A Practical Full Key Recovery Attack on TFHE and FHEW by Inducing Decryption Errors

Bhuvnesh Chaturvedi
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Kharagpur, India
bhuvneshchaturvedi2512@gmail.com

Anirban Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Kharagpur, India
ch.anirban00727@gmail.com

Ayantika Chatterjee
Advanced Technology Development Centre
Indian Institute of Technology, Kharagpur
Kharagpur, India
cayantika@gmail.com

Debdeep Mukhopadhyay
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Kharagpur, India
debdeep.mukhopadhyay@gmail.com

***Abstract*—Fully Homomorphic Encryption (FHE) promises to secure our data on the untrusted cloud, while allowing arbitrary computations. Recent research has shown two side channel attacks on the client side running a popular HE library. However, no side channel attacks have yet been reported on the server side in existing literature. The current paper shows that it is possible for adversaries to inject perturbations in the ciphertexts stored in the cloud to result in decryption errors. Most importantly, we highlight that when the client reports of such aberrations to the cloud service provider the complete secret key can be extracted in few attempts. Technically, this implies a break of the IND-CVA (Indistinguishability against Ciphertext Verification Attacks) security of the FHE schemes. The core idea of the attack is to extract the underlying error values for each homomorphically computed ciphertext and thereby construct an exact system of equations. As the security of the underlying Learning with Errors (LWE) collapse with the leakage of the errors, the adversary is capable of ascertaining the secret keys. We demonstrate this attack on two well-known FHE libraries, namely FHEW and TFHE. The objective of the server is to perform the attack in a stealthy manner, without raising any suspicion from the innocent client. Therefore in a practical scenario, the successful key retrieval from a client would require the server to perform the attack with as few queries as possible. Thus we craftily use timing information during homomorphic gate computations to optimise our attack and significantly reduce the required number of queries per ciphertext. More precisely, we need 8 and 23 queries to the client for each error recovery for FHEW and TFHE, respectively. We mount a full-key recovery attack [1] on TFHE (without and with bootstrapping) with key size of 630 bits and successfully faulted 739 and 930 ciphertexts to recover correct errors. This required a total of 19838 and 29200 client queries respectively. In case of FHEW with key size 500, we successfully faulted 766 ciphertexts to recover correct errors, which required 7565 client queries. 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.**

***Index Terms*—FHE, LWE, IND-CVA, ciphertext verification attack, key recovery**

[1]The demo code for our attack is available on https://github.com/SEAL-IIT-KGP/CVO-TFHE. Please feel free to check out this demo, and let us know if you have any questions on the same.

I. Introduction

Fully Homomorphic Encryption (FHE) schemes [1], [2], [3], [4], [5], [6], [7], [8] are a class of encryption schemes that allow arbitrary computations on encrypted data without the need to decrypt it first, while also ensuring that the output remains encrypted as well. In other words, such schemes accept a set of ciphertexts $\pi_1, \pi_2, \ldots, \pi_n$ and evaluates a known function f on them to obtain a set of resulting ciphertexts $c_1, c_2, \ldots, c_m$, without any knowledge of the secret key used to generate the input ciphertexts. Such schemes are helpful in construction of privacy-preserving protocols in cloud computation scenario that allows a user (client) to offload its confidential data onto a remote cloud. The cloud can also perform arbitrary computations on it on behalf of the user without revealing the original data. The client is anyone in possession of the secret key and uses it to encrypt its data. Once encrypted, it sends the data to a server to be processed. The server, in general, evaluates a known function on this encrypted data using a (publicly available) bootstrapping or evaluation key which is an encryption of the secret key itself. The server sends the result of the computation, which is still in encrypted form, back to the client. The client, upon receiving the encrypted result, decrypts it using the secret key it possesses. The owner of the secret key also may use it to generate a set of public keys for others to encrypt their data which can then be sent to the cloud for computation. However, in both scenarios, decryption can only be performed by the client who possesses the secret key. Moreover, in both scenarios, once the data leaves the client machine, it remains encrypted throughout the transmission and computation stages. The server remains oblivious to the inputs as well as the output(s); however, it does know the function that is being evaluated and the design of the circuit that implements it.

