“vr^{2}FHE” - Securing FHE from Reaction-based Key Recovery Attacks

Bhuvnesh Chaturvedi[1], Anirban Chakraborty[1], Ayantika Chatterjee[1], and Debdeep Mukhopadhyay[1]

Indian Institute of Technology Kharagpur, Kharagpur, India,
`bhuvneshchaturvedi2512@gmail.com, ch.anirban00727@gmail.com, cayantika@gmail.com, debdeep.mukhopadhyay@gmail.com`

Keywords: FHE · IND-CVA · Reaction Attack · Full Key Recovery · Application Level Security · Merkle tree

Abstract. Fully Homomorphic Encryption (FHE) promises to secure our data on the untrusted cloud, by allowing arbitrary computations on encrypted data. However, the malleability and flexibility provided by FHE schemes also open up arena for integrity issues where a cloud server can intentionally or accidentally perturb client’s data. Contemporary FHE schemes do not provide integrity guarantees and, thus, assume a *honest-but-curious* server who, although curious to glean sensitive information, performs all operations judiciously. However, in practice, a server can also be *malicious* as well as *compromised*, where it can perform crafted perturbations in the cloud-stored data and computational results to entice the client into providing feedback. While some effort has been made to protect FHE schemes against such adversaries, they do not completely stop such attacks, given the wide scope of deployment of contemporary FHE schemes in modern-day applications. In this work, we demonstrate *reaction-based* full-key recovery attack on two of the well-known FHE schemes, TFHE and FHEW. We first define practical scenarios where a client pursuing FHE services from a malicious server can inadvertently act as a Ciphertext Verification Oracle (CVO) by reacting to craftily perturbed computations. In particular, we propose two novel and distinct *reaction attacks* on both TFHE and FHEW. In the first attack, the adversary (malicious server) extracts the underlying error values to form an exact system of Learning with Errors (LWE) equations. As the security of LWE collapses with the leakage of the errors, the adversary is capable of extracting the secret key. In the second attack, we show that the attacker can directly recover the secret key in a bit-by-bit fashion by taking advantage of the key distribution of these FHE schemes. The results serve as a stark reminder that FHE schemes need to be secured at the application level apart from being secure at the primitive level so that the security of participants against realistic attacks can be ensured. As the currently available *verifiable FHE* schemes in literature cannot stop such attacks, we propose `vr`^{2}FHE (`Verify` - then - `Repair` or `React`) that is built on top of present implementations of TFHE and FHEW, using the concept of the Merkle tree. `vr`^{2}FHE first verifies the computational results at the client end and then, depending on the perturbation pattern, either repairs the message or chooses to request for recomputation. We show that such requests are benign as they do not leak exploitable information to the server, thereby thwarting both the attacks on TFHE and FHEW.

1 Introduction

Fully Homomorphic Encryption (FHE) allows the ability to perform arbitrary computation over encrypted data without the need to decrypt it first. This allows a client to offload its private data securely on an untrusted cloud server [1] while simultaneously allowing it to avail the computation ability of the server. In FHE setting, the general consensus is that the server is considered to be *honest-but-curious*. In other words, it is assumed that the server might try to infer information about the client's data but it will perform the computations judiciously. However, in practical, real-world settings, the server can be *actively malicious* and start undertaking spurious activities, including manipulating the client's data to leak secret information. Such malicious operations are more potent and dangerous as they are not restricted to only leaking information about data; rather, they target the secret key, thereby transforming privacy leakage into a security breach. However, a major challenge for undertaking such an attack is that the server cannot directly observe the effect of data manipulations on its end since it does not possess the secret key. On the contrary, it can tamper with the result of a computation and send it to the client for decryption. If the decryption produces an incorrect result, the client might inform the server of such an erroneous result and ask for re-computations. As mentioned in [27], such a scenario exists in practice in the pay-per-computation model, where the client pays for each correct computation only. In case of a wrong result, it will certainly ask the server for a free re-computation. Such a *reaction* [45,47] from the client might result in the leaking of secret information.

In an FHE setting, the client does not have any means to differentiate between the correct result and a random value sent by the server, since it does not know the result of the computation [2]. In such cases, the client relies on an application-level constraint [64] and only accepts the result if it satisfies such constraint (post decryption); otherwise, it asks the server to perform the computation again. This becomes easier in practice since the client uses certain software to communicate with the cloud and process the results instead of performing them manually [65]. A suitable example of such application-level constraint could be the count of returned rows in the context of encrypted databases. When the server processes a

[1] We have used the terms cloud and server interchangeably throughout the paper.

[2] The premise of FHE rests on the assumption that the client wants to offload the computation to the server, which has computational power and offers it as a service. The client can ask the server to evaluate a circuit on its encrypted data and send back the results in encrypted form. In that case, the client does not know the end result of the computation and, therefore, cannot verify the genuineness of the result.

query of the client, it sends back the resultant rows along with a count of number of rows returned, both in encrypted form. The client matches the returned value with the actual count to check whether it has received all the rows or some got lost (e.g., packet drop) in the transmission. In case of a mismatch, the client can ask the server to re-run the query. It is easy to observe that while the server may honestly evaluate the query to generate correct results, it might tamper the count of rows to force a mismatch and thus induce a *reaction from the client*. In spite of few holistic FHE interity solutions explored in context of Verifiable Fully Homomorphic Encryption [64], such reactions can be induced in general. We consider this reaction, or the lack thereof, from the client as a response of an oracle that, given a ciphertext, outputs whether it decrypted correctly or not. Throughout our paper, we refer to this oracle as Ciphertext Verification Oracle or CVO. We note that such CVO exists due to the application-level constraint (the row count in our example), as the client might not be able to verify the correctness of other computational results (e.g., content of the rows).

Reaction-based attacks have been explored in literature, in the context of FHE schemes [53,65,27]. However, such explorations are limited, and the attacks proposed do not apply to state-of-the-art FHE schemes like TFHE [26] and FHEW [31]. We present a detailed discussion of such works and their application and limitations later in Sec. 3. We focus on these two schemes as ① they belong to the FHE category, i.e., any Boolean circuit of arbitrary depth can be computed and ② they are developed on top of Learning With Errors (LWE) security guarantees. These attacks, which target the lack of integrity guarantees in FHE ciphertexts and computations, have led to the development of techniques that provide the *verifiability of FHE computations*. Such techniques require the server to produce proof or attestation that the result was computed honestly by evaluating the requested circuit on the client's inputs. The client uses this proof to first verify whether the result received from the server was not tampered with and represents genuine computational result of correct circuit. The client proceeds to decrypt only those ciphertexts that have accompanying valid proof or attestation. Otherwise, the client may ask for re-computation for the same ciphertext [64].

1.1 Motivation

Prior works in literature [50,64] have highlighted that CVOs do exist in practice and are frequently unavoidable in practical settings such as Machine-Learning-as-a-Service (MLaaS) and Private Information Retrieval (PIR). Moreover, [53] and [65] have shown how an adversary can leverage feedback from the client to carry out reaction-based attacks on Somewhat Homomorphic Encryption (SHE) schemes. Given such attacks, it is well acknowledged in the literature that integrity issues in current FHE schemes can lead to serious security implications. In order to prevent such attacks as well as ensure genuineness of the computational output by the server, a number of works [16,23,36,48,54,60,64] have been proposed in literature that augments the core FHE scheme with a proof or attestation that can be verified at the client end. Such design approach has been termed as *verifiable FHE* where the server generates proof of genuineness of its computations and sends it to the client along with the actual computational result. The majority of these schemes rely on standard integrity techniques such as Message Authentication Codes (MACs), Zero Knowledge Proofs (ZKPs), and Trusted Execution Environments (TEEs). However, these approaches suffer from major drawbacks when applied to contemporary FHE application settings. For example, MACs cannot be used in applications where server inputs are involved since such MACs can only be generated by the client [64]. Similarly, ZKPs are limited to RLWE schemes that do not involve bootstrapping operations since the underlying mathematics differs across these two operations [64]. Finally, TEEs have a limited amount of memory and are slower compared to the untrusted hardware, while FHE ciphertexts are larger in size and computations are already slow even on untrusted hardware [64]. Therefore, these techniques do not work with Learning with Errors (LWE)-based FHE schemes, such as FHEW [31] and TFHE [26], which provides fastest gate-level bootstrapping.

Apart from the aforementioned limitations, these schemes fail to protect underlying FHE schemes in the presence of Ciphertext Verification Oracle. It must be noted that these works inherently assume that the proof generation process will be done honestly by the server. In other words, the client will be able to identify any perturbation by verifying the proof. However, the server being malicious, can deviate from the proof generation protocol. Moreover, the server might perform both the computation and proof generation honestly but perturb the final result before sending to the client. An alternate scenario could appear where a third-party adversary, snooping on the network traffic between the client and the server, perturbs the data such that it results in proof invalidation. In such cases, although the server is honest, the client might stop using the server's service, or the client can be starved of any correct result. Therefore, it is imperative to consider a *repair* mechanism such that some of the faulty computations can be rectified. In addition, the client must be protected from *reaction-based* attacks such that any reaction (request for re-computation) made by the client must not leak any information regarding the underlying plaintext message, error, or secret key.

1.2 Contribution

In this paper, we *for the first time in literature*, demonstrate *reaction-based* attack on two well-known FHE schemes, TFHE [25] and FHEW [32], to perform full key recovery. We propose two distinct attacks on both schemes by exploiting the client itself as CVO. Further, we also propose an end-to-end software-based countermeasure, built on top of existing implementations of TFHE and FHEW, to thwart such reaction-based attacks. To summarize, we make the following contributions:

1. **Client as a Verification Oracle in IND-CVA notion:** The immense possibility of wide-scale deployment of FHE in different applications across varied domains opens up diverse security scenarios. One such application

domain is Private Information Retrieval (PIR) where the information processing is done homomorphically at the server side, while the results are validated using some application-level constraints at the client side. We define a security game to establish the practicality of application-level constraint under the security notion of IND-CVA (Indistinguishability against Ciphertext Verification Attack). Such a practical application is encrypted database in the context of PIR, where the client performs query on encrypted database and homomorphically computes the number of rows returned by the query.

2. **Attack 1: Recovering Errors by introducing crafted perturbations**: Using the CVO, we propose a reaction-based attack on TFHE and FHEW, where the server maliciously inserts crafted perturbations in the final computational result and send it to the client. Due to the application-level constraint, the client being able to identify decryption failure, responds to the server about re-computation. We, *for the first time*, show that the underlying error of homomorphically computed ciphertext can be recovered by the server through reactions of the client. We undertake a binary search-based approach that, using client feedback, reduces the search space of the errors until only one error value remains, which is the correct error. Once errors are recovered from sufficient ciphertexts, we form a system of exact equations and solve them to recover the secret key.

3. **Attack 2: Optimised Key recovery by exploiting key distribution**: In the first attack, we perturbed the component of the ciphertext that contain information about message, error and the secret key. In this attack, we rather target the other component of the ciphertext which is essentially a randomly generated (by the client) vector and does not posses any sensitive information. We, *for the first time*, show the security implication of this random variable-vector by exploiting the key distribution of the FHE schemes. This attack is more potent that the previous one as it leaks the secret key directly, requiring only "one" reaction from the client to leak a bit of the secret key. This attack shows the implication of key distribution space for FHE schemes from a completely different security perspective. As an interesting outcome, we show that almost half of the secret key bits can be leaked without requiring any decryption failures (or reaction from the client).

4. **$\texttt{vr}^2\texttt{FHE}$ - verifiable and reaction-attack secure FHE**: The currently available works under the aegis of verifiable FHE do not protect the two schemes targeted in this paper. As a final contribution, we propose a countermeasure, called $\texttt{vr}^2\texttt{FHE}$, driven by the rationale of verification as well as repair mechanism depending on the perturbation patterns. $\texttt{vr}^2\texttt{FHE}$ is an effective countermeasure against both the attack proposed in this paper. Additionally, it provides the client with the ability to both verify the result as well as compute the correct result locally in case of a decryption failure, without requiring significant overhead. Our proposed technique does not require any modification to the existing FHE libraries, and it relies on existing cryptographic primitives that are also post-quantum secure.

1.3 Organization

The rest of the paper is organized as follows: Section 2 provides the background of LWE [61] problem along with a brief working of FHEW [32] and TFHE [25] libraries and the concept of Merkle tree which we use to develop our countermeasure, while Section 3 provides a summary of existing attacks on FHE schemes. Section 4 explains the threat model under which our attack works and how the client works as a ciphertext verification oracle. Section 5 explains how we leverage the CVO to mount a key recovery attack. Section 6 provides our experimental results for simulating the attacks. Section 7 provides existing countermeasures and their limitations. Section 8 provides details of our proposed countermeasure while section 9 provides an explanation of how our countermeasure works. Section 10 provides the overhead that our proposed countermeasure introduces, and Section 11 concludes our paper.

2 Background

In this section, we provide a brief background on the LWE problem, which is the underlying mathematical foundation for the FHE schemes we discuss in this paper. We follow it up with the working principles of two well-known FHE libraries that are based upon the LWE primitive. Finally, we provide a brief overview of the working principle of the Merkle tree which we use in the countermeasure against our proposed attack.

2.1 Learning With Errors problem

The idea of the Learning With Errors problem was introduced by Regev in 2005 [61]. Since its inception, LWE and its ring variant RLWE [56] has been used as a foundation of multiple cryptographic constructions [42,8,58,2,18,55,19,14] due to the assumption that it is as hard as worst-case lattice problems. LWE is based on the addition of random noises to each equation in a system of equations, thus turning it into a system of approximate equations, as follows

$$\begin{aligned} a_{11}s_1 + a_{12}s_2 + \cdots + a_{1k}s_k &\approx b_1 \ (mod\ q) \\ a_{21}s_1 + a_{22}s_2 + \cdots + a_{2k}s_k &\approx b_2 \ (mod\ q) \\ &\vdots \\ a_{m1}s_1 + a_{m2}s_2 + \cdots + a_{mk}s_k &\approx b_m \ (mod\ q) \end{aligned}$$

For brevity, let $k > 1$ be an integer and $\mathbf{s}$ be a secret sampled uniformly from some set $\mathbf{S} \in \mathbb{Z}^k$. An LWE sample is denoted by a tuple $(\mathbf{a}, b) \in {\mathbb{Z}_{||}}^k \times \mathbb{Z}_{||}$, where $\mathbf{a} \in {\mathbb{Z}_{||}}^k$ is chosen uniformly and $b = \mathbf{a} \cdot \mathbf{s} + e \in \mathbb{Z}_{||}$. Here e is a noise value, also called error, sampled uniformly from a Gaussian distribution with mean 0 and standard deviation $\sigma \in \mathbb{R}^+$. LWE problem has the following two variants -

- *Search problem*: having access to polynomially many LWE samples, retrieve s.

- *Decision problem*: distinguish between LWE samples and uniformly random samples drawn from ${\mathbb{Z}_{||}}^{k} \times \mathbb{Z}_{||}$.

Both versions are considered to be hard to solve, even for a quantum computer. The attacks on LWE-based schemes try to solve any one of the above problems or to estimate the security level of the schemes based on the parameter set used to implement them. However, once these error values are recovered, they can be removed from the corresponding ciphertext to obtain a system of exact equations which can then be trivially solved.

2.2 Domain of Homomorphic Encryption Schemes

Torus Domain For TFHE Torus [26] is defined as a set of real numbers modulo 1, or real values lying between 0 and 1. It is denoted as $\mathbb{T} = \mathbb{R}/\mathbb{Z} = \mathbb{R}$ mod 1. This set $\mathbb{T}$ along with two operators, namely addition '+' and external product '$\cdot$', forms a $\mathbb{Z}$-module. It means that addition is defined over two torus elements while external product is defined as a product between an integer and a torus element, both of which result in a torus element. The product between two torus elements is not defined. In the CPU implementation of TFHE library [25], Torus elements are defined as 32-bit unsigned integers, and all the operations are performed modulo 2^{32}. The plaintext bits 1 and 0 are encoded as Torus elements μ and $-\mu$.

Integer Domain for FHEW The plaintexts and ciphertexts as well as the underlying operations in the FHEW library [32] are defined over Integers modulo 512. The plaintext space is divided into two halves with each half either representing a 0 (encoded as 0) or 1 (encoded as 128). On the other hand, the ciphertext space is divided into four quadrants representing one of the four possible ciphertext values between 0 to 3. Thus, unlike TFHE where plaintext and ciphertext space is the same, they are different in the case of FHEW.

2.3 Fully Homomorphic Encryption Libraries

In this work, we focus on two well-known FHE libraries, namely FHEW [32] and TFHE [25], both based on LWE. The overall working principle of these two FHE schemes can be broadly broken into three stages. First is the *encryption stage* that runs on the client side and involves the encryption key. Once the ciphertexts are generated, they are sent to the server upon which *homomorphic gate evaluation* is performed. Finally, *bootstrapping* is performed on the resulting ciphertext to reduce the overall noise. The last two stages run on the server and do not directly involve the secret key. Once the computations are done at the server, the final encrypted result is sent to the client for decryption and involves the decryption key.

