). In these latter cases, the server cannot trace the keys under which each data has been encrypted which is a desirable property in some cases. Hence, the server cannot link the contributors' data to the requesters for which they are intended. This prevents statistical or frequency analysis of the data usable by the server for treating the requester query.

We also specify a conditional key switching operator for TFHE (Chillotti et al., 2020), which takes a ciphertext and either returns a ciphertext of the same message if encrypted under a given key or otherwise, a random message encrypted under this same key.

In the following, Sec. 2 is preliminaries on FHE, Sec. 3 presents motivating use-cases. Our contribution is in Sec. 4, followed by the above-mentioned conditional key switching in Sec. 5. Experimental results are in Sec. 6. We conclude with Sec. 7.

Checri, M., Bultel, J., Sirdey, R. and Boudguiga, A.
Lightweight FHE-based Protocols Achieving Results Consistency for Data Encrypted Under Different Keys.
DOI: 10.5220/0012085200003555
In *Proceedings of the 20th International Conference on Security and Cryptography (SECRYPT 2023)*, pages 704-709
ISBN: 978-989-758-666-8; ISSN: 2184-7711

2 PRELIMINARIES ON FHE

In this paper, we consider two homomorphic cryptosystems: BFV and TFHE. BFV (Brakerski, 2012; ?) is based on the RLWE problem and supports *batching*, i.e., a technique for leveraging SIMD operations by encoding multiple messages in different *slots* of a plaintext. The TFHE encryption scheme was proposed in 2016 and updated in (Chillotti et al., 2020). It relies on the TLWE problem, an adaptation of the LWE problem to the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. TFHE provides the most efficient bootstrapping operation for binary plaintexts running in tens of millisecond. BFV and TFHE rely on 3 keys: a public key for encryption, a secret key for decryption, and an evaluation key for re-linearization with BFV, and for bootstrapping with TFHE.

Multi-user approaches for implementing FHE schemes refer to Threshold Homomorphic Encryption (ThHE) and Multi-Key Homomorphic Encryption (MKHE). They involve multiple keys during the decryption of ciphertexts and ensure that no single entity holds the decryption key (i.e. the private key). ThHE schemes were introduced in (Asharov et al., 2012; ?; ?). ThHE has a static setup and needs to remove and re-encrypt all the data each time a user is removed. In contrast, MKHE schemes (described in (López-Alt et al., 2013; Doröz et al., 2016; ?)) remove the need for a key setup by allowing the evaluation server to dynamically extend ciphertexts from encryption under individual keys to ones under the concatenation of several users keys. Then, all the private key owners have to collaborate for decryption. However, the increase of ciphertext size in the number of users results in a huge increase in homomorphic operators' computational costs. Recently, hybrid approaches have been proposed (Aloufi and Hu, 2019) to combine the advantages of both ThHE and MKHE.

3 MOTIVATING USE-CASES

3.1 Communication Between Hospitals

Assume hospital H_0 is the national referral hospital for an orphan disease, and wishes to confirm a new hypothesis by comparing correlations found from its own data with those observed in other hospitals. The other hospitals accept to send the requested indicators to H_0 under two conditions: H_0 must not have access to individual patients' data, covered by medical secrecy, hence data will be encrypted and sent to an external server for homomorphic calculations; moreover, neither H_0 nor the server should identify the contributing hospital, because the few patients it treats for this rare disease might be identified. Thus, in the protocol, hospital H_n encrypts its indicator I_n under H_0's public key and sends it to the server. The latter homomorphically aggregates contributions and sends the result to H_0. Eventually, only H_0 can decrypt the statistic with its private key.

This paper proposes approaches ensuring that neither H_0 nor the server learn anything about the users' private data, nor may identify the contributors themselves. As the referral hospitals for different orphan diseases may not be the same, each hospital may play the role of H_0: thus, the server's database contains data intended for different hospitals without knowing the associated encryption key, hence the referral hospital and the disease data relates to.

3.2 Biometric Templates

Suppose now that an organization maintains a database of biometric data (e.g. fingerprint, facial recognition,...) that regulates accesses to a number of secured rooms according to users' accreditations. Accesses are physically controlled by connected doors D having each a key pair $(\mathrm{sk}_D, \mathrm{pk}_D)$. Upon accreditation to a room controlled by door D, the new user's biometric data is encrypted under public key pk_D and sent in the server. The server does not know the sending door, nor a fortiori the public key used, and only stores the encrypted data.