The basic assumption of FHE is that the server is untrusted, and thus it should not know any information about the client data. This stems from the fact that the data stored on the server is encrypted under the client's key which the server does not possess. On the other hand, the server is free to perform any computation on the encrypted data as it is not under the client's control. Therefore, considering the underlying crypto-primitive to be mathematically strong, the server may start undertaking spurious activities including, but not limited to, manipulating client's data in order to extract private information. Thus the server itself is considered as untrusted and malicious. It must also be mentioned that the aim of such a server is to retrieve private information about the client's data while also ensuring that the attack remains undetected, so as not to lose trust of the client. Thus it becomes necessary to evaluate the security of such FHE schemes from the practical aspect as well, apart from primitive and implementation levels. As observed by authors in [9], to make an informed choice on the security guarantees of homomorphic schemes, one needs to consider a broader view of the overall system, along with the primitive.

Interestingly, if the data gets corrupted during computation or transmission, it might result in incorrect decryption. In such a scenario, the client might inform the server of such erroneous results and ask for re-computations. As stated in [9], such a scenario exists in practice in pay-per-computation model, where the client pays for each correct computation. In case of a wrong result, it will certainly ask the server for a free re-computation. Moreover, in a practical setting, the client can send a sample set of ciphertext and verify the results at its end in order to determine the quality-of-service, before availing the actual computational service. Previous works [9], [10], [11] have already shown how this "reaction" from the client can be exploited by the server to leak secret information. In this paper, we show that this "reaction" attack [12], [13] can be used to mount a full key recovery attack on two well known and practically used FHE schemes, namely FHEW and TFHE.

The security of FHE schemes [14], [15], [16], [17], [18], [19], [20] relies on mathematically hard problems such as Learning With Errors (LWE) [21], or its ring variant Ring-LWE [22]. The intractability of these schemes depends upon the idea of *noise (or error)* [2], a small value that is added to ciphertexts during encryption operation which grows when homomorphic operations are performed on these ciphertexts. Once this noise grows beyond a certain threshold limit, decryption will no longer work correctly and will give wrong result. Thus a refreshing operation is required to bring back the noise to an acceptable level. Gentry in 2009 [23] introduced the idea of bootstrapping, which performs a decryption operation on the ciphertext using an encryption of the secret key. The problem with bootstrapping is that it is a very costly operation, and the efficiency of an FHE scheme depends on how fast the bootstrapping can be performed. FHEW [15] was the first scheme to implement a bootstrapped gate that works under 1 second, which was further brought down to under 0.1 seconds in TFHE [24].

[2]Moving forward, we use the term 'error' to denote the noise in LWE equations.

A. Motivation

The majority of the works that tried to break the security of FHE schemes either reported the asymptotic complexity of solving the mathematically hard problems [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], or reported the security level of the schemes built using these problems [38]. Since these mathematical problems are considered to be hard both in classical and quantum world, schemes built using these primitives are considered to be post quantum secure. However, as established through decades of research [39], [40], [41], [42], [43], a mathematically secure cryptographic scheme could be jeopardised in practice due to implementational hindsight. Such implementations may inadvertently leak secret information through passive, or side-channels. Recently, two side-channel attacks [44], [45] have been shown on two different versions of a popular HE library [46], [3], that extracts the plaintext message being encrypted or the secret key that is being generated using a single power trace. However, one must note that both these attacks target the operations running on the *client side* to observe side channel information.