The Encryption Stage In the secret key setting, the encryption process starts with sampling a noise value $e \in \mathbb{Z}_{||}$ from a Gaussian distribution and adding it to the encoded message x to obtain an intermediate value of $b' = x \pm e$. It then samples a random vector $\mathbf{a} \in {\mathbb{Z}_{\shortparallel}}^{k}$ and performs a dot product with the secret vector $\mathbf{s} \in \mathbb{B}^{k}$ where $\mathbb{B} \in \{0, 1\}$ for TFHE and $\mathbb{B} \in \{0, \pm 1\}$ for FHEW. The result of this dot product is then added to the intermediate value b' to obtain its final value as $b = \mathbf{a} \cdot \mathbf{s} + x \pm e$. The final ciphertext comes out to be $(\mathbf{a}, b)$. The above process is the same in the case of both FHEW and TFHE, the only difference being the length of secret key k and the standard deviation σ of the Gaussian distribution.

In the public key setting, the owner of the secret key generates its public key by first generating a random matrix $\mathbf{A} \in {\mathbb{Z}_{\shortparallel}}^{m \times k}$ and a random vector $\mathbf{e} \in {\mathbb{Z}_{\shortparallel}}^{m}$ consisting of noise values randomly sampled from a Gaussian distribution. It then computes a vector $\mathbf{b} = \mathbf{A} \times \mathbf{s} + \mathbf{e} \in {\mathbb{Z}_{\shortparallel}}^{m}$, where "×" represents the matrix-vector product. This matrix-vector pair $(\mathbf{A}, \mathbf{b})$ acts as its public key. To encrypt an encoded message $x \in \mathbb{Z}_{\shortparallel}$, a user randomly selects a row $(\mathbf{a}, b')$ from the public key and then adds x to b' to obtain b. The pair $(\mathbf{a}, b)$ acts as the ciphertext corresponding to the plaintext message x.

We would like to mention that our attack works irrespective of whether the user is working under the secret key setting or public key setting as our attack targets the decryption stage which involves the secret key in both these settings.

Homomorphic gate evaluation and bootstrapping In both FHEW and TFHE, the server receives two ciphertexts $c_1 = (\mathbf{a_1}, b_1)$ and $c_2 = (\mathbf{a_2}, b_2)$ on which it performs the gate evaluation operation. It does so by defining a gate constant as a pair $(\mathbf{1024}, b^{F}_{gc})$ and $(\mathbf{0}, b^{T}_{gc})$ for FHEW and TFHE, respectively. The second part of these constants, i.e., b^{F}_{gc} and b^{T}_{gc}, are defined differently for the 4 and 10 homomorphic gates, apart from NOT-gate, defined in FHEW and TFHE, respectively. The result $c = (\mathbf{a}, b)$ of the gate computation is evaluated by computing $\mathbf{a} = \mathbf{1024} - (\mathbf{a_1} + \mathbf{a_2})$ and $b = b^{F}_{gc} - (b_1 + b_2)$ under modulo-512 in FHEW. In TFHE it is evaluated by computing $\mathbf{a} = \mathbf{0} \pm (\mathbf{a_1} \pm \mathbf{a_2})$ and $b = b^{T}_{gc} \pm (b_1 \pm b_2)$ under modulo-2^{32}, where the ordering of $+$ or $-$ depends on the homomorphic gate being evaluated. During bootstrapping, which takes place immediately after the gate operation, the noise is reduced followed by a key-switching procedure to switch back to the original secret key as the refreshing operation changes the underlying secret key. In the case of FHEW, an additional modulus switching operation is required to switch the modulus from the ciphertext to the plaintext domain, while in TFHE it is carried out during the noise reduction phase itself.

The Decryption Stage Once the client receives the ciphertext $c = (\mathbf{a}, b)$, a result of some homomorphic computation, it begins the decryption process by computing $\mathbf{a} \cdot \mathbf{s}$, and then subtracting this result from b. As a result of this computation, the client receives a noisy version $x \pm e$ of the underlying encoded plaintext message x, called the phase ϕ of the message. In the case of TFHE, the sign of this phase is checked to output 1 if it is positive and 0 if it is negative. In the case of FHEW, a constant value 64 is added to this phase such that it becomes $x+e'$, where $e' = \pm e+64$. This is then divided by 128 to obtain $x'+e''$, where $0 < e'' < 1$, and x' is the plaintext bit corresponding to the encoded bit x. Finally, the floor value of $x' + e''$ is taken, which removes e'' and reveals the plaintext bit x'. However the last step (checking sign in TFHE and flooring in FHEW) of the decryption operation extracts the correct message only when the associated noise e is below a pre-determined threshold, otherwise, it decrypts incorrectly.

2.4 Merkle Tree

Merkle tree [57] or hash tree is a data structure where the leaf nodes store the hash values of the data blocks while the non-leaf nodes (internal) store the hash values of its child nodes. The root node, which is referred to as Merkle root, stores a single hash value which becomes a part of the proof. Merkle tree is generally used to check the membership of an element (such as a file) in a data block. The data owner generates a Merkle tree by first generating hashes of its data elements, and then repeatedly generates their pair-wise hashes until it is left with only one hash value. The owner stores this hash value on its end and offloads its data along with the generated Merkle tree onto a public system, say a cloud server. Now whenever it requires a data element, it sends a request to the cloud and receives the requested element along with a Merkle proof. This proof consists of the sibling data element of the requested element along with all the non-leaf hash values of the previously generated Merkle tree. To verify, the data owner first generates the hash of the requested element along with its sibling. It then uses the other hashes it received as part of the proof to re-calculate the Merkle root, which it then compares with the one stored with it. If they match then it has received the correct data element, otherwise, the element was modified. The data owner can also publish the Merkle root so that others can also verify its data elements. Thus Merkle tree can achieve both *designated* as well as *public verifiablility.* Originally designed as a signature scheme, it is now used to verify the integrity of data in applications such as file systems [17], NoSQL database [29], distributed version control systems [52], and cryptocurrency [59].

3 Existing Attacks on FHE Schemes

The majority of the works that tried to break the security of FHE schemes either target the underlying hard problems [51,12,3,15,44,13,49,9,4,5,7,6,24,33] or target the implementation of such schemes through side-channel leakages [11,10]. However, the theoretical attacks targeting the underlying mathematical problems do not work efficiently for lattice dimensions that are used in the practical instantiation of such schemes. Moreover, the side channel attacks on such schemes are carried out on the *client side*, since the key generation as well as encryption and decryption operations run on the client side. On the other hand, works also exist which show how a malicious server can carry out key recovery attacks on Somewhat Homomorphic Encryption (SHE) schemes. [24]

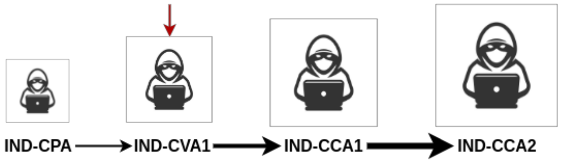


Fig. 1: Strength of active adversary in increasing order (from left to right); weakest in IND-CPA and strongest in IND-CCA2. Our attack works under the IND-CVA1 model.

showed full key recovery attacks on BVG [21], Brakerski [20] and GSW[43] SHE schemes. Similarly, [53] showed a full key recovery attack on Gentry-Haveli [41] SHE scheme. However, all these attacks rely on the adversary having access to a decryption oracle, which is both impractical and infeasible [65].

Recently, a new class of attacks on FHE schemes have been proposed that is based on the idea of *reaction attacks* [45,47] which uses the client as a *verification oracle*. [65] showed a message and key recovery attacks on both the original Gentry's scheme [39,40] as well as the FHE scheme over the integers [30]. However, their attack requires the key to be sparse. Similarly, [53] showed a message recovery attack on a ciphertext-checking SHE scheme, which is a variant of Smart-Vercauteren [63] scheme. However, they did not show a key recovery attack on the scheme. [27] showed the possibility of a key recovery attack by targeting the bootstrapping key, which is an encryption of the secret key, but they did not show it on any existing scheme. Compared to these attacks, ours is a full key recovery attack on two existing FHE schemes, namely FHEW and TFHE. We show two attacks where the first attack recovers the key by first recovering the underlying error values, while the second attack directly recovers the secret key. Our attacks works irrespective of whether the key is sparse or not, and whether or not bootstrapping is present, which makes it more generic and directed toward any LWE-based scheme.

4 Security Notions and Threat Model

In this section, we first explain the different security notions that are relevant in the context of FHE settings. Next, we present a practical scenario in the form of a security game to establish the client as a verification oracle. Finally, we argue how the verification oracle is different from the decryption oracle.

4.1 Security Notions

The security of a cryptographic scheme in the presence of an active adversary is evaluated under various security notions that are defined based on the oracles that an adversary can query during the entire attack. The strength of such an adversary increases with the number and type(s) of oracle(s) it has access to during the entire phase of the attack. In [28], the authors provide a comparison of these security notions. In this paper, we majorly focus on 4 different notions that are relevant in the context of FHE settings, which are shown in Fig. 1 in the increasing order of adversarial strength. These notions are defined as a game between an adversary and a challenger, where the adversary is provided (limited or unlimited) access to certain oracles that it can query. At some point, it sends a pair of messages of its choice to the challenger and receives a challenge ciphertext which is an encryption of one of these messages at random. The adversary wins the game if it can successfully guess the message whose encryption it received with an advantage significantly better than making a random guess. We now briefly explain these notions and their relevance in the context of existing practical FHE schemes.

Indistinguishability against Chosen Plaintext Attack or IND-CPA: This is the weakest notion, where the adversary has (unlimited) access to only an encryption oracle. In a public key setting, an adversary has de-facto access to such an oracle since it can use the public key to run the encryption operation locally. Since this notion ensures that the ciphertext does not leak any information about the underlying plaintext message, all FHE schemes are required to be at least IND-CPA secure.

(Non-Adaptive) Indistinguishability against Ciphertext Verification Attack or IND-CVA1: This notion is stronger than IND-CPA since the adversary is provided access to an additional oracle in the form of a verification oracle. Such an oracle, upon receiving a ciphertext, outputs whether it is valid or not. However, under IND-CVA1 notion, the adversary has access to a verification oracle only before the challenge ciphertext is published.

(Non-Adaptive) Indistinguishability against Chosen Ciphertext Attack or IND-CCA1: This notion is stronger than IND-CVA1 since the verification oracle is replaced with a decryption oracle that directly outputs the result of the decryption. Similar to the IND-CVA1 model, the adversary has access to the decryption oracle only before the challenge ciphertext is published. However, this notion does not put a restriction on the type of ciphertext that the adversary can use to query this oracle. Since most of the existing FHE schemes including FHEW and TFHE involve publishing encryptions of the secret key, such as bootstrapping keys, that are required to perform ciphertext maintenance operations, an adversary can simply query the decryption oracle on them to directly obtain the secret key [34]. Thus these schemes are not IND-CCA1 secure.

(Adaptive) Indistinguishability against Chosen Ciphertext Attack or IND-CCA2: This is the strongest notion, where the adversary has unlimited access to both encryption and decryption oracles, with the only restriction being that the adversary cannot query the decryption oracle on the challenge ciphertext itself. However, IND-CCA2 security requires the ciphertext to be non-malleable, i.e., the adversary should not be able to modify the challenge ciphertext c which encrypts a message m to another ciphertext c' which encrypts a message m' such that $m' = f(m)$ and f is known to the adversary. Since the basic feature

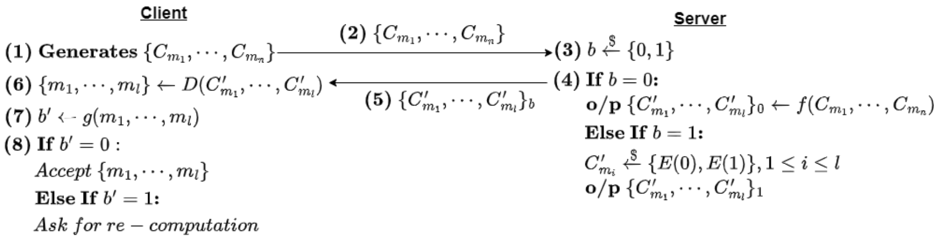

Client
Server
(1) Generates $\{C_{m_1},\cdots,C_{m_n}\}$
(2) $\{C_{m_1},\cdots,C_{m_n}\}$
(3) $b \xleftarrow{\$} \{0,1\}$
(4) If $b=0$:
o/p $\{C'_{m_1},\cdots,C'_{m_l}\}_0 \leftarrow f(C_{m_1},\cdots,C_{m_n})$
Else If $b=1$:
$C'_{m_i} \xleftarrow{\$} \{E(0),E(1)\}, 1 \leq i \leq l$
o/p $\{C'_{m_1},\cdots,C'_{m_l}\}_1$
(5) $\{C'_{m_1},\cdots,C'_{m_l}\}_b$
(6) $\{m_1,\cdots,m_l\} \leftarrow D(C'_{m_1},\cdots,C'_{m_l})$
(7) $b' \leftarrow g(m_1,\cdots,m_l)$
(8) If $b'=0$:
Accept $\{m_1,\cdots,m_l\}$
Else If $b'=1$:
Ask for re − computation

Fig. 2: The game works as follows: (1) the client generates ciphertexts by encrypting its inputs, (2) the client sends these ciphertexts to the server, (3) the server samples a random bit b, (4) if b is 0, the server evaluates the known function f on the input ciphertexts, otherwise randomly samples ciphertexts, (5) it sends back the obtained ciphertexts to the client, (6) client decrypts the received ciphertexts to obtain resultant plaintexts, (7) evaluates function g to obtain bit b', (8) it sends feedback if b' is 1 as it obtained a random result, otherwise keeps the message.

of all FHE schemes is ciphertext malleability, no schemes including FHEW and TFHE can be IND-CCA2 secure [27].

From the above definitions of the security notions, it is easy to observe that while the existing FHE schemes are IND-CPA secure, they are not IND-CCA1 secure. Therefore, current FHE schemes can be broken in presence of a decryption oracle. However, the comparatively weaker security notion IND-CVA1, which allows the availability of verification oracle, applies to FHE and thus requires critical security evaluation. Interestingly, such verification oracles can be found in many practical applications where FHE can be applied. In the following subsection, we discuss about one such application.

4.2 Client as a verification oracle

In the context of FHE, the client acts as a verification oracle when it, upon decrypting the received ciphertext, finds the result to be incorrect and thus asks the server for a free re-computation. However, since the client does not know the result of the requested computation, the decrypted value will essentially look random to it. In other words, the client will not be able to differentiate between a correct result and a random value. We explain this situation in the form of a game, as shown in Fig. 2. The game starts with the client encrypting its input using a homomorphic encryption scheme. It sends these ciphertexts $\{C_{m_1},\cdots,C_{m_n}\}$ to the server. The server randomly samples a bit b, and based on its value performs one of the following two operations:

1. If $b=0$, it computes the function f on the input ciphertexts $\{C_{m_1},\cdots,C_{m_n}\}$ to obtain the resulting ciphertexts $\{C'_{m_1},\cdots,C'_{m_l}\}_0$.
2. If $b=1$, it generates the resulting ciphertexts $\{C'_{m_1},\cdots,C'_{m_l}\}_1$ by generating l ciphertexts randomly. It can do so by first randomly sampling a bit m_i

where $1 \leq i \leq l$ and then generating an encryption of m_i by running a homomorphic gate operation on a combination of some input ciphertext C_{m_j}, where $1 \leq j \leq n$. For example if $m_i = 0$, it performs $XOR(C_{m_j}, C_{m_j})$ in case of TFHE, and $AND(C_{m_j}, NOT(C_{m_j}))$ in case of FHEW, to obtain an encryption of 0. On the other hand if $m_i = 1$, it performs $XNOR(C_{m_j}, C_{m_j})$ in case of TFHE, and $OR(C_{m_j}, NOT(C_{m_j}))$ in case of FHEW, to obtain an encryption of 1.

Finally it sends the resulting ciphertexts $\{C'_{m_1}, \cdots, C'_{m_l}\}_b$ to the client. The client decrypts these ciphertexts to obtain the corresponding plaintext messages. At this point the client does not know whether these messages correspond to the actual computation of the function, i.e., they have come from $world - 0$, or are randomly generated, i.e., they have come from $world - 1$. To differentiate between these two cases, the client relies on some pre-defined *application-level constraints* [64]. One can think of this constraint as some function g that the client applies on either the whole set or a subset of the resultant plaintexts. The outcome of this function is a bit b' that has the same value as b. In other words, this function g helps the client in identifying whether it has received the correct result or incorrect result (random value). However, since this function is a part of the (publicly-available) application that is being run in the cloud-computing scenario, the server is well aware of this function and thus can target it to instigate reactions from the client. It can do so by tampering with the ciphertexts that it returns to the client whose decryption forms the input to this function. The client will ask for a free re-computation if the application-level constraint fails, otherwise, it will simply accept the returned result while assuming it to be correct.