When an individual i requests access to this room, door D acquires i's biometric data, encrypts it under public key pk_D and sends it to the server. The server then seeks a match with biometric identification data stored in its database: recall that data is encrypted hence search must be performed, say through a homomorphic Hamming distance. The server then issues to the door an encrypted boolean value, depending on whether the match was found or not.

In a non-homomorphic scheme, biometric data would still be stored as an encrypted vector, but the database would be indexed in clear by the doors' identifier, enabling the server to trace it. Our protocol prevents this situation. In addition, the server may be used by several companies: each may be identified through its proxy (IP address), but in no way the rooms that were accessed in it. Thus, the server cannot determine which room has very restricted access nor which one is open to almost everyone. The price to pay is that the entire stored data, encrypted under several keys, must be scanned to seek a match before authorizing or denying such access.

4 ACHIEVING RESULTS CONSISTENCY UNDER DIFFERENT KEYS

We present four protocols that correspond to successive scenarios of increasing usability. We denote by "requester" the user who initiates the request and "contributors" the other participants to the protocols.

4.1 System Setting

The protocols defined in this section manage interacting multi-user data, which are encrypted with different keys. During setup, each user U_n can ask the other users to send their contribution, encrypted under his own public key pk_n, to the server. The server does not know under which key a contributor sent his (encrypted) data. When receiving a request, the server only performs homomorphic calculations using an evaluation key either provided by the requester or stored with each record (depending on the scenario). It then produces results encrypted by construction under the private/public key pair associated to the evaluation key used for the calculations. As such, every user is responsible for identifying those results that were produced under its own key (as those that he, and he alone, can decrypt).

The setup is a semi-honest model. More precisely, the server is assumed to perform correctly what it is expected to do, but observes all behaviours and data exposed to it, trying to extract as much information as possible from its observations. The other participants are assumed to send valid information. The setup assumes that the server and the requester are not colluding. Otherwise they could determine and decrypt all the data under the requester's key.

Needless to emphasize that this semi-honest threat model is too weak for real-world deployment scenarios and in such context, relying on homomorphic encryption alone is unreasonable. Homomorphic encryption should only be considered as what it is: a countermeasure to confidentiality threats from the server. In real-world deployment scenarios, it must therefore be embedded in higher-level protocols resorting to additional (often off-the-shelf) techniques, such as strong authentication of all parties or confidentiality and integrity on all exchanges.

4.2 Detecting Key Inconsistencies

This setup (namely scenario 0) deals with the basic requirement that a single user (the requester) should only be able to detect whether or not a result has been produced by mean only of data encrypted under his key. The server however assumes that all data have been encrypted under the requester key and, hence, that it generates consistent FHE results but also performs redundant FHE calculation results which allow the requester to check this. In reply to its request, the requester must therefore decrypt the result getting a valid value if all data involved in the FHE calculation were encrypted under his own key and a random result (noted \$) otherwise.

A simple approach to sort valid results out of random ones is to require each contributor to concatenate his data with a padding of zeroes before encrypting it under the requester's key. The server then performs two homomorphic calculations with these ciphertexts. It computes both the payload function and the sum of the padding encryptions, which will serve as a verification ciphertext because a homomorphic sum of paddings of zeroes should remain a padding of zeroes. The requester then only needs to check that the decrypted padding (of the verification ciphertext) is indeed all zeroes to assert that the result is consistent and to ensure that it has not been "polluted" by contributions encrypted under a wrong key. For a uniform distribution of encrypted paddings of ℓ zeroes in $\mathbb{F}_q$, false positives occurs with a probability of $1/q^{\ell}$.

4.3 Masking Key Inconsistencies

Scenario 1. From now on, we assume that each of the contributors sends its data multiple times to the server, encrypted under the keys of some of the requesters. That is, the contributors explicitly grant the requesters of their choice access to by-products of their encrypted data through FHE calculations on the server. Besides its ignorance of the requester and contributors, the server now deals with data encrypted under several keys, without being able to determine on its own which data should be used in which calculations. Scenario 1 therefore assumes that every contributor sends his encrypted data, together with a padding of zeroes encrypted under the same key, as well as an associated evaluation key (evaluation keys have to be rerandomized[1] to avoid leaking which data are encrypted under the same key to the server). The server stores each record with these three fields in its database. When the server receives a query from a requester, it performs calculations for each record, using the record's evaluation key then concatenates the associated encrypted padding of zeroes to the encrypted result before sending it. For each result, if the key used was the requester's, the padding decryp-