Given the client-server settings for FHE applications, performing side channel attacks at the client side is a relatively stronger attack model as it requires access to the client device by the adversary, which is certainly not easy. Moreover, the essence of FHE lies in the fact that the cloud server is untrusted and thus, can indulge in malicious activities to extract sensitive information. It must be mentioned in this context that attacking the FHE applications at the server side is not trivial as neither the secret key gets involved in the computations nor error values (noise) are generated on the server. More specifically, while the secret key is directly involved in the computations at the client end, the ciphertext at the server side does not directly reveal the secret key. However, the error values incorporated in the ciphertext does grow in magnitude due to computations (homomorphic operations) at the server side, thereby necessitating the use of *bootstrapping mechanism* to contain the error. Moreover, since the plaintext bits are encrypted using a semantically secure encryption scheme, the ciphertext does not leak any information about the underlying plaintext message. This prevents an attacker to conduct any attack on its own to reveal sensitive data, without an external feedback [9]. This external feedback can come in the form of a reaction from the client when it decrypts a perturbed ciphertext, which a malicious server can leverage to leak sensitive information. However, such queries may attract attention from the client, and if detected, it will result in the client to loose trust on the server and revoke its service. Thus it is important for the server to keep the number and frequency of such queries to as low as possible. Thus the server may look for any additional information that may help it to reduce the number of queries.

B. Contribution

In this paper, we make the following contributions.

- We show that a malicious server can introduce intended and calculated perturbations in the homomorphically computed ciphertext and perform "reaction" attacks on the client by using the feedback from the client.
- We provide a novel attack scheme using the error threshold for decryption and a reduced error range to induce faults in the ciphertext.
- Unlike prior attacks, we target the server side where the secret key is not directly involved in any computation and show that *reactions* from client can leak potential information when considered in a cloud computing scenario.
- Finally, we recover exact error corresponding to each ciphertext by using a binary-search based approach to induce curated perturbations in the original ciphertext and utilizing feedback from the client. Once the errors are recovered, the secret key is extracted by forming a system of equations with the ciphertexts and solving them using Gaussian Elimination method.

C. Organization

The rest of the paper is organized as follows: Section II provides the background of LWE [21] problem along with a brief working of FHEW [15] and TFHE [24] libraries, while Section III provides a summary of existing attacks on FHE schemes. Section IV describes the threat model under which our attack works. Section V explains in details how the client works as a ciphertext verification oracle. Section VI explains how we leverage the CVO to mount a key recovery attack, while Section VII explains how we utilize timing information to reduce the number of queries. Section VIII provides our experimental results. Section IX provides potential countermeasures to our attack, and Section X concludes our paper.

II. Background

In this section, we provide a brief background on the Learning With Errors problem, which is the underlying mathematical foundation for the FHE schemes we discuss in this paper. We follow it up with working principles of two well-known FHE libraries that are based upon the LWE primitive.

A. Learning With Errors problem

The idea of Learning With Errors problem was introduced by Regev in 2005 [21]. Since its inception, LWE has been used as a foundation of multiple cryptographic constructions [47], [48], [49], [50], [51], [52], [53], [54] due to the assumption that it is as hard as worst case lattice problems. LWE is based on 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 $k \geq 1$ 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 $(\mathbf{a}, b) \in {\mathbb{Z}_{\shortparallel}}^k \times \mathbb{Z}_{\shortparallel}$, where $\mathbf{a} \in {\mathbb{Z}_{\shortparallel}}^k$ is chosen uniformly and $b = \mathbf{a} \cdot \mathbf{s} + e \in \mathbb{Z}_{\shortparallel}$. 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 ${\mathbb{Z}_{\shortparallel}}^k \times \mathbb{Z}_{\shortparallel}$.

Both the 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 problem or to estimate the security level of the schemes based on the parameter set used to implement them. However, once these error (noise) values are recovered, they can be removed from the corresponding ciphertext to obtain a system of exact equation which can then be trivially solved.

B. Torus Based Homomorphic Encryption

1) Torus Domain For TFHE: Torus [24] 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 results in a torus element. Product between two torus elements is not defined. In the CPU implementation of TFHE library [1], 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 μ and $-\mu$, and are represented as 32-bit unsigned integers.

2) Integer Domain for FHEW: The plaintexts and ciphertexts as well as the underlying operations in the FHEW library [2] 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 same, they are different in case of FHEW.