Private Information Retrieval (PIR). We now take an example of a PIR application to understand the implementation of such constraints in a practical setting and show how an adversary can take advantage of it to obtain reactions from the client. We assume a scenario where the client stores its database in an encrypted form on a server and then queries it with encrypted parameters. The server runs the query and returns the resultant rows to the client, *along with homomorphically calculated count of the rows returned.* Upon receiving the ciphertext stream, the client decrypts it and matches the row-count value with the number of received rows to ensure that there are no missing rows. This acts as the application-level constraint as the counts will match if the query is evaluated correctly, otherwise not. As the server knows the query evaluation circuit, it also knows the position of the encrypted row-count in the encrypted bit-stream. Moreover, since the row-count is a result of a homomorphic evaluation, it cannot be decrypted by the server to communicate in plaintext. The server can simply manipulate this encrypted row-count to cause a mismatch between the decrypted value and the actual number of rows returned. The client requests a re-computation if it detects a mismatch, assuming that this was caused due to an incorrect query processing. The server can exploit this reaction to leak secret information (we show a detailed exploit using such reaction in later sections). In a real-life banking scenario using the aforementioned database application,

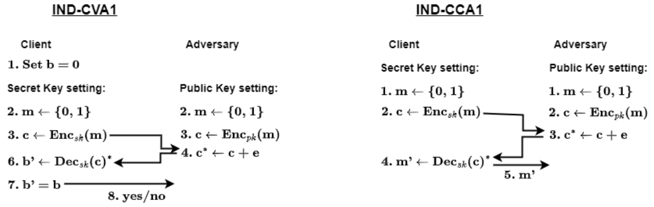

IND-CVA1
Client
Adversary
1. Set $b = 0$
Secret Key setting:
Public Key setting:
2. $m \leftarrow \{0, 1\}$
2. $m \leftarrow \{0, 1\}$
3. $c \leftarrow Enc_{sk}(m)$
3. $c \leftarrow Enc_{pk}(m)$
4. $c^* \leftarrow c + e$
6. $b' \leftarrow Dec_{sk}(c)^*$
7. $b' = b$
8. yes/no
IND-CCA1
Client
Adversary
Secret Key setting:
Public Key setting:
1. $m \leftarrow \{0, 1\}$
1. $m \leftarrow \{0, 1\}$
2. $c \leftarrow Enc_{sk}(m)$
2. $c \leftarrow Enc_{pk}(m)$
3. $c^* \leftarrow c + e$
4. $m' \leftarrow Dec_{sk}(c)^*$
5. m'

Fig. 3: Difference between the client as IND-CVA1 vs IND-CCA1 oracle in the context of FHE settings.

millions of such queries are handled per second by the server. Assuming a modest $100,000$ queries per minute and perturbation in the row-count field for just 10 queries (i.e., perturbing 10 ciphertexts), the failure rate would be 0.0001% if all these perturbations cause erroneous decryption. Such failures can occur even when the query is run in plaintext on an unencrypted database and thus is tolerable in such strained applications. The server can mask the decryption failure with this application-level failure to avoid raising suspicion.

4.3 IND-CVA1 vs IND-CCA1 Oracles

Unlike contemporary encryption schemes, in the context of FHE settings, ciphertext validity cannot be ensured at the primitive level since the FHE ciphertexts are usually full-domain. In other words, given a set of all possible FHE ciphertexts $\mathcal{C}$, all $c \in \mathcal{C}$ decrypt to a valid plaintext. Moreover, for the FHE schemes that we target in this paper, the message space is binary which is not sufficient to have an in-built constraint [27]. However, the validity of such ciphertexts is ensured at the application level when these ciphertexts are combined to generate meaningful information. This is done by applying an application-level constraint on the resulting plaintext value and checking whether the constraint is satisfied or not. For example, in the PIR application provided in the preceding subsection, the validity of the ciphertexts is checked by matching the returned value of the row-count field with the actual number of rows returned and the client only reacts when there is a mismatch between the counts. One can clearly observe that such reactions do not leak either the value of the row-count or the contents of the returned rows. On the other hand, the client behaves as a decryption oracle when it decrypts the result of some homomorphic computation and directly publishes it. However, in our example, the client will not publish the decryption result since these queries usually return the information pertaining to specific clients and are meant to be kept confidential for privacy reasons.

For brevity, we formally explain the difference between verification oracle (IND-CVA1 notion) and decryption oracle (IND-CCA1 notion) under both secret key and public key settings. We show the difference in the form of a game (as shown in Fig. 3) between the client and an adversary, which is the server itself in our case.

Client as (IND-CVA1 style) verification oracle: In this game, the client first sets a secret bit b to 0, irrespective of whether we are working in the secret key setting or the public key setting. We assume that the client does not make the value of b public.

- **Secret key setting**: The client generates a random message bit and encrypts it using its secret key to generate the ciphertext c. It then sends it to the server to evaluate a known function f. The server simply adds a perturbation e to this ciphertext to obtain the final ciphertext c^* and then sends it to the client to be verified. The client decrypts this ciphertext c^* using its secret key to recover the plaintext bit b' and matches it with b.
- **Public key setting**: The server itself generates a random message bit and encrypts it using the public key to generate the ciphertext c. It adds a perturbation e to this ciphertext to obtain the final ciphertext c^* and then sends it to the client to be verified. The client decrypts this ciphertext c^* using its secret key to recover the plaintext bit b' and matches it with b.

In both settings, the client outputs *yes* if $b' = b$, otherwise, it outputs *no*. The output of this oracle provides no additional information to the adversary since it does not know the value of b. Moreover, in the secret key setting, it does not even know the value of m. On the other hand, in the public key setting, when the client outputs *yes*, the server will not be sure whether the perturbation caused the underlying message m to flip to b, or whether the perturbation was unsuccessful and the original value of m was itself equal to b. Thus in both cases, the probability that it can successfully generate a ciphertext c^* that decrypts to b is no better than if it had randomly guessed the value of b. However, in the public key setting, if the client outputs *no*, then the server knows that the added perturbation was successful in causing the bit to flip. Moreover, since it generated the message bit m and successfully flipped it to some message bit m', it now knows that the secret bit b initially generated by the client is same as m. Thus the feedback from the client inadvertently leaks the secret information.

We would like to highlight the fact that the adversary would always win the game if the value of b is made public since it can always deterministically generate b' that is equal to b. Thus the adversary gains no additional information even if it is provided an access to the verification oracle since it will already know the outcome of the query.

Client as (IND-CCA1 style) decryption oracle: In this game, the client does not generate any secret bit b and the game directly starts with the generation of a message.

- **Secret key setting**: The client generates a random message bit and encrypts it using its secret key to generate the ciphertext c. It then sends c to the server to evaluate a known function f. The server simply adds a perturbation e to this ciphertext to obtain the final ciphertext c^* and then sends it to the client to be decrypted. The client decrypts this ciphertext c^* using its secret key to recover the plaintext bit m' and sends it back to the server.

– **Public key setting**: The server itself generates a random message bit and encrypts it using the public key to generate the ciphertext c. It adds a perturbation e to this ciphertext to obtain the final ciphertext c^* and then sends it to the client to be decrypted. The client decrypts this ciphertext c^* using its secret key to recover the plaintext bit m' and sends it back to the server.

Thus, unlike the verification oracle, which outputs whether the result of some computation was correct or not, the decryption oracle directly reveals the result of the computation. Moreover, in the public key setting, the adversary can directly observe the effect of the perturbation based on the decrypted result since it knows the original plaintext message. One can clearly observe from these games that in the case of verification oracle, information is leaked passively only when the client provides a feedback, and not otherwise. On the other hand, in the case of decryption oracle, the client always leaks information directly by providing the decrypted result to the adversary itself. This provides the adversary with more power since it can simply query the decryption oracle on the output of some homomorphic operation and obtain the result in plaintext. On the other hand, querying the verification oracle on the output of some homomorphic operation may or may not leak information about the underlying plaintext.

5 Reaction Attack LWE-based on FHE Schemes

With the stream of ciphertext messages at the helm of the server, it can now launch “reaction” attacks on randomly chosen ciphertext samples which are part of the result of some homomorphic computation. We assume wlog. that out of m such ciphertexts, the server randomly samples n ciphertexts, where $n \ll m$, to introduce purposeful perturbations. This is a reasonable assumption in the cloud computing setting as the ciphertexts in the schemes that we are targeting are essentially encryptions of single-bit information and a collection of such ciphertexts are required to denote a meaningful plaintext message. It is worth mentioning here that the value n is of the order $\Omega(k)$ where k is the size of the secret key in bits.

Targeting the decryption error threshold The decryption process in FHE schemes takes place at the client end after the homomorphically computed result on ciphertexts reaches the client. Due to the accumulation of the errors after homomorphic gate operations at the server, the total error in the computed result increases which is then brought down using the bootstrapping operation to retain homomorphicity. Otherwise, once the accumulated error crosses the pre-defined threshold e_{th}, it results in an incorrect decryption. We leverage this fact to forcefully induce failed decryption by introducing errors purposefully. The objective of the server is to breach the threshold e_{th} during decryption. Now for every ciphertext, the server already knows the error range in which the underlying error value lies. More precisely, given a ciphertext C_r containing

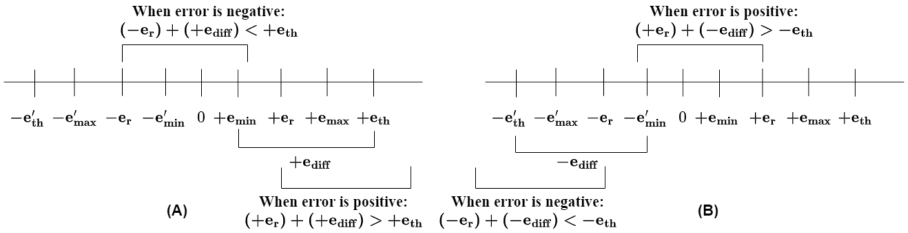


Fig. 4: Different bounds of errors plotted on a number line where (A) shows the effects of added perturbation when the positive range is chosen, and (B) shows the effects of added perturbation when the negative range is chosen.

unknown error value e_r, the server already knows a range of absolute values of error bounded by a minimum value, $\pm e_{min}$, and a maximum value, $\pm e_{max}$. However, the server does not have the knowledge about the sign of e_r, and therefore, the exact value and sign of e_{min} and e_{max}.

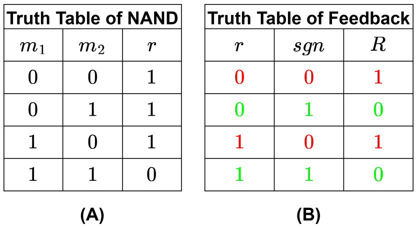
Modifying the final computed result Consider the error number scale denoted in Fig. 4. The actual error e_r and the error threshold can be either positive (e_{th}) or negative (e'_{th}). As a consequence, the error range denoted by e_{min} and e_{max} can have either positive or negative (e'_{min} and e'_{max}) values. Therefore, any positive error value $+e_r$ would essentially lie between the range $+e_{min}$ and $+e_{max}$. The converse is true for negative error values. Therefore, the following relations hold for both positive and negative error values.

$$-e'_{th} \leq -e'_{max} < -e_r < -e'_{min}$$

$$+e_{min} < +e_r < +e_{max} \leq +e_{th}$$

For correct decryption at the client's end, the actual error e_r must be less than $+e_{th}$ or greater than $-e_{th}$. We further note that the error e_r also lies between either of the known ranges $-e'_{max}$ and $-e'_{min}$ or $+e_{min}$ and $+e_{max}$. However, the server neither knows the value nor the sign of the actual error e_r. Thus the server first has to make a random guess about the sign of the error. Suppose the server guesses that the error e_r is $+ve$. In that case it computes the quantity $e_{diff} = +e_{th} - (+e_{min}) = +e_{th} - e_{min}$. Now, we add the term e_{diff} with the computed result of the homomorphic gate operation. As depicted in Fig. 4(A), if the server guessed incorrectly and the original error (after homomorphic gate operation) is negative, the final error after perturbation lies within the permissible range (less than the threshold), albeit in the opposite sign domain. In contrast, if the server guessed correctly and the error is positive, the final error after the addition of perturbation lies beyond the permissible range (more than the threshold) in the positive domain.

The server may also start by guessing that the error e_r is $-ve$. In that case it computes the quantity $e_{diff} = -e_{th} - (-e_{min}) = -e_{th} + e_{min}$. Now, we add



Truth Table of NAND		
m_1	m_2	r
0	0	1
0	1	1
1	0	1
1	1	0

(A)

Truth Table of Feedback		
r	sgn	R
0	0	1
0	1	0
1	0	1
1	1	0

(B)

Fig. 5: (A) Truth table of NAND gate, and (B) Truth table for initial client feedback, where r denotes the result of gate computation, sgn represents whether the sign of error in this result is positive (0) or negative (1), and R represents whether the feedback is received (1) or not (0).

the term e_{diff} with the computed result of the homomorphic gate operation. As depicted in Fig. 4(B), if the server guessed incorrectly and the original error (after homomorphic gate operation) is positive, the final error after perturbation lies within the permissible range (less than the threshold), albeit in the opposite sign domain. In contrast, if the server guessed correctly and the error is negative, the final error after the addition of perturbation lies beyond the permissible range (more than the threshold) in the negative domain. Therefore, it is easy to note that the decryption failure would only occur when the server correctly guesses the sign of the error.

In the next subsections, we show two attacks that work for both TFHE and FHEW. In the first attack, we begin with recovering the underlying error values of the ciphertexts and then use them to recover the secret key. In the second attack, we directly recover the secret key in a bit-by-bit manner. We explain the idea of the attack wlog. when the client intends to perform homomorphic NAND computations on the cloud. We choose the NAND gate as it is a universal gate. However, our attack would work irrespective of the gate being computed.

5.1 Attack 1: Error and Key Recovery Attack on TFHE and FHEW

In this section, we show how an adversary can recover the secret key by first recovering the errors from $\approx n$ ciphertexts. We have already shown in section 2.1 that once the underlying errors are recovered, the system of approximate equations can be turned into a system of exact equations which can then be trivially solved. However, the server also needs to recover the corresponding plaintext message along with the errors since the final ciphertexts are of the form $b = \mathbf{a} \cdot \mathbf{s} \pm e + m$. Thus the server not only has to make a guess about the sign of the underlying error but also about the underlying message. We note that the FHE schemes discussed in this paper perform bit-wise encryption of the plaintext messages and then perform homomorphic operations on those single-bit ciphertexts. More precisely, each ciphertext received at the server is either an encryption of '0' or an encryption of '1'. Therefore, given a ciphertext, the original plaintext value would be in binary. Thus the server can make a random or informed guess about the plaintext message and then rely on the reaction of the client to know whether its guess was correct or not. For balanced gates like XOR or XNOR, i.e., gates for which both outcomes are equally likely or $Pr[m_r = 0] = Pr[m_r = 1] = \frac{1}{2}$, the server has to rely on a random guess. On the other hand, the server can make an informed guess for biased gates where the probability of one message is higher than the probability of other. For example, as per the truth table of NAND gate (as shown in Fig. 5(A)), 75% of times the result of the computation would turn out to be 1. In short, given two ciphertexts C_{x_1} and C_{x_2}, corresponding to two unknown and uniformly chosen plaintext bits x_1 and x_2, the output of the NAND operation between C_{x_1} and C_{x_2} has a 0.75 probability of being 1. Thus we target encryption of 1 when computing a NAND gate as we will have a higher chance of receiving a reaction from the client. Such biases exist for other gates as well. For example, in the case of NOR gate, the result of the computation would turn out to be 0 in 75% of times. Thus in case we target NOR instead of NAND, we will target encryption of 0 instead of encryption of 1.

Elimination of probable choices While the malicious server could perturb the output ciphertexts to instigate a reaction from the client, there exist two major challenges that the server needs to deal with. ① knowledge of the plaintext value for the corresponding ciphertext and ② sign of the actual error. Now, given two ciphertexts C_{x_1} and C_{x_2} from the client, let the NAND output be denoted as C_r. As per the truth table, C_r could be either encryption of 0 or 1, denoted by C_r^0 or C_r^1, respectively. Now, following the strategy of introducing perturbations (discussed previously), the server assumes the underlying error to be $+ve$ and thus adds the error term $e_{diff} = +e_{th} - e_{min}$ into the computed ciphertext C_r. Depending on the input plaintext (r) of the perturbed ciphertext, one of the following four conditions will take place.

❶ **$r = 0$, sign $= +ve$:** The perturbed ciphertext is $C_r^0 + e_{diff}$ with the actual error being positive $(+e_r)$ and underlying plaintext is 0. As the original error was positive, the decryption of C_r^0 will result in 1. However, since the original plaintext was 0 and the decrypted one at the client's end is 1, the client will inform the server regarding the incorrect computation. Therefore, this particular combination ensures a *feedback from the client.*
❷ **$r = 0$, sign $= -ve$:** The perturbed ciphertext is $C_r^0 + e_{diff}$ with the actual error being negative (-e_r) and underlying plaintext is 0. In this case, the decryption will result in 0, since the overall error after perturbation will still remain within the error threshold $+e_{th}$. Therefore, the *client will not provide any feedback* in this case as the decrypted output matches with the expected result for the client.
❸ **$r = 1$, sign $= +ve$:** The perturbed ciphertext is $C_r^1 + e_{diff}$ with the actual error being positive $(+e_r)$ and underlying plaintext is 1. We note that in this case, the decrypted result would be 0 since the perturbed ciphertext was encryption of 1 with a $+ve$ error, thereby essentially flipping the result. Therefore the client

decrypts the result as 0 but the expected outcome was 1, thereby *sending feedback to the server* for the incorrect result.
❹ **$r = 1$, sign $= -ve$:** The perturbed ciphertext is $C_r^1 + e_{diff}$ with the actual error being negative (-e_r) and underlying plaintext is 1. The original error being $-ve$, the final result after decryption does not exceed the threshold $+e_{th}$. Therefore, it would *not generate feedback* from the client since the decrypted result matches the expected result.