[1]FHE evaluation keys usually are encryptions of a secret key and can be rerandomized by homomorphically adding encryptions of 0.

tion is a padding of zeroes, otherwise, it is a random padding. The requester can thus quickly determine which is the correct result (the one with a padding of ℓ zeroes). The drawback is that, in reply to a single request, the requester must then decrypt one result per database entry. Then, he uses the padding to identify those results derived from database entries intended for him. In addition, this scenario requires every contributor to send a (usually large) key per database record it is creating.

Scenario 2. Scenario 2 is a variant of the previous one, aiming to avoid transmissions of evaluation keys by the contributors, as well as their storage on the server. To do so, the requester sends his evaluation key along with his request, and the server performs FHE computations under this key to produce the encrypted results. As the calculations are homomorphic, only results obtained from data encrypted under the same key will be meaningfully decryptable by the requester (thanks to the padding approach, as previously described). Scenarios 1 and 2 are similar, but the former imposes on the server to keep K evaluation keys (which may be costly in terms of storage capacity), while the latter only requires the requester's key to be sent (which may have some limitations in terms of bandwidth: typically, for TFHE (Chillotti et al., 2020), bootstrapping key used as an evaluation key is about 113Mb in size with default parameters). In other words, scenario 1 and 2 achieve different communication trade-offs: scenario 2 replaces offline communications and storage (in scenario 1) by online (per-request) communications. Which of the two is the most appropriate depends on the concrete use-case.

Scenario 3. In this last approach, we achieve result compactness and remove the need for post-decryption padding verification on the requester's side. That is, the server only had to produce one result per request, that the requester and only he can decrypt. The idea is to rely on the server to obliviously determine in the encrypted domain whether data are part of a calculation, depending on whether the associated padding of zeroes is encrypted under the appropriate key. First, the server performs a homomorphic equality test to zero on the encrypted padding, resulting in an encrypted boolean value $[b]$. In this section, the boolean function chk denotes a homomorphic equality test to zero: it returns an encryption of 1 if the padding of zeroes is encrypted under a given key and 0 otherwise. Then, depending on whether the equality test succeeds or fails (i.e., $[b] = [1]$ or $[0]$), the server combines $[b]$ with *both* the FHE calculation result $[m]$ *and* a default application-dependant value $[\alpha]$ (think of a neutral element or a 'not-an-answer' value) as: $[m] \cdot [b] + (1 - [b]) \cdot [\alpha]$. As such, the equality test and the selection are homomorphic. Still, since the server does not know which ciphertext to consider, it must perform these calculations on the whole database without knowing if the data of the record is effectively used in the computation.

5 A CONDITIONAL KEY SWITCH OPERATOR FOR TFHE

As we shall see in Sec. 6, scenario 3 (described in Sec. 4.3) can be naturally implemented by means of the TFHE cryptosystem. However, with that cryptosystem, allowing homomorphic interactions of data encrypted under different keys requires an additional manipulation which we now specify.

TFHE uses the bootstrapping key as the evaluation key. More precisely, the bootstrapping key is a TRGSW encryption of the private key under an appropriate distinct key (refer to (Chillotti et al., 2020), for more details). The bootstrapping is performed in several steps, including a blind rotation whose index is precisely the bootstrapping key. In other words, the operation multiplies the input by $X^{(\Sigma_{i=0}^{n} a_i \cdot s_i) - b}$, where s_i is a TRGSW ciphertext of the i^{th} component of the secret key sk. If the encryption key used does not match the bootstrapping key, the indices kept during the blind rotation will not match. This is precisely what occurs in a multi-user context when an operation is performed on an argument encrypted with a *wrong* key pk_{WK} that is not matching the bootstrapping key bk_{RK} (the *right* key).

To compute on a message encrypted under a wrong key pk_{WK}, one needs to force an evaluation under the right key. Recall that messages not encrypted under the right key can be recognized because they give a random result when decrypted under this key. Hence it is sufficient to ensure that the result of the bootstrapping operation performed on a "random" argument remains random and otherwise can be correctly decrypted. Forcing the evaluations to be done with the right key bk_{RK} can be achieved by multiplying the argument by the constant 1 encrypted under the right key pk_{RK}. This amounts to implementing a new conditional key-switching that takes a ciphertext as input and either returns another ciphertext of the same message, if it was encrypted under pk_{RK}, or, otherwise, a random message encrypted under pk_{RK}.