C. Fully Homomorphic Encryption Libraries

In this work, we focus on two well-known LWE based FHE libraries, namely FHEW [2] and TFHE [1]. The overall working principle of these two FHE schemes can be broadly broken into three stages. First is the *encryption stage* that runs on 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 server and does not directly involve the secret key. Once the computations is done at the server, the final encrypted result is sent to the client for decryption and involves the decryption key.

1) The Encryption Stage: The encryption process starts with sampling a noise value $e \in \mathbb{Z}_{\shortparallel}$ from a Gaussian distribution and adding it to the message m to obtain an intermediate value of $b = m + e$. 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 of b to obtain its final value as $b = \mathbf{a} \cdot \mathbf{s} + m + e$. The final ciphertext comes out to be $(\mathbf{a}, b)$. The above process is same in 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 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 $e \in {\mathbb{Z}_{\shortparallel}}^m$ consisting of noise values randomly sampled from a Gaussian distribution. It then computes a vector $b = \mathbf{A} \times \mathbf{s} + \mathbf{e} \in {\mathbb{Z}_{\shortparallel}}^m$, where "$\times$" represents the matrix-vector product. This matrix-vector pair $(\mathbf{A}, \mathbf{b})$ acts as its public key. To encrypt a 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.

2) 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 $(\mathbf{1024}, b_{gc}^F)$ and $(\mathbf{0}, b_{gc}^T)$ for FHEW and TFHE, respectively. The second part of these constants, i.e., b_{gc}^F and b_{gc}^T, 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 $b = b_{gc}^F - (b_1 + b_2)$ under modulo-512 in FHEW and by computing $\mathbf{a} = \mathbf{0} \pm (\mathbf{a_1} \pm \mathbf{a_2})$ and $b = b_{gc}^T \pm (b_1 \pm b_2)$ under modulo-2^{32} in TFHE, 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 refreshing operation changes the underlying secret key. In case of FHEW, an additional modulus switching operation is required to switch modulus from the ciphertext to the plaintext domain, while in TFHE it is carried out during noise reduction phase itself.

3) 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 $\langle \mathbf{a} \cdot \mathbf{s} \rangle$, which denotes the dot product between the vectors $\mathbf{a}$ and $\mathbf{s}$, 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 plaintext message x. In case of TFHE, the sign of this phase is checked to output 1 if it is positive and 0 if it is negative. In case of FHEW, a constant value 64 is added to this phase such that it becomes $x + e'$, where $e' = e + 64$. 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 is below a pre-determined threshold, otherwise it decrypts incorrectly.

III. Existing Attacks on FHE Schemes

The security guarantee of the FHE schemes, discussed in [24], [15], is based on the underlying hardness of LWE problem. Unsurprisingly, the earlier attempts to break the semantic security of these schemes were majorly focused on attacking the underlying LWE problem. For example the authors in [25] showed an attack on LWE problem when the coefficients of secret key $\mathbf{s} \in \mathbb{Z}_q^k$ (a vector of dimension k) are taken from integers modulo some pre-defined q. However, it does not take into consideration the case when $\mathbf{s} \in \{0, 1\}^k$ or $\mathbf{s} \in \{0, \pm 1\}^k$, i.e., the secret key is a Binary or Ternary vector of dimension k, which is the case for TFHE and FHEW. Interestingly, authors in [26] showed an attack on Binary LWE problem where the secret key is a Binary vector. This attack belongs to the class of Primal Attacks, which directly tries to solve the search version of the LWE problem. Similarly, [27] shows a Dual Attack on small-secret LWE, i.e., LWE problem where secret key is Binary or Ternary vector, to solve the decision version of the LWE problem and then use the *Search-to-Decision* reduction to recover the secret key. Recently, authors in [38] proposed a Dual attack on TFHE but the attack does not practically break the security of the scheme. Rather, the authors reported a drop in the security level of the scheme from the one reported in the original TFHE scheme [14]. It must be noted that all the above mentioned attacks are generic and are not practically feasible for the lattice dimensions used in the recent FHE schemes.