Considering r as the expected plaintext, sgn as the sign of the error, and R denoting whether the feedback is received from the client, we record the different combinations of these events from the above-mentioned four cases. Fig. 5(B) shows the record of all possible combinations where sgn is considered as 0 on the error being $+ve$ and 1 on $-ve$. Likewise, R is set as 1 on receiving feedback from the client, and 0 otherwise. Since the server relies on the feedback from the client as a signal for determining the effect of the error, we strictly focus on cases 1 and 3, or more precisely, 1^{st} and 3^{rd} rows in the table shown in Fig. 5(B). We observe that the server receives feedback only when the sign of the error is $+ve$ and does not receive feedback when the error is $-ve$. Thus presence or absence of feedback from the client leaks the sign of the error with probability 1.

Recovering the plaintext value To recover the underlying plaintext message, we introduce another perturbation in the original ciphertext C_r. In the case of TFHE, we simply subtract 2μ, where $\mu = 2^{29}$, from the ciphertext which causes the underlying plaintext message to flip from 1 to 0 while keeping 0 to remain the same. This follows from the decryption function, `approxPhase`(C_r), which represents the sign bit of the underlying plaintext. Originally, $b = \mathbf{s} \cdot \mathbf{a} + \mu + e$ corresponds to encryption of 1, which implies, $b - \mathbf{s} \cdot \mathbf{a} = \mu + e$. When perturbed to $b^* = b - 2\mu$, we have $b^* = -\mu + e$. Here, assuming a small e, we have a flip in the sign bit, thereby transforming μ to $-\mu$. On the other hand, for an encryption of 0, we have $b = \mathbf{s} \cdot \mathbf{a} - \mu + e$, implying, $b - \mathbf{s} \cdot \mathbf{a} = -\mu + e$. Next it is perturbed to $b^* = b - 2\mu = -3\mu + e$. Thus, the sign bit remains $-ve$ in both cases, which corresponds to a decryption of 0. The client will send feedback in the first case as it was expecting 1 whereas it received 0. On the other hand, the client will simply accept the message in the second case as it was expecting a 0 and it received a 0. *This observation reveals the underlying plaintext message to be* 1 *(in case of reaction) or* 0 *(in case of no reaction).*

In the case of FHEW, we obtain a new ciphertext C_r' by performing the operation $HomAND(C_r, HomNOT(C_r))$. The obtained ciphertext C_r' will always be an encryption of 0 irrespective of whether C_r is an encryption of 1 or 0. Similar to TFHE, the client will send feedback in the first case as it was expecting 1 whereas it received 0. On the other hand, the client will simply accept the message in the second case as it was expecting a 0 and it received a 0. This observation reveals the underlying plaintext message to be 1 (in case of reaction) or 0 (in case of no reaction). We would like to emphasize that the FHEW library [32] does not allow homomorphic gate evaluations on a pair of related ciphertexts, where both the inputs are either the same or one is the complement

Algorithm 1 Error Recovery using Binary Search

```
$e_{th}$ := positive error threshold
$e_{min}$ := minimum bound of error
$e_{max}$ := maximum bound of error
$c$ := ciphertext with the original error $e_r$
$start \leftarrow e_{min}$
$end \leftarrow e_{max}$
$e_{temp} \leftarrow 0$
function getErrorPositive($c$, $start$, $end$)
    if $start == end - 1$ then return $e_{temp}$
    else
        $mid \leftarrow \lfloor \frac{start+end}{2} \rfloor$
        $e_{diff} \leftarrow e_{th} - mid$
        $c \leftarrow c + e_{diff} = \mathbf{a} \cdot \mathbf{s} + x_r + e_r + e_{diff}$
        $feedback \leftarrow CVO(c)$
        $c \leftarrow c - e_{diff} = \mathbf{a} \cdot \mathbf{s} + x_r + e_r + e_{diff} - e_{diff}$
         $= \mathbf{a} \cdot \mathbf{s} + x_r + e_r$
        if $feedback$ = "correct decryption" then
            $e_{temp} \leftarrow mid$
            getErrorPositive($c$, $start$, $mid$)
        else
            getErrorPositive($c$, $mid$, $end$)
        end if
    end if
end function
```

of the other. However, this validation is performed over the server side as part of the homomorphic gate evaluation and thus can be simply disabled.

Coming back to our attack, *we will target only those ciphertexts whose underlying plaintext message r is* 1 *and the underlying error value is* $+ve$. We target an encryption of 1 to leverage the biasness of NAND gate towards 1. With this combination of knowledge about the sign of the error and the underlying plaintext message, the adversary launches its final phase of the attack to recover the error value and then the secret key.

Recovering the error value In this section, we show how an adversary can launch a key recovery attack by setting $e_{min} = 0$ and $e_{max} = +e_{th}$, which encompasses the entire range of possible positive error values. Once done, the adversary proceeds to use active perturbations in the computed ciphertext result and iteratively sends perturbed ciphertexts to the client, while awaiting its reaction. In the previous paragraphs, we explained how the adversary can uniquely ascertain the plaintext message and the corresponding error's sign ($+ve$ or $-ve$) of the homomorphically computed ciphertext by carefully introducing additional error and making just two queries to the client for a particular ciphertext. The final error in the ciphertext result after the introduction of the perturbation can be computed as $e' = e_r + (e_{th} - e_{min})$ where e_r, e_{th}, e_{min} are the original error in the computed ciphertext, positive error threshold for decryption and minimum error bound, respectively. With the knowledge of the error sign, underlying plaintext message, and a range of errors, the server now recursively perturbs the originally computed ciphertext by changing the amount of additional error and sending it back to the client for checking its reaction. The overall process for exact error recovery is shown in Algorithm 1. We propose a recursive binary-search-based approach to introduce different perturbations in the original ciphertext. The central idea is that given two bounds e_{min} and e_{max}, we first determine whether the error lies closer to the e_{min} or e_{max}. This can be found out using the same idea that we used to determine the sign of the error. The variables *start* and *end* are first initialized with e_{min} and e_{max} respectively. The first condition we check is if *start* becomes equal to $end - 1$, which implies that there is only one error value left in the range, which will be the original error e_r [3]. Otherwise, we compute a term *mid* as the mid-point of the range $[start, end]$. Following the notion of binary search, our objective is to recursively divide the range into half and ascertain whether the e_r lies in the first or second half. We calculate the error term to be added as $e_{diff} = e_{th} - mid$. This error term e_{diff} is then added to the original ciphertext c. The idea is that if the error lies to the right of *mid* on the error number line (refer Fig. 4), then the addition of this error term e_{diff} would make the overall error $(e_r + e_{diff})$ to cross the positive threshold e_{th}. In such a case, the client experiences a decryption failure and reverts with feedback to the server. On receiving the feedback, the server can understand that the actual error e_r lies between *mid* and *end*. However, if the error e_r lies to the left of *mid*, then the addition of the term e_{diff} would still not cross the error threshold e_{th}. Quite obviously, the client would successfully decrypt the ciphertext and thus will not send any feedback. Here again, on *non-receival of feedback*, the server would understand that the error e_r lies between *start* and *mid*. Therefore, similar to the working process of binary search, the server can eliminate half of the error space on every iteration and gradually progress toward the actual error. Therefore, the output of the algorithm is the actual error e_r of the ciphertext.

Recovering The Secret Key Once the error is recovered for each ciphertext, the server can trivially retrieve the secret key using Gaussian Elimination [46]. The number of such ciphertexts required to create the system of equations depends on the size of the key. For example, if the key size is k bits, one will need at least k ciphertext with correct error values for solving the equations and retrieving the key. We note that the number of ciphertexts required to launch the attack is in the order of the size of the key, more precisely, $\Omega(k)$. Fig. 6 shows the entire end-to-end process of our attack.

Why this is not a CCA1 attack? We would like to clarify that in the case of the plaintext recovery step, even though the feedback from the client

[3] since we are considering the entire range of error, the recovered error will always be correct

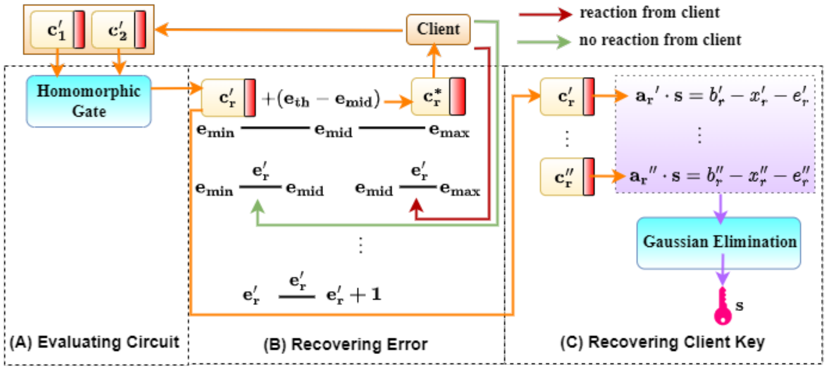

$\mathbf{c}'_1$
$\mathbf{c}'_2$
Client
reaction from client
no reaction from client
Homomorphic Gate
$\mathbf{c}'_\mathbf{r}$ $+(\mathbf{e}_{\mathbf{th}} - \mathbf{e}_{\mathbf{mid}})$
$\mathbf{c}^*_\mathbf{r}$
$\mathbf{e}_{\mathbf{min}}$
$\mathbf{e}_{\mathbf{mid}}$
$\mathbf{e}_{\mathbf{max}}$
$\mathbf{e}'_\mathbf{r}$
$\mathbf{e}'_\mathbf{r} + \mathbf{1}$
$\mathbf{c}'_\mathbf{r}$
$\mathbf{c}''_\mathbf{r}$
$\mathbf{a_r}' \cdot \mathbf{s} = b'_r - x'_r - e'_r$
$\mathbf{a_r}'' \cdot \mathbf{s} = b''_r - x''_r - e''_r$
Gaussian Elimination
s
(A) Evaluating Circuit
(B) Recovering Error
(C) Recovering Client Key

Fig. 6: End-to-End attack process on FHEW and TFHE showing (A) how homomorphic gate is evaluated on client's ciphertexts, (B) how the reaction from the client is used to reduce search space which ultimately leads to recovery of error, and (C) how recovered errors along with original ciphertexts are used to form a system of exact equations which are then solved using Gaussian Elimination to recover client key.

inadvertently leaks the underlying plaintext value of the targeted ciphertext, the client is still behaving as a verification oracle and not as a decryption oracle. The reason is that, since we only have two possibilities for the plaintext values due to the message space of the targeted schemes being binary, feedback or the lack of it from the client confirms that our guessed plaintext bit was correct or not (which implies that the other bit is correct). For schemes where the plaintext space is non-binary (say integers), feedback from the client will only reveal that the sent value was incorrect and will not reveal the correct value. Moreover, during the error recovery step, the client completely behaves as a verification oracle and not as a decryption oracle. The reason is that when the adversary sends a perturbed ciphertext to the client but does not receive feedback, it cannot differentiate between a successful perturbation and an unsuccessful one. To put this as an example, say the original ciphertext c was an encryption of 1 and it became c^* after adding an error value e. The client decrypts this perturbed ciphertext and does not provide feedback. At this point, the adversary is unsure of whether the added perturbation successfully caused the message to flip to 0 but the client was expecting 0 in the first place, or the added perturbation was unsuccessful but the client was expecting 1. Thus the adversary gains no additional information even after having access to a verification oracle. Hence this attack falls under the IND-CVA1 security notion and not under the IND-CCA1 security notion.

5.2 Attack 2: Exploiting Key Distribution to Recover Secret Key

In this section, we show how an adversary can recover the entire secret key of both schemes in a bit-by-bit fashion. Similar to our previous attack, we target the decryption stage which, upon receiving a ciphertext $c = (\mathbf{a}, b)$ encrypted under a secret key $\mathbf{s}$, first computes the phase of the message as $\phi = b - \mathbf{a} \cdot \mathbf{s} = m \pm e$. It is followed by rounding off this phase to the nearest encoded message m, which is then decoded to output the original plaintext message (we refer the readers to section 2.3 for more details). The intuition behind this attack is that adding an error value into a coefficient a_j of vector $\mathbf{a}$ will have no effect on the decryption result if the corresponding secret bit s_j is 0 (zero), as the added error will be canceled out. On the other hand, the added error value might affect the decryption result if the corresponding secret bit s_j is 1 or -1, as the added error value might cause the overall error to cross the threshold. Whether this error crosses the threshold or not depends upon both the error value added to the coefficient a_j and the sign of the error value e already present in the ciphertext. Since the goal of this attack is to flip the underlying message, it can be carried out by targeting any ciphertext irrespective of the underlying plaintext message and the sign of the original error value.

Intuition behind the attack The ciphertext in both TFHE and FHEW is of the form $c = (\mathbf{a}, b)$ where $b = \mathbf{a} \cdot \mathbf{s} + m \pm e$ and m is the underlying encoded message. We can expand the representation of b as

$$b = \sum_{i=1}^{k} (a_i * s_i) + m \pm e \tag{1}$$

where k is the length of the secret key. During decryption, the phase of the message is computed as

$$\phi = b - \sum_{i=1}^{k} (a_i * s_i) = m \pm e \tag{2}$$

The attacker chooses a value $j \in \{1, 2, \cdots, k\}$, either randomly or in some specific order, and adds an error value e' to the coefficient a_j to turn it to $a'_j = a_j + e'$. The new phase of the underlying message can now be written as

$$\phi' = b - \sum_{i=1}^{j-1} (a_i * s_i) - (a'_j * s_j) - \sum_{i=j+1}^{k} (a_i * s_i)$$

$$\phi' = \sum_{i=1}^{k} (a_i * s_i) + m \pm e - \sum_{i=1}^{j-1} (a_i * s_i) - (a_j * s_j) - (e' * s_j) - \sum_{i=j+1}^{k} (a_i * s_i)$$

$$\phi' = \sum_{i=1}^{k} (a_i * s_i) - \sum_{i=1}^{k} (a_i * s_i) + m \pm e - (e' * s_j)$$

$$\phi' = m \pm e - (e' * s_j)$$

Now, based on the value of the corresponding secret bit s_j, the following three cases arise:

❶ $\mathbf{s_j = 0}$: In this case, the value of ϕ' reduces to

$$\phi' = m \pm e - (e' * 0) = m \pm e = \phi$$

One can clearly see that $s_j = 0$ cancels out the effect of the added error value in a_j, thus the phase of the message does not change. The ciphertexts decrypt correctly and the client does not react.
❷ $\mathbf{s_j = 1}$: In this case, the value of ϕ' reduces to

$$\phi' = m \pm e - (e' * 1) = m \pm e - e' = m + e''$$

where $e'' = -e' + e$, if e is $+ve$, or $e'' = -e' - e$, if e is $-ve$. At this point, whether e'' will cross the threshold or not will depend on the choice of e'. The adversary may choose either an arbitrary value for e' or set $e' = e_{th}$. Choosing the value of e' arbitrarily is not a good choice, since it might generate unpredictable results. On the other hand, which e_{th} to choose (either $+e_{th}$ or $-e_{th}$) will depend on the sign of the original error e, which the adversary does not know. Thus similar to our previous attack, the adversary makes a guess about the sign of the original error.
Assuming the sign of e to be $-ve$: The adversary sets the value of e' to $-e_{th}$. If the original error e is indeed $-ve$, then $e'' = -e_{th} - e < -e_{th}$ which results in incorrect decryption causing the client to react. On the other hand if the original error e is $+ve$, then $e'' = -e_{th} + e > -e_{th}$ which results in correct decryption. The client will not react in this case.
Assuming the sign of e to be $+ve$: The adversary sets the value of e' to $+e_{th}$. If the original error e is indeed $+ve$, then $e'' = +e_{th} + e > +e_{th}$ which results in incorrect decryption causing the client to react. On the other hand if the original error e is $-ve$, then $e'' = +e_{th} - e < +e_{th}$ which results in correct decryption. The client will not react in this case.
❸ $\mathbf{s_j = -1}$: In this case, the value of ϕ' reduces to

$$\phi' = m \pm e - (e' * -1) = m \pm e + e' = m + e''$$

where $e'' = e' + e$, if e is $+ve$, or $e'' = +e' - e$, if e is $-ve$. Similar to case ❷ above, the adversary will have to choose a value of e' based on whether e is $+ve$ or $-ve$, which again it does not know. The adversary again has to make a guess about the sign of the original error.
Assuming the sign of e to be $+ve$: The adversary sets the value of e' to $+e_{th}$. If the original error e is indeed $+ve$, then $e'' = +e_{th} + e > +e_{th}$ which results in incorrect decryption causing the client to react. On the other hand if the original error e is $-ve$, then $e'' = +e_{th} - e < +e_{th}$ which results in correct decryption. The client will not react in this case.
Assuming the sign of e to be $-ve$: The adversary sets the value of e' to $-e_{th}$. If the original error e is indeed $-ve$, then $e'' = -e_{th} - e < -e_{th}$ which results in incorrect decryption causing the client to react. On the other hand if the original error e is $+ve$, then $e'' = -e_{th} + e > -e_{th}$ which results in correct decryption. The client will not react in this case.