6 EXPERIMENTAL RESULTS

The following results have been obtained in a DELL LATITUDE E7450 computer running on Ubuntu 20.04.5 LTS x86_64 with the CPU Intel i5-5200U (4) @ 2.700GHz and only one core activated. For the BFV cryptosystem (Fan and Vercauteren, 2012), using the SEAL library, we chose a polynomial modulus degree of size 4096, a coefficient modulus size of 109 (36+36+37) bits, and a plain modulus of 1032193. For the TFHE cryptosystem (Chillotti et al., 2020), using the Cingulata compiler (Carpov et al., 2015), we used the default parameters of the TFHE library.

6.1 Homomorphic Hamming Distance

Table 1: Average runtime for the computation of a Hamming distance in seconds, according to the cryptosystem and the size of vectors to be compared (size in bits).

	16b	**32b**	**64b**	**1024b**
TFHE/Cingu.	00,27	00,28	00,33	01,51
BFV/SEAL	00,59	00,62	00,66	00,83

As a “Hello world” functionnality we chose to work with Hamming distance calculations.This distance, which is simple to calculate, is often used as an indicator, according to which a result is acceptable if the number of components that differ between the two vectors is less than a certain bound. The use-case in Sec. 3.2 illustrates this usage. We have implemented the Hamming distance on $\mathbb{F}_2$ under three cryptosystems: BFV (Fan and Vercauteren, 2012), TFHE (Chillotti et al., 2020), and extended Paillier as presented in (Catalano and Fiore, 2015). However, the extended Paillier cryptosystem was more than five hundred time slower than the other two for 1024-bit vectors, so we did not investigate it further. For BFV, the library used was Microsoft SEAL. The BFV implementation uses the batching technique to parallelize the computations. In TFHE, we used the Cingulata compiler, providing the CiBit class, which represents an encrypted bit. This representation is particularly adapted to compute on $\mathbb{F}_2$. Table 1 presents the unitary performances according to the size n of the vectors and the cryptosystem used.

6.2 Performance Result

Scenarios 1 and 2. The first two scenarios are implemented under the BFV cryptosystem using the Microsoft SEAL library and batched ciphertexts (we did not test this scenarios with TFHE since the unitary results in Table 1 show BFV is the most efficient option, when no such things as zero testing are required). The function to compute for the requester is a homomorphic Hamming distance between two binary vectors of size 1024. Table 2 illustrates that, in these scenarios, one can scale (assuming a latency threshold of less than one minute) up to 50 users, which may be usable for some applications (albeit resorting to parallelism to help either downing the latency or increasing the database size by an order of magnitude). Nevertheless, as many results as there are records in the database are returned to the requester, who must then decrypt all these results by himself and extract those intended for him. However, according to table 2, the decryption time for all these results stays acceptable (about one second for 50 results) and so is the volume of data exchanged ($\approx$ 110 ko per ciphertexts so around $\approx$ 5.5 Mo for a 50 records database). These tests also allow us to observe that the scenario takes only slightly less time when the evaluation keys are stored (i.e. in scenario 1), and that scenario 2 works just as well, despite the transmission of the evaluation key of the calculation requester (the requester). The elapsed time for a hundred records remains acceptable for homomorphic computations (less than a minute and a half).

Table 2: Scenario 1 and 2 - Average runtime for evaluations and decryptions, according to the number of database records (min:sec,ms).

DB	**Scenario 1**		**Scenario 2**	
records	Eval	Decrypt	Eval	Decrypt
5	0:04,05	0:00,13	0:04,06	0:00,13
10	0:08,10	0:00,25	0:08,25	0:00,26
50	0:40,57	0:01,27	0:40,58	0:01,26
100	1:21,10	0:02,53	1:21,20	0:02,52

Scenario 3. The third scenario needs a homomorphic test to zero, a capability which is more difficult to provide with certain cryptosystems (BFV, BGV,...) than with others (TFHE). For that case, we chose to use TFHE using the Cingulata compiler for this scenario, as the time tests of the Hamming distance of table 1 do not differ much from those of BFV, and it implements a homomorphic zero test, as it work at the boolean circuit level. As previously, the function to compute for the requester is a Hamming distance between two binary vectors of size 1024. As already emphasized in section 5, the multi-user context requires a new conditional key switching operator for TFHE to do operations between two ciphertexts (possibly over different keys): we force the use of the *right* key on these two ciphertexts before operating. This adaptation adds exactly one bootstrapping for each record in the database.