Apart from theoretical attacks on LWE primitives, side channel attacks have also been proposed in recent times that target the implementational aspects of FHE schemes. The first side-channel attack on HE has been demonstrated recently in [44], targeting the client side that is running the encryption operation of SEAL library [46]. The attack targets the conditional statements executed in the Gaussian Sampler routine to obtain the coefficients for the error polynomial. Another side-channel attack on HE has been demonstrated in [45], targeting the client side running the key generation step of latest version of SEAL library [3]. The attack targets the stage where the generated secret key is being converted to its Number Theoretic Transform (NTT) domain.

In FHE setting, attacking a client system poses realistic challenges, and one must assume a stronger attack model where the attacker has access to the client's machine. Whereas the server, itself being untrusted and malicious, provides a more realistic scenario where it can perform arbitrary compu-

tations on the ciphertext, make calculated perturbations, and even manipulate the FHE libraries to decipher the data stored and computed at the server. However, while the server is free to manipulate the ciphertexts that it possesses, it cannot decrypt them directly as it does not possess the secret key. In addition, the operations at the server do not directly involve the secret key [3]. In the context of HE, prior works have tried using *reaction-based attacks* to extract any meaningful information [9]. However, they have limited scope and do not directly apply to contemporary FHE schemes like FHEW and TFHE. For example, the attacks [9], [11] both target the encryption of secret key, i.e the bootstrapping key. Moreover, [11] assumes sparse key and a decryption oracle. In contrast, our attack works irrespective of bootstrapping stage, making it more generic and directed towards any LWE-based scheme. The attack in [10] shows a message recovery attack. On the other hand, ours is a full key recovery attack that targets the errors present in the output ciphertexts.

IV. Attacker Assumptions and Threat Model

In this section, we present the basic security assumptions which are valid for FHE. We further discuss on the attacker threat model which is relevant to the cloud computing scenario, under which FHE based applications are meant to operate in reality.

To begin with, the security of FHE schemes is evaluated under two security models, namely IND-CPA and IND-CCA. FHE schemes are expected to be IND-CPA secure, ensuring that an adversary can gain no information about the underlying plaintext from the ciphertext [9]. On the other hand, it has also been established that no FHE schemes can be either IND-CCA [10], [11], [55] or IND-CCA2 secure, which implies that an adversary can break such schemes if it has access to a decryption oracle [9]. Our attack operates under the notion of IND-CVA (Indistinguishability against Ciphertext Verification Attack) security [13], which is based on the idea of "reaction" attack from [12]. Under this premise, it is assumed that an adversary has access to an oracle that accepts a ciphertext as input and returns as output whether the decryption was successful or not. This oracle, which we refer to as *Ciphertext Verification Oracle* or CVO, is essentially the client itself in a "pay per running times model", where client pays for each correct computation [10], [11]. In such a model, the client could ask for a free re-computation in case the result returned by the cloud is incorrect. The client before using the FHE cloud services would typically have a verification phase, wherein it will check the correctness of the homomorphic ciphertexts. In case of decryption failures the client would need to report the same to the cloud, to avoid payments for erroneous service of the cloud. The client may be paying for a service on encrypted information on the cloud, which could be pertaining to data analytics, predictive analytics, etc. In case of inferior performance, the client company can analyze the exchanges and report on the possible erroneous instances to the service-providing entity, which is the cloud. In such cases, the existence of the decryption verification oracle is not necessarily restricted to the beginning period of the availed service but for the entire duration of the usage. Consider the setting where the server or cloud [4] offers homomorphic computations as a service through well-known FHE libraries like FHEW and TFHE. It must be noted that these FHE schemes operate on binary message space, i.e, the plaintext is either 0 or 1. Therefore, in order to obtain homomorphic computations on encrypted data, the actual data stream is represented in binary form and each bit is encrypted at the client end and sent to the server for homomorphic operations on individual ciphertext. It must be mentioned here that FHE allows homomorphic encryption in both public and private key settings. In public key setting, the messages are encrypted using public key while the encrypted computational result from the server is decrypted back using the secret key. Whereas in private key setting, both encryption and decryption is performed using the secret key. Irrespective of the key variant chosen by the client, *we target the decryption key, which is secret in both the cases.*