Algorithm 2 Direct Key Recovery

```
$+e_{th}$ := positive error threshold
$-e_{th}$ := negative error threshold
$c$ := target ciphertext with the original error $e$
$r$ := number of iterations to be made
function recoverSecretBit($j$)
    for i := 1; i ¡= r; ++i do
        $c.a[j] \leftarrow c.a[j] - (-e_{th})$
        $feedback \leftarrow CVO(c)$
        $c.a[j] \leftarrow c.a[j] + (-e_{th})$
        if $feedback$ = "incorrect decryption" then return 1 (for both FHEW and TFHE)
        end if
    end for
    for i := 1; i ¡= r; ++i do
        $c.a[j] \leftarrow c.a[j] + (+e_{th})$
        $feedback \leftarrow CVO(c)$
        $c.a[j] \leftarrow c.a[j] - (+e_{th})$
        if $feedback$ = "incorrect decryption" then return $-1$ (for FHEW) or 1 (for TFHE)
        end if
    end for
    return 0
end function
```

Based on the above 3 cases, we make 2 key observations. First, we observe that we never receive a reaction from the client if the targeted secret bit s_j is 0, since it masks the effect of the additional error. Second, we observe that adding $-e_{th}$ causes incorrect decryption only when s_j is 1 and e is $-ve$, for both FHEW and TFHE. Similarly, adding $+e_{th}$ causes incorrect decryption only when e is $+ve$ but s_j is -1 in case of FHEW and s_j is 1 in case of TFHE.

Recovering The Secret Key An adversary can exploit the aforementioned two observations to launch a direct key recovery attack on both FHEW and TFHE. It does so by first making an assumption on both the secret key bit and the sign of e. It initially assumes that $s_j = 1$ and then proceeds to subtract $-e_{th}$ or $+e_{th}$ from a_j based on the assumed sign of error. On the other hand, if it assumes that $s_j = -1$ then it adds $-e_{th}$ or $+e_{th}$ to a_j based on the assumed sign of error. Assuming equal likelihood of all key bits and both signs, the probability that this guess turns out to be correct is $\frac{1}{3} * \frac{1}{2} = \frac{1}{6}$ in case of FHEW and $\frac{1}{2} * \frac{1}{2} = \frac{1}{4}$. On the other hand, the guess may turn out to be wrong with probabilities $\frac{5}{6}$ and $\frac{3}{4}$ in the case of FHEW and TFHE, respectively. If the guess turns out to be wrong then the adversary discards the current ciphertext and proceeds with the next one. For sufficient repetitions r of this assumption, the probability that the guess turns to be wrong for all samples is $(\frac{5}{6})^r$ and $(\frac{3}{4})^r$ for FHEW and TFHE, respectively. On the other hand, the probability that the guess turns out to be

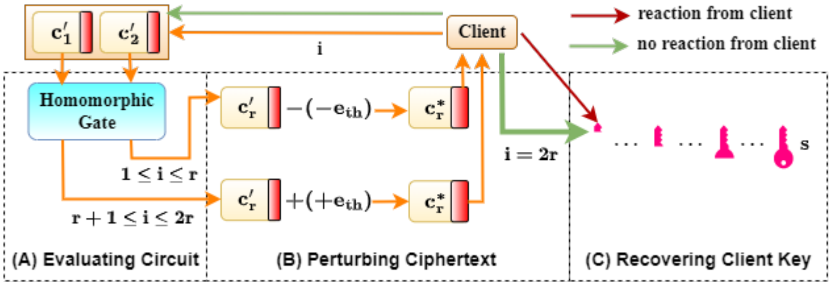


Fig. 7: End-to-End process of our improved attack showing (A) how homomorphic gate is evaluated on client's ciphertexts, (B) how the reaction from the client is used to recover a single bit of the secret key, and (C) how the entire process is repeated to recover the key bit-by-bit. For each secret bit, 'i' goes from 1 to 2r, where 'r' denotes the number of repetitions.

correct for at least 1 sample out of r samples is $1-(\frac{5}{6})^r$ and $1-(\frac{3}{4})^r$ for FHEW and TFHE, respectively. Thus the higher the number of repetitions, the higher the probability to recover the secret bit. Once the guess turns out to be correct, the targeted sample decrypts incorrectly and the client reacts. At this point, the adversary knows that the targeted bit s_j is 1 for both FHEW and TFHE, in case it subtracted $-e_{th}$. Moreover, it knows that the targeted bit s_j is -1 and 1 for FHEW and TFHE, respectively, in case it added $+e_{th}$. Finally, if it does not get a reaction even after $2 \times r$ repetitions, then it knows for sure that the targeted bit s_j is 0. The overall procedure of this attack has been explained in algorithm 2 and shown in Fig. 7.

Why this is not a CCA1 attack? At this point, we would like to reiterate the fact that the aforementioned attack does not depend on the underlying plaintext bit, since the aim of this attack is to flip the plaintext bit irrespective of its value. Thus feedback from the client only reveals that the plaintext was flipped successfully, while a lack of it only reveals that the plaintext is still the same. Moreover, the actual value of the plaintext bit is not revealed during the entire duration of this attack. Thus the client truly behaves as a verification oracle and this attack falls under the IND-CVA1 security notion and not under the IND-CCA1 security notion.

6 Experimental results

In this section, we provide the experimental results of the attack for both FHEW and TFHE. We first show the results of our attack when computing a single gate, which is NAND in our case, to show the feasibility of our attack. We follow this up with a possible implementation of our database example from section 4 and then provide the results for leaking the key in such implementation.

Table 1: Shows the total number of samples perturbed to recover error for the required number of samples, the total number of CVO queries made, the total time required for running the first attack on a NAND gate, and the number of samples that resulted in incorrect decryption, for 5 different, random keys. All times are in mins. and secs.

		key_1	key_2	key_3	key_4	key_5
FHEW	samples perturbed	1376	1444	1541	1509	1464
	CVO queries made	**5043**	**5136**	**5251**	**5202**	**5147**
	time required	1m41s	1m47s	1m53s	1m50s	1m46s
	decryption failure	**2034**	**2101**	**2099**	**2058**	**2076**
TFHE	samples perturbed	1836	1819	1741	1734	1743
	CVO queries made	**22573**	**22584**	**22419**	**22445**	**22455**
	time required	18m45s	18m17s	17m28s	17m30s	17m35s
	decryption failure	**9542**	**9596**	**9577**	**9424**	**9409**

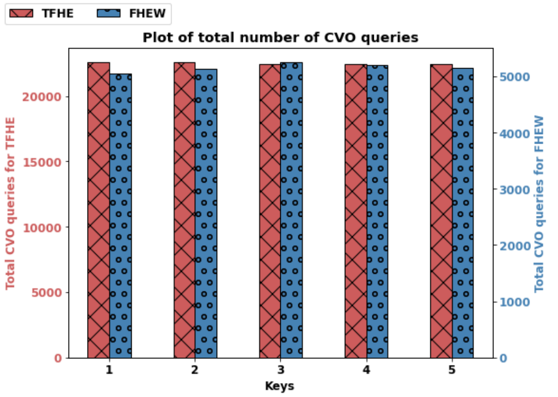


Fig. 8: Plot of the total number of CVO queries made to extract errors and recover 5 different, randomly generated secret keys in the case of TFHE and FHEW, respectively.

6.1 Results of key recovery attack on NAND gate

In our first attack where we recover the error values in the resultant ciphertext, we set the range of errors to be $[0, 63]$ and $[0, 10200547327]$, where 63 and 10200547327 are the $+ve$ error threshold of homomorphic NAND gate for FHEW and TFHE, respectively. During the experiment, we generated 2000 pair of random plaintext bits for both schemes. We encrypted these plaintext pairs using 5 different, randomly generated secret keys. We proceeded to run a homomorphic NAND gate on each of these ciphertext pairs to obtain the corresponding computation result, upon which we ran our attack. Since the dimension of the secret key for FHEW and TFHE is 500 and 630, respectively, we require to recover errors from at least 500 and 630 ciphertexts. However, for certain cases, we observed during experimentation that Gaussian elimination was only able to partially recover the secret key. Thus we increased the number of samples to gen-

Table 2: Shows the total number of CVO queries made, the total time required for our improved attack on a NAND gate, and the number of samples that resulted in incorrect decryption, which is equal to the number of non-zero values in the secret key. We provide values for 5 different, random keys. All times are in mins. and secs.

		key_1	key_2	key_3	key_4	key_5
FHEW	**CVO queries made**	**7379**	**7397**	**7296**	**7313**	**7394**
	time required	7m47s	7m48s	7m43s	7m43s	7m52s
	decryption failure	**247**	**247**	**254**	**255**	**248**
TFHE	**CVO queries made**	**4502**	**4617**	**4624**	**4649**	**4654**
	time required	1m38s	1m41s	1m41s	1m43s	1m42s
	decryption failure	**323**	**312**	**312**	**306**	**307**

erate more equations, which in our case are 510 and 640. We chose these values at random, however, an adversary can empirically find the minimum number of equations eq_{min} for which it can always recover the entire key. Once found, the adversary can perform the perturbations until it successfully recovers errors for eq_{min} number of ciphertexts, after which it can form the system of equations and solve it to retrieve the secret key. The server will anyhow try to keep this number as low as possible so as to also minimize the number of queries while also increasing the probability of successful key recovery. Table 1 shows the count of ciphertexts out of these 2000 samples which we needed to perturb to successfully recover the error values of 510 and 640 ciphertexts, the total number of CVO queries made for the same, the total time required to perform the end-to-end attack and the number of samples that resulted in incorrect decryption. For the first key, we perturbed 1376 and 1836 ciphertexts to recover error values from 510 and 640 ciphertexts for FHEW and TFHE respectively. The overall attack took around 2 minutes and 19 minutes, and required 5043 and 22573 CVO queries in the case of FHEW and TFHE, respectively. It also required 2034 and 9542 decryption failures, which induces the client to react. We required 8 and 33 queries to successfully recover the error from 1 ciphertext for FHEW and TFHE, respectively. We require an average of 5200 and 22000 queries and an average of 2 and 18 minutes to recover errors from 510 and 640 ciphertexts in the case of FHEW and TFHE, respectively, for the 5 keys. Once we recover the original error from ciphertexts, a system of equations is formed which is then solved to recover the entire key. We were able to recover the secret key successfully in all 5 cases. Fig. 8 shows the plot of the total number of CVO queries made to extract errors and recover the secret key in the case of TFHE and FHEW, respectively. We can observe from the plot that the number of queries required to recover errors is almost the same across all 5 cases for both FHEW and TFHE, showing that the number of required queries does not depend on the underlying secret key.

In our improved attack, we set $+e_{th}$ to 63 and 10200547327, and $-e_{th}$ to 64 and 536870911. These are the $+ve$ and $-ve$ error thresholds of the homomorphic NAND gate for FHEW and TFHE, respectively. We generated a stream of

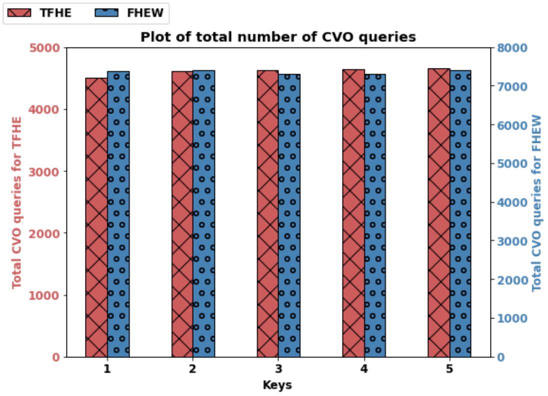


Fig. 9: Plot of the total number of CVO queries made to directly recover 5 different, randomly generated secret keys in the case of TFHE and FHEW, respectively.

random plaintext bits pairs and encrypted them using the same 5 secret keys as above. We proceeded to run a homomorphic NAND gate on each of these ciphertext pairs to obtain the corresponding computation result, upon which we ran our attack. We empirically selected the value of r, i.e., the number of repetitions, to be 11 and 6 for FHEW and TFHE, respectively. We target each resulting ciphertext to recover 1 bit of the secret key. Table 2 shows the total number of CVO queries made to recover the entire secret key, the total time required to perform the end-to-end attack, and the number of samples that resulted in incorrect decryption. We would like to highlight the fact that the number of decryption failures was the same as the number of non-zero secret key bits, i.e. $\sum(s_i) : s_i \neq 0$. For the first key, it took around 8 and 2 minutes and required 7379 and 4502 queries to recover the entire 500 and 630 bits of the secret key for FHEW and TFHE, respectively. We required an average of 7300 and 4300 queries and an average of 8 and 2 minutes to recover the entire secret key across the 5 cases for FHEW and TFHE, respectively. Fig. 9 shows the plot of the total number of CVO queries made and the total number of ciphertexts that failed to decrypt correctly, to directly recover the secret key across both FHEW and TFHE. Similar to the previous attack results, we can observe from the plots that the number of queries required to recover errors is almost the same across all 5 cases for both FHEW and TFHE, showing that the number of required queries does not depend on the underlying secret key.

Fig. 10 shows the plot of the total number of CVO queries made to extract the entire secret key for all 4 and 6 major gates in FHEW and TFHE. The results are shown for a randomly generated secret key. We observe that in the case of FHEW, AND and NAND gates require comparatively less number of queries as compared to OR and NOR to recover the secret key. Similarly, in the case of TFHE, the number of queries are comparatively less for NOR, AND, and

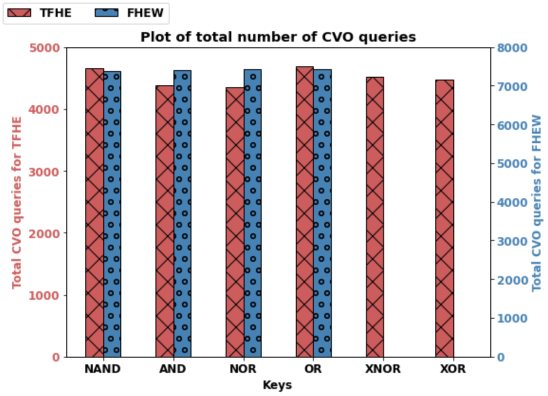


Fig. 10: Total number of CVO queries to directly recover a random secret key across all 4 gates of FHEW and across all 6 gates of TFHE. FHEW does not support XOR and XNOR gates.

XNOR than for XOR, NAND, and OR gates. This implies that an adversary can target the output of any gate to carry out the key recovery attack. Moreover, it can utilize this distribution to select the best gate for which the least amount of queries is required, if it can target multiple gates.

6.2 Results of key recovery attack on encrypted database example

Fig. 11 shows the entire end-to-end scenario of our example of an encrypted database application. We consider a situation where this application is being used by a bank, where the data of the customers has already been encrypted and stored on the server in the form of a database. For simplicity, we assume that the only query that will be performed on this database is a *range* query, which returns all the rows of the queried table where the value of a field lies in a certain range. Whenever the client, which is the bank staff in our case, wants to evaluate such a query, it first encrypts all the parameters such as the table name, the field name, and the range values. It then sends these encrypted values to the server. Upon receiving these values, the server feeds them in the *query processing* circuit which homomorphically evaluates the known query using the encrypted inputs. The circuit outputs a block of ciphertexts that is an encryption of the contents of the selected rows. Since the server does not know the table on which the query has been processed, it does not know the size (in terms of the number of ciphertexts) of the rows being returned. Also, since the output rows are the result of a series of homomorphic operations on the input rows and the query parameters, the output rows will look different than their corresponding input rows in terms of ciphertexts, even though the underlying plaintext values will remain the same. For example if an input row consists of the ciphertexts $\{C_{11}, C_{12}, \cdots, C_{1r}\}$, the homomorphic operations outputs the same rows as $\{C'_{11}, C'_{12}, \cdots, C'_{1r}\}$ where

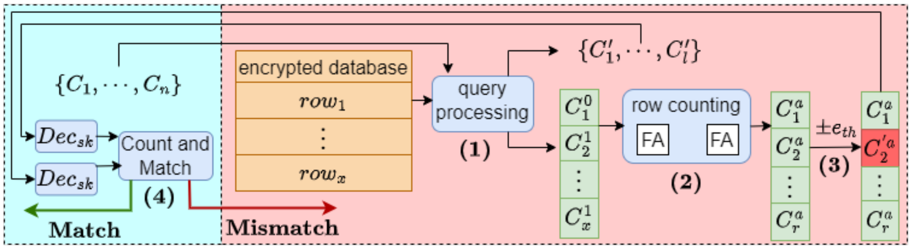


Fig. 11: Our PIR example works by (1) evaluating query on an encrypted table to output matching rows along with a vector of matched rows, (2) counting the number of matched rows, (3) perturbing one ciphertext corresponding to matched rows, and (4) client decrypting and matching the row counts. If the count matches then the client accepts the result, otherwise asks the server for re-computation. Here query processing circuit and row counting circuits are part of the function f, while the count and match circuit is part of the function g.

$C_{1i} \neq C'_{1i}, 1 \leq i \leq r$. This prevents the server from looking up these output values in the stored table to find which rows have been selected. Thus the server neither knows which rows have been selected nor does it knows the count of such rows.

Coming back to our query processing circuit, it also outputs a vector whose elements are encryptions of 1 or 0 that signifies whether the corresponding row has been selected or not. The server sends this vector as an input to the *row counting* circuit which is built using a combination of full adder circuits. This circuit homomorphically adds up all the entries of this vector and outputs another vector of encryptions of 0 and 1. The plaintext entries of this new vector, when combined, reveal the total number of rows that satisfied the query. The server chooses one of the entries of this vector and adds a perturbation to it. Finally, it sends back this perturbed vector and the block of ciphertexts containing the encrypted rows to the client. Upon receiving these values, the client first decrypts and then feeds them into the *count and match* circuit. The circuit counts the number of returned rows and matches this value with the returned count. If the count matches, then the client assumes that the query was processed correctly and it has received all the required rows. On the other hand, if there is a mismatch in the count then the client assumes that either the query was not evaluated correctly or some of the rows were lost during transmission. Since the client cannot differentiate between these two cases, it asks the server to recompute the query. The query processing and row counting circuits form the function f that is evaluated homomorphically by the server, while the count and match circuit g acts as the application-level constraint that runs on the client side.