Table 3: Scenario 3 - Average runtime according to the number of database records, ranging from 16 bits of padding (low reliability) to 64 bits of padding (strong reliability) (min:sec,ms).

Scenario 3	**16 bits**	**32 bits**	**64 bits**
5 records	00:21,93	00:25,34	00:35,23
10 records	00:44,37	00:52,88	01:13,23
50 records	03:47,27	04:28,43	06:01,55
100 records	07:38,88	08:50,99	12:07,74

The running times shown in table 3 illustrate that this case is significantly more computationally costly (more than 12 minutes for a hundred records and 2^{-64} false positive probability) than the previous ones, and scales only to about ten records (about one minute for 2^{-64} false positive probability). Furthermore, according to tables 1 and 3, for 10 records and 2^{-64} false positive probability, we have 73s of computations, of which 15s are devoted to calculating the Hamming distance, which leaves 58s of computation time for the ten tests to zero. Similarly, for 50 records and 2^{-64} false positive probability, 76s stands for the Hamming distance and 286s of runtime for the homomorphic tests to zero. This is the price to pay for results compactness, thus for a lower communication cost. Indeed, scenario 3 results in a $O(1)$ size for the reply, when scenarios 1 and 2 lead to an $O(r)$ reply size, where r is the number of records in the database.

7 CONCLUSION

This paper investigated multi-user setups, the first two setups show two similar protocols allowing each user to recognize the messages that are intended for him and guaranteeing result consistency under multiple keys. In terms of latency, these scenarios can be practically relevant (about one minute of execution time) at a scale of 50 to 100 records in the database for sequentially performed calculations. However, these protocols have a natural potential for parallelization, allowing to compute on a few thousands records per minute. Indeed, these scenarios return one response per record in the database, and the homomorphic calculations on the records can be performed in parallel. So for a server with 50 cores (a standard scale on the NUMA machine market), the execution time would be almost divided by 50. Our third setup allows users to request the result of a calculation on the data addressed to them and returns only one response. This is interesting for many use cases but results in a smaller scaling potential since it can only handle about ten records in one minute, sequentially.

ACKNOWLEDGEMENTS

This work was supported by the France 2030 ANR Project ANR-22-PECY-003 SecureCompute.

REFERENCES

Aloufi, A. and Hu, P. (2019). Collaborative Homomorphic Computation on Data Encrypted under Multiple Keys. *The International Workshop on Privacy Engineering (IWPE'19) co-located with S&P'19*.

Asharov, G., Jain, A., López-Alt, A., Tromer, E., Vaikuntanathan, V., and Wichs, D. (2012). Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *EUROCRYPT 2012*, pages 483–501.

Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P. M. R., and Sahai, A. (2018). Threshold Cryptosystems from Threshold Fully Homomorphic Encryption.

Brakerski, Z. (2012). Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO 2012*, pages 868–886.

Carpov, S., Dubrulle, P., and Sirdey, R. (2015). Armadillo: a compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing*, pages 13–19.

Catalano, D. and Fiore, D. (2015). Using Linearly-Homomorphic Encryption to Evaluate Degree-2 Functions on Encrypted Data. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*.

Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, pages 34–91.

Chowdhury, S., Sinha, S., Singh, A., Mishra, S., Chaudhary, C., Patranabis, S., Mukherjee, P., Chatterjee, A., and Mukhopadhyay, D. (2022). Efficient threshold FHE with application to real-time systems. *IACR ePrint*, page 1625.

Doröz, Y., Hu, Y., and Sunar, B. (2016). Homomorphic AES evaluation using the modified LTV scheme. *Designs, Codes and Cryptography*, 80(2):333–358.

Fan, J. and Vercauteren, F. (2012). Somewhat practical fully homomorphic encryption. *IACR ePrint*.

Kim, T., Kwak, H., Lee, D., Seo, J., and Song, Y. (2022). Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition. *IACR ePrint*, page 347.

López-Alt, A., Tromer, E., and Vaikuntanathan, V. (2013). On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *IACR ePrint*, page 94.