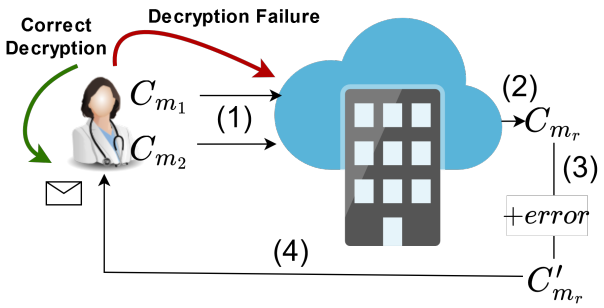

Fig. 1: Receiving client reaction works by (1) receiving two ciphertexts from client, (2) computing a homomorphic gate, (3) introducing carefully crafted perturbation, and (4) sending this modified ciphertext to the client. Client sends a feedback if decryption fails, otherwise keeps the message.

FHE libraries implement different Boolean circuits to perform homomorphic gate operations at the server. While FHEW supports AND, NAND, OR and NOR gates, TFHE provides all the basic gates. In this work, we assume the client wants to compute homomorphic NAND operations on its encrypted data on the cloud server. One must note that any Boolean circuit and function could be homomorphically implemented at the server given the availability of basic gates in the libraries. We choose NAND gate as it is a universal gate and any logical circuit can be implemented using proper chaining of NAND gates. We further assume that the server is untrusted and malicious and intends to extract sensitive information from client's data. Being in control of the FHE libraries, the server could perform any homomorphic operation on the client's data, in addition to the operation requested by the client. Finally, we highlight that the client provides *feedback* to the server when an expected computational result comes out to be incorrect at the client end after decryption. In other words, the server is able to observe the *reaction* from the client only when the

[3] An encrypted version of the secret key, called Bootstrapping key, is involved at the server end.

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

decryption results in a failure. The overall process of receiving client's feedback is shown in Fig. 1.

V. Client as the Decryption Verification Oracle

In the context of cloud computing, the server stores private information of its clients in encrypted form, thereby ensuring security and privacy of client's data. However the server, being untrusted, acts as a potential adversary and carefully introduces perturbations on the stored data of the client and then checks if these modifications trigger any error later in the process. The objective of the attacker is to ascertain the exact value of the random noise (error) in the resulting ciphertext obtained after the homomorphic gate computation. Since this ciphertext will still be encrypted under the original secret key, obtaining the errors in these ciphertexts will also lead to extraction of the secret key. As already discussed (cf. Section II), the mathematical robustness of LWE-based schemes is based on the intractability of both the secret key and the random error. Thus, leakage of these noise values can trivially leak the underlying secret key. Also, for a key size of k bits, at least k ciphertext messages with their corresponding error values is required to retrieve the key.

In cloud setting, the effect of introducing faults in the data cannot be directly observed by the server. Therefore, the malicious server needs reaction or feedback from the client to understand the effect of the purposeful faults [9]. As explained in Section IV, the client being in a pay-per-use model, could insist the server for recomputation of homomorphic operations on specific ciphertexts if a decryption error is observed. However, the adversary would have to send perturbed ciphertexts, only corresponding to those which are queried by the client. In other words, the attacker should be able to extract information by inducing errors in ciphertexts which are resultant of an intended homomorphic query. Rather, it has to introduce measured perturbations in the client's valid ciphertext in order for the client to decrypt it and send feedback to the server on decryption error. As an example, suppose a client intends to perform homomorphic encryptions like AES (Advanced Encryption Standard) [56], [57] on the cloud. The client before paying for the service and using it for a business would like to check the validities of the result by performing a Known-Answer-Test (KAT) [58]. In another instance, the client may be paying for a machine learning as a service (MLaaS) on encrypted data. In case of inferior results, the client may subsequently place a log to the server to indicate the pathological cases. We essentially discuss how such a log can be utilized by the server in turn to determine the secret key. However, the challenge in this case is that the server does not have information regarding the plaintext for a corresponding ciphertext. Moreover, the random error values introduced into the ciphertext during encryption is sampled from a Guassian distribution where the sign of the error could be either positive or negative. Thus, in order to obtain the exact error value, the server first needs to determine the value of the corresponding plaintext message and the sign of the error value.