For demonstrating our attack on the encrypted database, we take a toy example where the table being queried has only 3 rows. We assume that this application has been built using the TFHE library [25] as it is faster as com-

Table 3: Shows the total number of CVO queries made and total time required for the improved attack on a full adder circuit implemented using TFHE and the number of samples that resulted in incorrect decryption, which is equal to the number of non-zero values in the secret key. We provide values for 5 different, random keys. All times are in mins. and secs.

	key_1	key_2	key_3	key_4	key_5
CVO queries made	**4325**	**4392**	**4464**	**4525**	**4474**
time required	7m50s	8m1s	8m6s	8m15s	8m12s
decryption failure	**323**	**312**	**312**	**306**	**307**

pared to the FHEW library [32]. In this toy example, the row counting circuit was implemented as a single full adder constructed using basic gates of XOR, AND, and OR. The input to this circuit was taken as a vector of 3 ciphertexts. To simulate the entries for this vector, we generated a stream of random plaintext bits and encrypted them using 5 different, randomly generated keys. We fed these simulated rows into the full adder circuit to obtain another vector of size 2. Once obtained, we first decrypted these values to obtain the correct result which signifies the actual number of rows returned by the query. We then proceeded to add perturbation to the ciphertext obtained as the output of the XOR (sum) gate. The reason we chose the output of the XOR gate and not of the OR (carry) gate is that in the actual implementation of the row counting circuit, the output will be a result of a series of full adders where the output of the XOR gate forms a part of the output while the output of the OR gate becomes the carry input of the next adder. Once perturbed, we decrypted this entry and matched the new value with the old value that was obtained before perturbation. If these values matched, then the application-level constraint succeeds and we accept the result. On the other hand, if these values do not match then the application-level constraint fails and we ask for a re-computation.

Since the application is assumed to be built using TFHE and we have two possible attack vectors for the same, we provide the results for our improved attack. In the real-life setting, an adversary will want to go for the better attack vector to minimize its chances of getting caught and also to recover the key as quickly as possible. Table 3 shows the total number of CVO queries made to successfully recover the entire secret key of size 630. For the first key, the overall attack took around 8 minutes and required 4325 CVO queries. We required an average of 4400 queries and an average of 8 minutes to recover the entire secret key across the 5 cases.

We performed our attacks using a Desktop computer running Intel Xeon Silver 4210R @ 2.4GHz powered by Ubuntu 18.04. To solve the system of equations, we ran Gaussian Elimination from SageMath9.0 and Python 3.8 to recover the entire secret key. However, our attack is not device specific and it can be carried out using any system.

7 Need for verifiable FHE

Key recovery attacks on existing FHE schemes, including ours, target the lack of integrity which a malicious server can exploit to tamper with the data associated with a computation. When the client decrypts such tampered ciphertext, it provides feedback to the server in the form of a re-computation request in case the result fails to satisfy the application-level constraint. Thus it has become necessary to incorporate integrity checks in order to prevent such attacks. However, as stated in section 4.3, ciphertext integrity is not ensured at the primitive level but at the application level. Moreover, there is a recent rise in a new class of works in the form of verifiable FHE that provides both confidentiality and correctness with a single technique. These works are based on providing proof of correct computation along with its result. The associated proof helps the client to verify whether the provided result is indeed generated by performing the requested computation on the actual inputs or whether the server has deviated from the computation. We provide an overview of existing verifiable FHE techniques and their limitations along with the applicability of reaction attacks in the presence of these techniques.

7.1 Existing verifiable FHE techniques and their limitations

There have been recent developments in the domain of verifiable FHE, which uses either cryptographic techniques such as Message Authentication Codes (MACs) and Zero-Knowledge Proof (ZKP) systems, or tamper-resistant hardware environments such as Trusted Execution Environments (TEE). However, since generating normal MACs require a secret key that remains in possession of the client, the server cannot generate them on its end for the intermediate or final results of homomorphic computations. Recent works have proposed the construction of homomorphic MACs, which provides the server with the ability to homomorphically generate valid tags for ciphertexts without requiring the client's secret key. [38] introduced the notion of fully homomorphic message authenticators, which were later used by [22,23,35,62] to provide verifiable computation. However, all these schemes are limited to the evaluation of polynomial arithmetic which forms only a subset of FHE computations. [16] and [36] proposed the idea of using SNARK (Succinct Non-interactive ARguments of Knowledge) to provide verifiable computation. However, they used specialized SNARKs that are meant to be used on specific rings. Moreover, their techniques were more focused on the correctness of the computation and not on the confidentiality aspect of such computations. Recently [64] introduced the notion of maliciously-verifiable FHE schemes and showed an instantiation of their scheme using a generic SNARK [37] that works over multiple rings. However, all these schemes have only been shown on RLWE-based schemes that do not require ciphertext-maintenance operations such as bootstrapping. Moreover, the underlying mathematics of ZKP systems are not compatible with these ciphertext-maintenance operations, as they involve either ring-switching or non-ring operations [64], and thus are not compatible with the schemes that we target in this paper. Finally, [60] showed how secure enclaves such as TEEs can be used to ensure the honest execution of a function over user-generated inputs. Such hardware provides proof of correct computation in the form of hardware attestation certificates. However, such secure enclaves suffer from both low memory and computation power as compared to the untrusted hardware, and thus are not enough for the memory-hungry FHE ciphertexts and computation-hungry operations [64]. [48] proposed the usage of one-digit checksum, however it does not work with the existing FHE schemes. Finally [54] proposed the usage of blind hash, however it is incompatible with FHE schemes where the plaintext space is binary, such as FHEW and TFHE.

7.2 Applicability of reaction attacks in the context of verifiable FHE

While the existing works on verifiable FHE provide efficient techniques for generating and verifying the proofs, they lack in clearly stating the response of the client in case the proof turns out to be false. In such a case, the client might still ask the server for a free re-computation, thus invalidating the whole point of providing the proof in the first place. Moreover, instead of relying on the application-level constraint, the server can rely on the failure of proof verification by intentionally providing false proof along with a correct answer to instigate a reaction from the client. On the other hand, the client might also abort further communication with the server. However, this itself is quite an impractical solution. In case the client decides to permanently abort further communications with the server, then it has to avail service of another server which would incur a huge cost and time. Moreover, in certain situations, even temporarily aborting further communication is not a viable solution. Considering our banking example, aborting the communication with the server for even a few seconds will lead to a huge loss to the bank. Thus it becomes important to accompany such verification techniques with some repairing mechanisms, which can help the client to locally rectify the incorrect results instead of asking for a re-computation. With such mechanisms in place, the client can further utilize an actively malicious server for its computational requirements.

8 `Verify` - then - `Repair` or `React` (`vr`^{2}FHE)

We propose a method that the client can use to verify whether the result it received is correct and has not been tampered with, and can use it to repair the result locally if it turns out to be incorrect. Our method is based on the concept of the Merkle tree, where we generate the hash of the intermediate results in a bottom-up manner, as shown in Fig. 12(a). Our proposed method is divided into two stages, namely, computation, and verification. We assume the circuit is logically divided into multiple levels, where the gates on level $i + 1$ can only be evaluated when all the gates at level i have been evaluated. The gates at level 1 operate on only client inputs, while the gates from level 2 and onward operate on intermediate results generated at some lower level, and may occasionally include

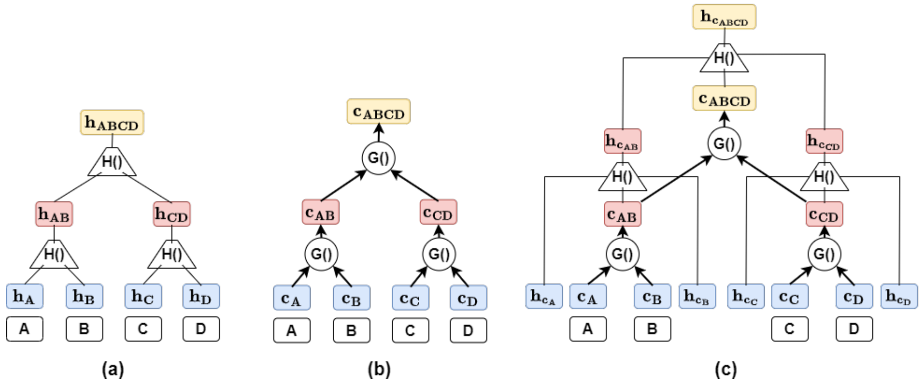


Fig. 12: Shows (a) how a Merkle tree is generated, (b) how a Boolean circuit is evaluated, and (c) how our countermeasure combines the concept of Merkle tree with a Boolean circuit. The inputs are shown in blue, intermediate results are shown in red, and outputs are shown in yellow.

the client inputs. The gates at the final level L generates either all or some of the final outputs. The server sends the final output to the client for decryption only when the final level has been evaluated and all outputs are ready. Fig. 12(b) shows a sample Boolean circuit and the order in which each gate is evaluated.

In existing FHE libraries, including FHEW and TFHE, the client encrypts its inputs and sends the generated ciphertexts to the server. On the other hand, in our proposed method the client additionally sends the hashes of these ciphertexts that are generated using some collision-resistant hash function. During the evaluation of the requested function, the server computes a series of gates on the input ciphertexts which are either provided by the client or are a combination of client inputs and intermediate results of previous computations. Moreover, each gate computation generates some intermediate or output ciphertexts. The server also computes the hash of the output ciphertexts of each gate by evaluating a hash function on a tuple consisting of this ciphertext value along with the hashes of the input ciphertexts of this gate. For example, the hash of an intermediate ciphertext c_{AB}, which is the result of a gate computation on some ciphertexts c_A and c_B, will be generated as

$$h_{c_{AB}} = Hash\{c_{AB}, h_{c_A}, h_{c_B}\}$$

where $Hash\{\cdot, \cdot, \cdot\}$ is a collision-resistant hash function agreed upon by the client and server, and h_{c_A} and h_{c_B} are the hashes corresponding to ciphertexts c_A and c_B, respectively. One can clearly see here that the hash of a ciphertext generated at some level k depends upon the hash of some ciphertexts generated at or below level $k-1$, whose hashes themselves depend upon the hashes generated at further lower levels. Thus tampering with a ciphertext at any level k will change its hash, due to the second-preimage-resistant property of the used hash function. This

Algorithm 3 Client Side Verification

```
function VERIFY($c_{ABCD}$, $h_{c_{ABCD}}$, $c_{AB}$, $c_{CD}$, $h_{c_{AB}}$, $h_{c_{CD}}$, $h_{c_A}$, $h_{c_B}$, $h_{c_C}$, $h_{c_D}$)
    $h'_{c_{ABCD}} \leftarrow Hash\{c_{ABCD}, h_{c_{AB}}, h_{c_{CD}}\}$
    $h'_{c_{AB}} \leftarrow Hash\{c_{AB}, h_{c_A}, h_{c_B}\}$
    $h'_{c_{CD}} \leftarrow Hash\{c_{CD}, h_{c_C}, h_{c_D}\}$
    if $h'_{c_{ABCD}} = h_{c_{ABCD}}$ and $h'_{c_{AB}} = h_{c_{AB}}$ and $h'_{c_{CD}} = h_{c_{CD}}$ then        ▷ Verify
        $answer \leftarrow Dec_{sk}(c_{ABCD})$
        if $answer \neq$ "correct answer" then
            $c'_{ABCD} \leftarrow HomGate(c_{AB}, c_{CD})$
            if $c'_{ABCD} \neq c_{ABCD}$ then                                    ▷ Repair
                "malicious tampering"
                $answer \leftarrow Dec_{sk}(c'_{ABCD})$
            else                                                             ▷ React
                "accidental decryption failure; ask for re-computation"
            end if
        else
            "accept answer"
        end if
    end if
end function
```

will further result in changing the hashes of ciphertexts at level $k+1$ and above since their hashes depend on the hashes at lower levels.

Once the evaluation is completed, the server sends the following pieces of information to the client:

1. The resulting ciphertext c_{ABCD} generated at level L along with its corresponding hash $h_{c_{ABCD}}$.
2. The ciphertexts c_{AB} and c_{CD} generated at level $L-1$ along with their corresponding hashes $h_{c_{AB}}$ and $h_{c_{CD}}$, respectively.
3. Only the hashes h_{c_A}, h_{c_B}, h_{c_C} and h_{c_D} that corresponds to the ciphertexts c_A, c_B, c_C and c_D generated at level $L-2$. Here c_A and c_B generated c_{AB}, while c_C and c_D generated c_{CD}.

Among these values, c_{ABCD} forms the output of the requested computations while the other values form the auxiliary information that aids the client in identifying whether the output was tampered with or not. Upon receiving the output ciphertext and the accompanying auxiliary data, the client first generates the hash of the output ciphertext c_{ABCD}, along with the hashes of the intermediate ciphertexts c_{AB} and c_{CD} as

$$\begin{aligned} h'_{c_{ABCD}} &= Hash\{c_{ABCD}, h_{c_{AB}}, h_{c_{CD}}\} \\ h'_{c_{AB}} &= Hash\{c_{AB}, h_{c_A}, h_{c_B}\} \\ h'_{c_{CD}} &= Hash\{c_{CD}, h_{c_C}, h_{c_D}\} \end{aligned}$$

It then checks whether $h'_{c_{ABCD}} = h_{c_{ABCD}}$, $h'_{c_{AB}} = h_{c_{AB}}$ and $h'_{c_{CD}} = h_{c_{CD}}$ or not. If they do not match, the client assumes that the received data was modified due to some transmission error and it asks the server to re-transmit the data. Since the server wants the hashes to match so that the client proceeds with decrypting the received ciphertext, it will honestly generate these hashes. The presence of feedback at this point is due to the hashes not matching, and not because the decryption was incorrect. On the other hand, if the hashes match, the client proceeds with the usual decryption. If the decryption result is correct, the client simply accepts the message and provides no feedback. On the contrary, if the decryption result turns out to be incorrect, it might raise suspicion in the client since it knows that the output ciphertext was not modified due to a transmission error. However, there is still a very small probability of decryption failure [27] even if the ciphertext was not perturbed.

To differentiate between an honest and a malicious decryption failure, the client re-computes the output ciphertext c'_{ABCD} using the ciphertexts c_{AB} and c_{CD}. It already knows that it has received correct values of c_{AB} and c_{CD}, due to the previous hash check. Once generated, it checks whether $c'_{ABCD} = c_{ABCD}$ or not. If $c'_{ABCD} \neq c_{ABCD}$, then the client knows that it has received a tampered ciphertext, and thus the decryption failure was not accidental. However, the client need not request a re-computation as it has already generated the correct ciphertext c'_{ABCD} locally which it can decrypt to obtain the correct result. On the other hand, if $c'_{ABCD} = c_{ABCD}$ then the client knows that the incorrect decryption was accidental and not intentional. In such a case, it asks the server for a re-computation. Since this re-computation request is due to an accidental decryption failure, it does not leak any information to the server. Fig. 12(c) shows the overall process of circuit evaluation and hash generation that runs on the server-side while Algorithm 3 provides an overview of overall client-side verification steps.

The repair mechanism of our proposed method requires a local re-computation of a single homomorphic gate. However, we only perform a re-computation when the output ciphertext fails to decrypt correctly. On the other hand, the client simply accepts the result if the decryption turns out to be correct. This leads to a possibility of a timing attack due to timing variation introduced by the gate computation. An adversary can potentially monitor this timing variation to learn whether the decryption was successful or not. However, the adversary needs to have access to the client machine to accurately measure this timing variation, which violates our threat model where we assume that having access to the client machine is difficult in practice and the adversary needs to be stronger in this model. On the other hand, our threat model considers the server to be malicious and it does not have direct access to the client machine, otherwise, the server could have decrypted the perturbed ciphertext itself and would not have to depend on the reaction of the client. A potential countermeasure to fix this issue might be to re-compute all the gates at the last level, irrespective of whether the decryption is correct or not.

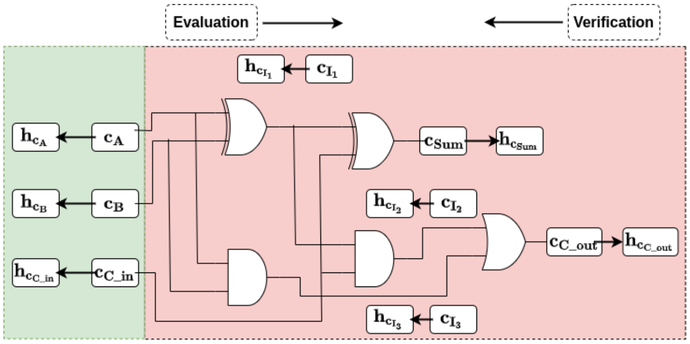


Fig. 13: Boolean circuit of a Full Adder. The client (green) generates the ciphertexts corresponding to plaintext inputs, along with its hashes. The server (red) evaluates the circuit homomorphically on the client inputs to generate outputs. While evaluation starts from inputs and moves toward output(s), verification starts from the output(s) and moves toward inputs.