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

In the following subsection, we explain the idea of the attack wlog. when the client intends to perform homomorphic NAND computations on the cloud. We choose NAND gate as it is a universal gate. However, before proceeding with the description of the attack, we would like to clarify why the decryption verification oracle is not a decryption oracle by taking the example of a homomorphic NAND computation.

Why Decryption Verification Oracle is not a Decryption Oracle?: Consider a homomorphic NAND gate computing on ciphertexts corresponding to messages x_1 and x_2, resulting in the ciphertext for the plaintext data $r = NAND(x_1, x_2)$. Let the ciphertexts be denoted as C_{x_1}, C_{x_2}, and C_r respectively. In our attack, we are considering situations wherein the adversarial cloud server perturbs the ciphertexts C_r and sends it to the client. The information the adversary exploits is the reaction of the client on a decryption error. It may be observed that the existence of a decryption error leaks the difference of the plaintexts corresponding to the ciphertexts C_r and its noisy version, i.e. $r \oplus r'$. On the contrary, a decryption oracle would leak the information of r, which is not leaked with the information in case of the decryption verification oracle. This shows that the attack we discuss is not an IND-CCA attack, but rather threatens the IND-CVA security of FHE schemes.

A. Recovering the Plaintext and Error Sign

As discussed, the client encrypts a stream of ciphertexts and sends them to the server for homomorphic NAND computations. We note that the FHE schemes discussed in this paper perform bit-wise encryption of the plaintext messages and then 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. Moreover, as per the truth table of NAND gate (as shown in Fig. 2(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. The server can use this bias to recover both the plaintext message and the error sign by craftily introducing additional error into the final computational result and sending it back to the client for

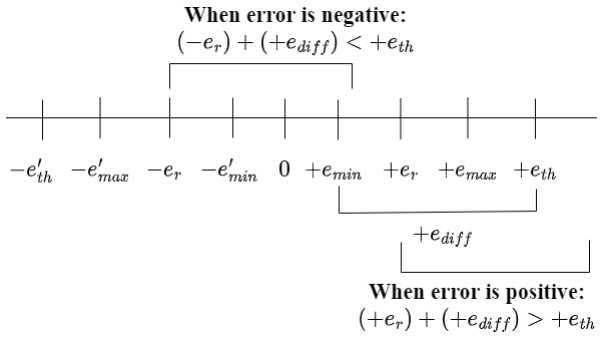

Fig. 3: Different bounds of errors plotted on a number line.

its reaction. We will like to highlight the fact that such biases exist for many other gates as well. For example, in 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 an encryption of 0 instead of encryption of 1. It must be mentioned that the bias in gate operation aids the attack but is not a necessary condition for the attack to work. *Even for balanced gates (like XOR, XNOR), our attack would still work, albeit requiring to perturb more ciphertexts which in turn requires more CVO queries.*

B. Perturbing Computed Result

With the stream of ciphertext messages at the helm of the server, it can now launch "reaction" attacks on randomly chosen ciphertext samples of the client. We assume wlog. that out of m ciphertexts sent by the client to the cloud, the server randomly samples n ciphertexts, where $m \gg n$, to introduce purposeful perturbations. This is a reasonable assumption in the cloud computing setting as the ciphertexts are essentially encryptions of single bit information and in order to obtain a meaningful computation from the server, the client would need to send a large number of ciphertexts. 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 take 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 bootstrapping operation to retain homomorphy. Otherwise once the accumulated error crosses the pre-determined 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 suppose, for every ciphertext, the server knows whether the underlying error value lies within a certain range. More precisely, given a ciphertext C_r containing unknown error value e_r, the server precisely 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. 3. 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 $+e_{max}$, all of which are less than the error threshold $+e_{th}$. The converse is true for negative error values. Therefore, the following relations hold for both positive and negative error values.