8.1 Working example using a Full-Adder Circuit

We show our method by taking an example of a 1-bit full adder, as shown in Fig. 13. We have specifically chosen this circuit as it forms a part of the application-level constraint of our PIR database example provided in section 6.2. The green shaded region shows the client operations, which start with encrypting the inputs and generating the corresponding ciphertexts c_A, c_B, and $c_{C_{in}}$ and their corresponding hashes h_{c_A}, h_{c_B} and $h_{c_{C_{in}}}$. The client then sends ciphertext-hash pairs (c_A, h_{c_A}), (c_B, h_{c_B}) and $(c_{C_{in}}, h_{c_{C_{in}}})$ to the server for the computation of a full adder circuit.

Upon receiving these pairs, the server (shown in red) starts the evaluation of the full adder circuit on a level-by-level basis. At the first level, it computes $c_{I_1} = c_A \oplus c_B$ and $c_{I_3} = c_A \wedge c_B$, where $\oplus$ and $\wedge$ represents homomorphic XOR and AND gates, respectively. It also generates their corresponding hashes using an agreed-upon hash function as $h_{c_{I_1}} = Hash\{c_{I_1}, h_{c_A}, h_{c_B}\}$ and $h_{c_{I_3}} = Hash\{c_{I_3}, h_{c_A}, h_{c_B}\}$. Once completed, it proceeds to the second level where it computes $c_{Sum} = c_{I_1} \oplus c_{C_{in}}$ and $c_{I_2} = c_{I_1} \wedge c_{C_{in}}$. It also generates their corresponding hashes as $h_{c_{Sum}} = Hash\{c_{Sum}, h_{c_{I_1}}, h_{c_{C_{in}}}\}$ and $h_{c_{I_2}} = Hash\{c_{I_2}, h_{c_{I_1}}, h_{c_{C_{in}}}\}$. Finally, at the third and last level, it computes the remaining output as $c_{C_{out}} = c_{I_2} \vee c_{I_3}$, where $\vee$ represents homomorphic OR gate. It also generates its corresponding hash as $h_{c_{C_{out}}} = Hash\{c_{C_{out}}, h_{c_{I_2}}, h_{c_{I_3}}\}$. It sends back to the client the output ciphertexts (c_{Sum} and $c_{C_{out}}$) along with the auxiliary information (c_{I_1}, c_{I_2}, c_{I_3}, c_A, c_B, $c_{C_{in}}$, $h_{c_{Sum}}$, $h_{c_{C_{out}}}$, h_{c_A}, h_{c_B}, $h_{c_{C_{in}}}$, $h_{c_{I_1}}$, $h_{c_{I_2}}$, and $h_{c_{I_3}}$). One may argue that the server need not send the ciphertexts and their hashes that corresponds to the inputs back to the client since they were generated by the client itself. We would like to clarify that this situation may not arrive for general circuits with depth more than 3. Since we are trying to show our method on a small circuit which can then be extended to other circuits of higher depth, we explain the mechanism from the point of a general circuit.

Coming back to our example, upon receiving the above information, the client first generates the hashes corresponding to the output ciphertexts c_{Sum} and $c_{C_{out}}$, along with the hashes of the intermediate ciphertexts c_{I_1}, c_{I_2}, c_{I_3} and $c_{C_{in}}$ as follows:

$$\begin{aligned} h'_{c_{Sum}} &= Hash\{c_{Sum}, h_{c_{I_1}}, h_{c_{C_{in}}}\} \\ h'_{c_{C_{out}}} &= Hash\{c_{C_{out}}, h_{c_{I_2}}, h_{c_{I_3}}\} \\ h'_{c_{I_1}} &= Hash\{c_{I_1}, h_{c_A}, h_{c_B}\} \\ h'_{c_{C_{in}}} &= Hash\{c_{C_{in}}\} \\ h'_{c_{I_2}} &= Hash\{c_{I_2}, h_{c_{I_1}}, h_{c_{C_{in}}}\} \\ h'_{c_{I_3}} &= Hash\{c_{I_3}, h_{c_A}, h_{c_B}\} \end{aligned}$$

Once generated, it compares these values with the received hashes to check whether $h'_{c_{Sum}} = h_{c_{Sum}}$, $h'_{c_{C_{out}}} = h_{c_{C_{out}}}$, $h'_{c_{I_1}} = h_{c_{I_1}}$, $h'_{c_{C_{in}}} = h_{c_{C_{in}}}$, $h'_{c_{I_2}} = h_{c_{I_2}}$ and $h'_{c_{I_3}} = h_{c_{I_3}}$. If anyone of these checks fails, the client assumes that the received data was modified due to some transmission error, and it asks the server to re-transmit the values. At this point, the server has no incentive of changing anything in the information to be resent since the client has not yet decrypted the underlying ciphertext. On the other hand, if the hashes match then the client proceeds with decrypting the ciphertexts. Now, if both the ciphertexts decrypt correctly then the client simply accepts the results and does not provide any feedback to the server. At this point, the server is sure that the hashes have matched and the ciphertexts have decrypted correctly. Conversely, if any one of the ciphertexts fails to decrypt correctly, the client only proceeds with verification for that particular ciphertext. We would like to highlight that in our attack the server is forced to perturb only one of the output ciphertexts. The reason is that if it perturbs more than one ciphertext then, upon receiving feedback, the server will not be sure which perturbations lead to decryption failure. However, a lack of feedback will ensure that the final result has been decrypted correctly at the client's end. Thus, the server only perturbs one ciphertext at a time so that at most one output ciphertext fails to decrypt correctly. The server will be sure which ciphertext decrypts incorrectly when it receives feedback.

We now show how the client proceeds with the verification process for the decryption of each of the two outputs. The given circuit can be divided into two sub-circuits, with the *Sum* circuit of depth 2 being our base case and the C_{out} circuit of depth 3 which can be extended to any circuit of arbitrary depth. A circuit of depth 1 is basically a single gate computation that the client itself can do. This is required for our method to work since the client will be evaluating at least one gate during the verification stage.

Sum : The client proceeds to recompute the value of c_{Sum} at its end by evaluating $c'_{Sum} = c_{I_1} \oplus c_{C_{in}}$, where c_{I_1} and $c_{C_{in}}$ were both sent by the server. Once generated, it checks whether $c'_{Sum} = c_{Sum}$ or not. If they do not match, the client will know that the server has maliciously tampered with the value of c_{Sum}.

$\underline{\mathbf{C_{out}}}$: The client proceeds to recompute the value of $c_{C_{out}}$ at its end by evaluating $c'_{C_{out}} = c_{I_2} \vee c_{I_3}$, where c_{I_2} and c_{I_3} are sent by the server. Once generated, it checks whether $c'_{C_{out}} = c_{C_{out}}$ or not. If they do not match, the client will know that the server has maliciously tampered with the value of $c_{C_{out}}$.

For only repair, client can compute c'_{Sum} and $c'_{C_{out}}$ values after decrypting c_{I_1}, c_{I_2} and c_{I_3} respectively. However, to get a clear understanding of the server's intention (whether the server itself is malicious or not) client-side homomorphic computations ($c'_{Sum} = c_{I_1} \oplus c_{C_{in}}$ and $c'_{C_{out}} = c_{I_2} \vee c_{I_3}$) followed by equality checks ($c'_{Sum} = c_{Sum}$ and $c'_{C_{out}} = c_{C_{out}}$) are necessary. That decides the overall overhead of the scheme. Also in both cases, the client does not ask for a re-computation since it already has computed the correct value locally. On the other hand, if the evaluated ciphertext matches the one sent by the server then the client assumes that the decryption failure was accidental. However, since it obtained the same ciphertext as the one sent by the server even after a local computation, the client is now forced to ask for a re-computation. This request does not leak any information to the client as this is a result of an accidental failure.

The above two cases can now be extended to any circuits of any depth. The client will only need to recompute one gate on its end while it needs to verify hashes for three ciphertexts.

9 Defense against Our Proposed Attacks

While the existing schemes offering verifiable FHE do not protect against the two attacks (refer Sec. 5.2) proposed earlier in this paper, we show that our countermeasure `vr`^{2}`FHE` reliably thwarts such attacks. Our proposed scheme first performs verification and then decides whether to repair or request for a re-computation. In this section, we establish that any re-computation request does not leak any information regarding the error or the secret key to the server.

Defense against message recovery in first attack In our first attack, we showed that the server could force the final output to be an encryption of some known bit m by adding a gate on top of the output gate and sending it to the client for decryption. Upon decryption, if the client does not react then the server will know that the underlying message is m, otherwise, it will know that the message is $\overline{m}$. We now show how our proposed countermeasure can help the client in detecting such modifications in the circuit. Fig. 14(A) shows the final output and its hash when such a gate, say XOR, is added while Fig. 14(B) shows the final output and its hash when no such gate is added and the final output is the result of some known gate, say NAND. Assuming that the gate is added, the client will get the final output as c'_{out} along with the auxiliary information $\{h'_{c'_{out}}, c_{out}, h_{c_{out}}, h_{c_1}, h_{c_2}, h_{c_1}, h_{c_2}\}$. Upon receiving this information, the client first proceeds with the hash verification and, assuming that they match, performs

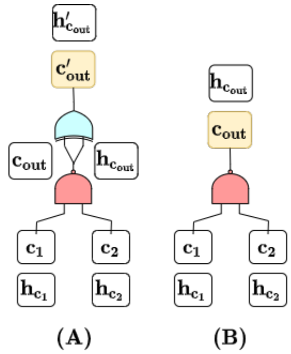

$\mathbf{h}'_{\mathbf{c}_{\text{out}}}$
$\mathbf{c}'_{\text{out}}$
$\mathbf{c}_{\text{out}}$
$\mathbf{h}_{\mathbf{c}_{\text{out}}}$
$\mathbf{c}_1$
$\mathbf{c}_2$
$\mathbf{h}_{\mathbf{c}_1}$
$\mathbf{h}_{\mathbf{c}_2}$
(A)
$\mathbf{h}_{\mathbf{c}_{\text{out}}}$
$\mathbf{c}_{\text{out}}$
$\mathbf{c}_1$
$\mathbf{c}_2$
$\mathbf{h}_{\mathbf{c}_1}$
$\mathbf{h}_{\mathbf{c}_2}$
(B)

Fig. 14: Shows the final output of the circuit (A) when a gate is added at the last level to force the output to be encryption of a particular bit, and (B) when no gate is added. The original output gate is shown in red, while the added gate is shown in blue. The ciphertexts that mismatch are highlighted in yellow.

the decryption of the received ciphertext c'_{out}. If the result turns out to be wrong, it proceeds with the re-computation of the final gate to verify whether the decryption error was forced intentionally or not. However, it evaluates the NAND gate instead of XOR, since it is the actual gate that is supposed to generate the final output. Since the added gate is different from the original gate, the computed ciphertext c''_{out} will obviously differ from the one received by the client which is c'_{out}. Moreover, even if the added gate is the same as the final gate, the generated output will still differ since the inputs to these gates will be different across the two gates due to them being on different levels. Hence, the server cannot use such an additional gate to force the final output to be an encryption of a known message without getting detected.

Defense against error recovery in first attack and key recovery in second attack In our first attack, we also showed that the server might replace an output ciphertext, which is the result of a genuine computation, with another ciphertext it computed during some previous execution thus performing a replay attack. Moreover, in both our attacks, we showed that the server might perturb the output ciphertext by adding an error value to either its scalar component b or its vector component $\mathbf{a}$. However, in case of decryption failures in both these attacks, the client computes the output ciphertexts on its end by using the auxiliary information it received from the server. In order to ensure that even re-computation on the client's end generates the same output ciphertext that was sent in the first place, the server will have to also carefully perturb the intermediate ciphertexts so that their error values add up to the one present in the final ciphertext. For example, the server adds an error e to the output ciphertext c_{out} and sends the modified output c'_{out}. To avoid detection, it has to add errors e_1 and e_2 to the intermediate ciphertexts c_1 and c_2 and send the modified ciphertexts c'_1 and c'_2 in such a way that when the client evaluates $c'_1 \oplus c'_2$ on its end, it gets c'_{out} containing error e. However, it needs to be kept in mind that this homomorphic operation $\oplus$ involves bootstrapping which will reduce the error $e_1 + e_2$ in the computed ciphertext if it is below the threshold, while it will flip the underlying message and still reduce the error if the total error is greater then the threshold. Thus in both cases, the final error will not be equal to e which will also ensure that the output of this local gate computation will not match the received ciphertext c'_{out}. Hence our proposed technique helps the client to detect whether the final ciphertext was perturbed or not. Moreover, it also helps the client to detect replay attacks, which were not possible previously.

Attacking intermediate gate(s) or result(s) and its limitations: Since our protocol only verifies the result of the last level and the hashes of the last two levels of the computation, the server might try to tamper with the circuit or the results at a lower level. For example, it can replace a gate or a result at a lower level to check whether it still has any effect on the final ciphertext or not. However, this attack may not be effective since the gates at the higher level of the circuit can mask the effect of these perturbations. For example, suppose the server replaces a NAND gate with an AND gate, which causes the resulting message bit to change from 1 to 0. It might happen that this ciphertext goes into an OR gate as one of the inputs while the other input is an encryption of 1. Thus the output of this OR gate remains 1, irrespective of whether the NAND gate was replaced with an AND gate (since 0 OR 1 = 1) or not (since 1 OR 1 = 1). It needs to be highlighted that the server does not know any of these plaintext values since all these computations are being evaluated homomorphically. Also replacing the input(s) and/or intermediate result(s) might not work, since even its effect may be masked by the gate being computed on the replaced ciphertexts. For example, if an encryption of 0 is replaced with an encryption of 1, then an OR gate computed on this modified ciphertext will introduce no further errors if its other input is an encryption of 1, since $0 \ OR \ 1 = 1 \ OR \ 1 = 1$. Hence, in order to maximize the chances of observing the effect of the perturbations, the server will have to introduce them in the output ciphertexts which will be easily observable through our proposed countermeasure.

10 Performance Overhead

In this section, we evaluate the overhead of our proposed method in terms of running time and data transmission. To generate the hashes, we used a C-implementation of SHA-256 hashing algorithm taken from [1]. Moreover, since FHEW and TFHE ciphertexts are of the form $(\mathbf{a}, b)$ where $\mathbf{a}$ is a vector of dimension k and each of the entries of the ciphertext is an integer, we summed up all these entries to generate a short version of the ciphertext. Mathematically speaking, the ciphertext $c = (\mathbf{a}, b)$ was shortened as $c_s = \texttt{shorten}(c) = \sum_{i=1}^{k}(a_i) + b$,

Table 4: Shows the time required to run the homomorphic gates in the original library and to run the same gates along with generating the hashes, for FHEW and TFHE, respectively.

		NAND	AND	NOR	OR	XNOR	XOR
FHEW	Original Gate Computation	0.0620s	0.0619s	0.0618s	0.0621s	-	-
	Gate Computation with Hashing	0.0625s	0.0623s	0.0625s	0.0624s	-	-
TFHE	Original Gate Computation	0.0219s	0.0218s	0.0215s	0.0215s	0.0218s	0.0216s
	Gate Computation with Hashing	0.0221s	0.0220s	0.0217s	0.0217s	0.0219s	0.0216s

Table 5: Shows the time required to run a full adder circuit without and with generating the hashes, along with the size of computation result transmitted by the server to the client, for FHEW and TFHE, respectively.

		Server-side Computation	Client-side Verification	Output File Size
FHEW	Full Adder without Hashing	0.6853s	-	7.2KB
	Full Adder with Hashing	0.6877s	0.0616s	31.7KB
TFHE	Full Adder without Hashing	0.1078s	-	13.5KB
	Full Adder with Hashing	0.1080s	0.0215s	54.9KB

where $\texttt{shorten}()$ is a function that accepts a FHEW or TFHE ciphertext and generates its shortened version by adding up all the individual entries of the ciphertext. Its hash h_c was then generated as $h_c = \texttt{Hash}\{\texttt{toString}(c_s)\}$, where $\texttt{toString}()$ is a function that takes an integer as input and converts it into a string. The intuition behind this approach of shortening the ciphertext by summing up all its entries is that this sum will change if any of the entries of the ciphertext is changed. Mathematically speaking, if a ciphertext $c = (\mathbf{a}, b)$ is modified to $c' = (\mathbf{a}, b')$ where $b' = b+1$, then the hash of c and c' will differ significantly due to the property of the hash function. The same effect will be observed if an element of $\mathbf{a}$ is changed. This ensures that any changes made by the server in either b or any element of $\mathbf{a}$ causes the hash of the corresponding ciphertext to change significantly. On the other hand, to generate the hash h_{c_I} of some intermediate ciphertext $c_I = (\mathbf{a}, b)$ using the hashes $h_{c_{I1}}$ and $h_{c_{I2}}$, we first shortened the ciphertext to c_{I_s} and then converted it to string to generate c'_{I_s}. The hash of this ciphertext was then generated as $h_{c_I} = \texttt{Hash}\{\texttt{concat}(c'_{I_s}, h_{c_{I1}}, h_{c_{I2}})\}$, where $\texttt{concat}()$ is a function that takes multiple strings as input and concatenates them to produce a single string. Finally, the hash comparison was done in the form of a string comparison as the hashes obtained are in the form of strings.

Coming to the empirical values, table 4 provides a comparison of the total running times of the original gates as defined in the FHEW and TFHE, and the total running times when the gate computation is followed by the hash generation step. From the table, we can observe that for FHEW and TFHE, original gate computation takes an average of 0.0619 and 0.0216 seconds, respectively, while the average time increases to 0.0624 and 0.0218 seconds, respectively, when the hash generation step is included. Thus the hash generation step increases the per-gate computation time by an average of 0.0005 and 0.0002 seconds, or by 0.8% and 0.9%, for FHEW and TFHE, respectively.

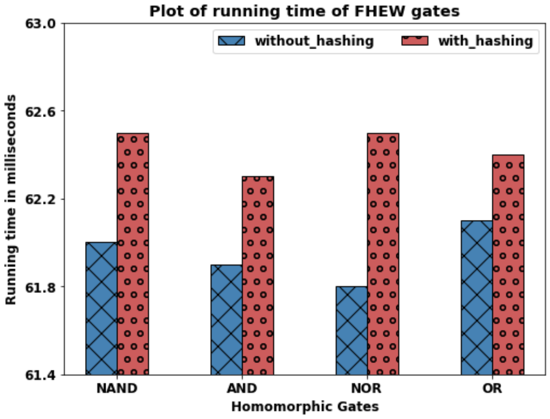


Fig. 15: Running time of all 4 gates of FHEW. The blue bar shows the running time when hashes of the gate outputs are not generated, while the red bar shows the running time when hashes of the gate outputs are generated.

Table 5 provides the running time of a full adder circuit implemented using the original gates and when these gates are followed by hash generation. The complete circuit takes around 0.6853 and 0.1078 seconds for FHEW and TFHE, respectively, when running without hash generation while it increases to 0.6877 and 0.1080 seconds when hash generation is involved. These values are shown as server-side computations in the table since the circuit is being evaluated by the server. The client-side verification, which involves hash re-generation and verification along with gate re-computation if required, takes 0.0616 and 0.0215 seconds for FHEW and TFHE, respectively. Thus the verification only takes around $\frac{1}{10}$-th and $\frac{1}{5}$-th of the time it would have required the client to evaluate the circuit (without the hash generation) on its end. Finally the table provides the amount of information the server needs to transmit to the client in both the cases. For the original circuit evaluation, the server needs to send about 7.2KB and 13.5KB of information, which consists of only the ciphertexts corresponding to the sum and carry out, in case of FHEW and TFHE, respectively. In case of the circuit evaluation with hashes, the server needs to send about 31.7KB and 54.9KB of information, which consists of other auxiliary information, in case of FHEW and TFHE, respectively. Thus our technique increases the overhead by about 24.5KB and 41.3KB, or by about 3.5× in case of FHEW and TFHE, respectively.

Fig. 15 and 16 provides a plot of the running times of all gates without and with hashing for all 4 gates of FHEW and 6 gates of TFHE, respectively. From the figures, we can observe that in the case of FHEW, AND takes the least amount of time while NAND and NOR both take the most amount of time when running with hashing. In the case of TFHE, we can observe that NOR, OR and XOR takes the least amount of time while NAND takes the most

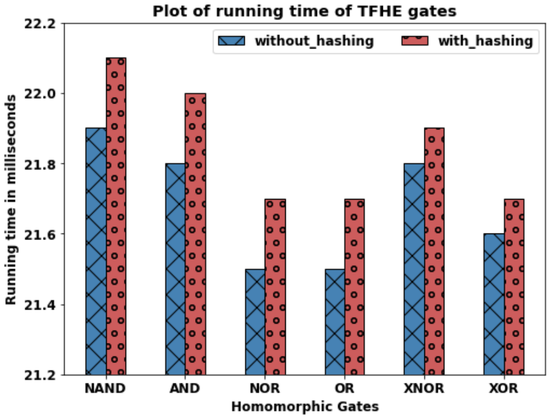


Fig. 16: Running time of all 6 gates of TFHE. The blue bar shows the running time when hashes of the gate outputs are not generated, while the red bar shows the running time when hashes of the gate outputs are generated.

amount of time when running with hashing. However, NOR and OR takes the least amount of time and NAND takes the most amount of time even when not running without hashing. Thus in the case of TFHE, our method does not add significant overhead in terms of per-gate computation.

11 Conclusion

In this paper, we have shown that access to a CVO can result in the leakage of the secret key to the malicious server. We have also shown that the error from a single ciphertext can be leaked with a constant number of queries to the CVO. In our experiment, we require 8 and 33 queries to extract errors from a single ciphertext for the libraries FHEW and TFHE, respectively. In our improved attack, we showed that we require at most 1 reaction from the client to recover a single bit of the secret key. While CVO-based attacks exist in the literature, in this paper we showed such an attack to recover the full secret key on practical schemes that are being used in the real-life construction of various applications. This attack highlights the fact that *additional protections need to be adopted at a system level to secure cloud applications* [27] built using such FHE schemes. This becomes all the more important since such schemes are gearing up for deployment in practice, and may handle sensitive information once they are deployed. Finally, we proposed a technique that aids the client in detecting such attacks and limits its feedback to the server. Our proposed technique does not incur significant overhead in terms of verification while incurring a slight overhead in terms of network overhead. Lastly, our technique also helps the client to repair the incorrect result while requiring minimal computation on its end.

References

1. Public domain c sha-256 implementation. https://github.com/983/SHA-256, accessed: 2023-04-02
2. Agrawal, S., Boneh, D., Boyen, X.: Efficient lattice (h) ibe in the standard model. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 553–572. Springer (2010)
3. Albrecht, M.R.: On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 103–129. Springer International Publishing, Cham (2017)
4. Albrecht, M.R., Cid, C., Faugère, J., Fitzpatrick, R., Perret, L.: Algebraic algorithms for LWE problems. ACM Commun. Comput. Algebra **49**(2), 62 (2015). https://doi.org/10.1145/2815111.2815158, https://doi.org/10.1145/2815111.2815158
5. Albrecht, M.R., Cid, C., Faugère, J., Fitzpatrick, R., Perret, L.: On the complexity of the BKW algorithm on LWE. Des. Codes Cryptogr. **74**(2), 325–354 (2015). https://doi.org/10.1007/s10623-013-9864-x, https://doi.org/10.1007/s10623-013-9864-x
6. Albrecht, M.R., Göpfert, F., Virdia, F., Wunderer, T.: Revisiting the expected cost of solving usvp and applications to LWE. IACR Cryptol. ePrint Arch. p. 815 (2017), http://eprint.iacr.org/2017/815
7. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015), http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml
8. Alkim, E., Ducas, L., Pöppelmann, T., Schwabe, P.: Post-quantum key {Exchange—A} new hope. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 327–343 (2016)
9. Arora, S., Ge, R.: New algorithms for learning in presence of errors. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I. Lecture Notes in Computer Science, vol. 6755, pp. 403–415. Springer (2011). https://doi.org/10.1007/978-3-642-22006-7_34, https://doi.org/10.1007/978-3-642-22006-7_34
10. Aydin, F., Aysu, A.: Exposing side-channel leakage of seal homomorphic encryption library. In: Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security. p. 95–100. ASHES'22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3560834.3563833, https://doi.org/10.1145/3560834.3563833
11. Aydin, F., Karabulut, E., Potluri, S., Alkim, E., Aysu, A.: Reveal: Single-trace side-channel leakage of the seal homomorphic encryption library. In: Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe. p. 1527–1532. DATE '22, European Design and Automation Association, Leuven, BEL (2022)
12. Bai, S., Galbraith, S.D.: Lattice decoding attacks on binary lwe. In: Susilo, W., Mu, Y. (eds.) Information Security and Privacy. pp. 322–337. Springer International Publishing, Cham (2014)
13. Bai, S., Miller, S., Wen, W.: A refined analysis of the cost for solving LWE via usvp. In: Buchmann, J., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa,

Rabat, Morocco, July 9-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11627, pp. 181–205. Springer (2019). https://doi.org/10.1007/978-3-030-23696-0_10, https://doi.org/10.1007/978-3-030-23696-0_10

14. Banerjee, A., Peikert, C., Rosen, A.: Pseudorandom functions and lattices. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 719–737. Springer (2012)
15. Bi, L., Lu, X., Luo, J., Wang, K., Zhang, Z.: Hybrid dual attack on LWE with arbitrary secrets. Cybersecur. **5**(1), 15 (2022)
16. Bois, A., Cascudo, I., Fiore, D., Kim, D.: Flexible and efficient verifiable computation on encrypted data. In: Garay, J.A. (ed.) Public-Key Cryptography - PKC 2021 - 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10-13, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12711, pp. 528–558. Springer (2021). https://doi.org/10.1007/978-3-030-75248-4_19, https://doi.org/10.1007/978-3-030-75248-4_19
17. Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M.: The zettabyte file system (03 2023)
18. Bos, J., Costello, C., Ducas, L., Mironov, I., Naehrig, M., Nikolaenko, V., Raghunathan, A., Stebila, D.: Frodo: Take off the ring! practical, quantum-secure key exchange from lwe. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 1006–1018 (2016)
19. Bos, J.W., Costello, C., Naehrig, M., Stebila, D.: Post-quantum key exchange for the tls protocol from the ring learning with errors problem. In: 2015 IEEE Symposium on Security and Privacy. pp. 553–570. IEEE (2015)
20. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical gapsvp. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 868–886. Springer (2012). https://doi.org/10.1007/978-3-642-32009-5_50, https://doi.org/10.1007/978-3-642-32009-5_50
21. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012. pp. 309–325. ACM (2012). https://doi.org/10.1145/2090236.2090262, https://doi.org/10.1145/2090236.2090262
22. Catalano, D., Fiore, D.: Practical homomorphic macs for arithmetic circuits. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. pp. 336–352. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
23. Chatel, S., Knabenhans, C., Pyrgelis, A., Hubaux, J.P.: Verifiable encodings for secure homomorphic analytics (2022)
24. Chenal, M., Tang, Q.: On key recovery attacks against existing somewhat homomorphic encryption schemes. In: Aranha, D.F., Menezes, A. (eds.) Progress in Cryptology - LATINCRYPT 2014. pp. 239–258. Springer International Publishing, Cham (2015)
25. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast fully homomorphic encryption library (August 2016), https://tfhe.github.io/tfhe/
26. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: ASIACRYPT (1). pp. 3–33. Springer (2016)

27. Chillotti, I., Gama, N., Goubin, L.: Attacking fhe-based applications by software fault injections. Cryptology ePrint Archive, Paper 2016/1164 (2016), https://eprint.iacr.org/2016/1164, https://eprint.iacr.org/2016/1164
28. Das, A., Dutta, S., Adhikari, A.: Indistinguishability against chosen ciphertext verification attack revisited: The complete picture. In: Susilo, W., Reyhanitabar, R. (eds.) Provable Security. pp. 104–120. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
29. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles. p. 205–220. SOSP '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1294261.1294281, https://doi.org/10.1145/1294261.1294281
30. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 24–43. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
31. Ducas, L., Micciancio, D.: Fhew: Bootstrapping homomorphic encryption in less than a second. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015. pp. 617–640. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
32. Ducas, L., Micciancio, D.: FHEW: A fully homomorphic encryption library (May 2017), https://github.com/lducas/FHEW
33. Espitau, T., Joux, A., Kharchenko, N.: On a dual/hybrid approach to small secret lwe: A dual/enumeration technique for learning with errors and application to security estimates of fhe schemes. In: Progress in Cryptology – INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings. p. 440–462. Springer-Verlag, Berlin, Heidelberg (2020)
34. Fauzi, P., Hovd, M.N., Raddum, H.: On the ind-cca1 security of fhe schemes. Cryptology ePrint Archive, Paper 2021/1624 (2021), https://eprint.iacr.org/2021/1624, https://eprint.iacr.org/2021/1624
35. Fiore, D., Gennaro, R., Pastro, V.: Efficiently verifiable computation on encrypted data. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. p. 844–855. CCS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2660267.2660366, https://doi.org/10.1145/2660267.2660366
36. Fiore, D., Nitulescu, A., Pointcheval, D.: Boosting verifiable computation on encrypted data. In: Kiayias, A., Kohlweiss, M., Wallden, P., Zikas, V. (eds.) Public-Key Cryptography – PKC 2020. pp. 124–154. Springer International Publishing, Cham (2020)
37. Ganesh, C., Nitulescu, A., Soria-Vazquez, E.: Rinocchio: Snarks for ring arithmetic. IACR Cryptol. ePrint Arch. p. 322 (2021), https://eprint.iacr.org/2021/322
38. Gennaro, R., Wichs, D.: Fully homomorphic message authenticators. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology - ASIACRYPT 2013. pp. 301–320. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
39. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), crypto.stanford.edu/craig
40. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing. p. 169–178. STOC '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1536414.1536440, https://doi.org/10.1145/1536414.1536440

41. Gentry, C., Halevi, S.: Implementing gentry's fully-homomorphic encryption scheme. In: Paterson, K.G. (ed.) Advances in Cryptology – EUROCRYPT 2011. pp. 129–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
42. Gentry, C., Peikert, C., Vaikuntanathan, V.: Trapdoors for hard lattices and new cryptographic constructions. In: Proceedings of the fortieth annual ACM symposium on Theory of computing. pp. 197–206 (2008)
43. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Annual Cryptology Conference. pp. 75–92. Springer (2013)
44. Guo, Q., Johansson, T.: Faster dual lattice attacks for solving LWE with applications to CRYSTALS. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 13093, pp. 33–62. Springer (2021). https://doi.org/10.1007/978-3-030-92068-5_2, https://doi.org/10.1007/978-3-030-92068-5_2
45. Hall, C., Goldberg, I., Schneier, B.: Reaction attacks against several public-key cryptosystem. In: Varadharajan, V., Mu, Y. (eds.) Information and Communication Security. pp. 2–12. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
46. Herrn, H., Gauss, R.: Anwendung der wahrscheinlichkeitsrechnung auf eine aufgabe der practischen geometrie. von herrn hofrath und ritter gauss. Astronomische Nachrichten **1**(6), 81–86 (1823). https://doi.org/https://doi.org/10.1002/asna.18230010602, https://onlinelibrary.wiley.com/doi/abs/10.1002/asna.18230010602
47. Hu, Z., Sun, F., Jiang, J.: Ciphertext verification security of symmetric encryption schemes. Sci. China Ser. F Inf. Sci. **52**(9), 1617–1631 (2009)
48. Kara, M., Laouid, A., dabbas, O.A., Hammoudeh, M., Bounceur, A.: One digit checksum for data integrity verification of cloud-executed homomorphic encryption operations. Cryptology ePrint Archive, Paper 2023/231 (2023), https://eprint.iacr.org/2023/231, https://eprint.iacr.org/2023/231
49. Laine, K., Lauter, K.E.: Key recovery for LWE in polynomial time. IACR Cryptol. ePrint Arch. p. 176 (2015), http://eprint.iacr.org/2015/176
50. Li, Z., Galbraith, S.D., Ma, C.: Preventing adaptive key recovery attacks on the gentry-sahai-waters leveled homomorphic encryption scheme. Cryptology ePrint Archive, Paper 2016/1146 (2016), https://eprint.iacr.org/2016/1146, https://eprint.iacr.org/2016/1146
51. Lindner, R., Peikert, C.: Better key sizes (and attacks) for lwe-based encryption. In: Kiayias, A. (ed.) Topics in Cryptology – CT-RSA 2011. pp. 319–339. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
52. Loeliger, J.: Version Control with Git - Powerful techniques for centralized and distributed project management. O'Reilly (2009), http://www.oreilly.de/catalog/9780596520120/index.html
53. Loftus, J., May, A., Smart, N.P., Vercauteren, F.: On cca-secure somewhat homomorphic encryption. In: In Selected Areas in Cryptography. pp. 55–72 (2011)
54. Lou, Q., Santriaji, M., Yudha, A.W.B., Xue, J., Solihin, Y.: vfhe: Verifiable fully homomorphic encryption with blind hash (2023)
55. Lyubashevsky, V.: Lattice signatures without trapdoors. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 738–755. Springer (2012)

56. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. pp. 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
57. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology — CRYPTO '87. pp. 369–378. Springer Berlin Heidelberg, Berlin, Heidelberg (1988)
58. Micciancio, D., Peikert, C.: Trapdoors for lattices: Simpler, tighter, faster, smaller. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 700–718. Springer (2012)
59. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Cryptography Mailing list at https://metzdowd.com (03 2009)
60. Natarajan, D., Loveless, A., Dai, W., Dreslinski, R.: Chex-mix: Combining homomorphic encryption with trusted execution environments for two-party oblivious inference in the cloud. Cryptology ePrint Archive, Paper 2021/1603 (2021), https://eprint.iacr.org/2021/1603, https://eprint.iacr.org/2021/1603
61. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing. p. 84–93. STOC '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1060590.1060603, https://doi.org/10.1145/1060590.1060603
62. Shimin Li, Xin Wang, Rui Zhang 016: Privacy-preserving homomorphic macs with efficient verification. In: Web Services - ICWS 2018 - 25th International Conference, Held as Part of the Services Conference Federation, SCF 2018, Seattle, WA, USA, June 25-30, 2018, Proceedings. Springer (2018)
63. Smart, N., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography - PKC 2010. vol. 6056, pp. 420–443. Springer Berlin Heidelberg, Germany (2010), other page information: 420-443 Conference Proceedings/Title of Journal: Public Key Cryptography - PKC 2010 Other identifier: 2001176
64. Viand, A., Knabenhans, C., Hithnawi, A.: Verifiable fully homomorphic encryption (2023). https://doi.org/10.48550/ARXIV.2301.07041, https://arxiv.org/abs/2301.07041
65. Zhang, Z., Plantard, T., Susilo, W.: Reaction attack on outsourced computing with fully homomorphic encryption schemes. In: Kim, H. (ed.) Information Security and Cryptology - ICISC 2011. pp. 419–436. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)