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

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

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 $+e_{max}$. Let us denote the quantity $e_{th} - e_{min}$ as e_{diff}. Now, we add the term e_{diff} with the computed result of homomorphic gate operation. As depicted in Fig. 3, if the original error (after homomorphic gate operation) is negative, the final error after perturbation lies within the permissible range (less than threshold), albeit in the opposite sign domain. In contrast, when the error is positive, the final error after addition of perturbation lies beyond the permissible range (more than the threshold) in the positive domain. Therefore, it is easy to note that the decryption failure would only occur when the original error is positive.

Elimination of probable choices: While the malicious server could perturb the output ciphertexts to instigate 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 ciphertext 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 in the preceding paragraph), the server adds the error term e_{diff} 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 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 with the expected result.

Considering r as the expected plaintext, sgn as the sign of the error and R denoting whether a feedback is received from the client, we record the different combinations of these events from the above mentioned four cases. Fig. 2(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 a feedback from client, 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. 2(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 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 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 the 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 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 FHEW library [2] does not allow homomorphic gate evaluations on a pair of related ciphertexts, where both the inputs are either same or one is the complement of the other. However the validation of whether the inputs are related or not is performed over the server side as part of the homomorphic gate evaluation, and thus can be simply disabled.

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 ← e_min
end ← e_max
e_temp ← 0
function GETERRORPOSITIVE(c, start, end)
    if start == end − 1 then return e_temp
    else
        mid ← ⌊(start+end)/2⌋
        e_diff ← e_th − mid
        c ← c + e_diff = a · s + x_r + e_r + e_diff
        feedback ← CVO(c)
        c ← c − e_diff = a · s + x_r + e_r + e_diff − e_diff
            = a · s + x_r + e_r
        if feedback = "correct decryption" then
            e_temp ← mid
            GETERRORPOSITIVE(c, start, mid)
        else
            GETERRORPOSITIVE(c, mid, end)
        end if
    end if
end function
```

Therefore, *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 attack to recover the secret key.

VI. Recovering The Original 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 faulty ciphertexts to the client, while awaiting its reaction. In Section V, 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 additional error introduced into the ciphertext result can be computed as $e_{temp} = e_r + (e_{th} - e_{min})$ 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 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 (since we are considering the entire range of error, the recovered error will always be correct). 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 then calculate the error term to be added as $e_{diff} = e_{th} - mid$. This additional 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. 3), 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 a 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 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 does lie 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 towards 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 [59]. The number of such ciphertext required to create the system of equation 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 retrieve the key. We note that the number of ciphertext required to launch the attack is in the order of the size of the key, more precisely, $\Omega(k)$.

VII. Exploiting Gaussian Nature of Timing and Error Distribution

The reaction-based attack on FHE schemes presented in Section VI requires a considerable number of additional queries to the CVO to retrieve the original error value corresponding to each ciphertext. Considering a practical scenario where the client provides a stream of ciphertext to the server for computation and verifies the result at its end (as part of the initial service quality trial phase), a large number of erroneous results could create suspicion as well as distrust in the quality of service on the server. Such incidents could entice the client to change the secret key or discard the service altogether, thereby dampening the entire purpose of the attack. Therefore, to make the attack stealthy, the server needs to reduce the number of queries per ciphertext, which in turn, reduces the total number of queries required to recover errors for $\Omega(k)$ ciphertexts.

In this section, we show how the server can utilize the execution time of homomorphic operations, to ascertain a lower and upper bound on the error ranges and thereby reduce the number of queries per ciphertext. In libraries like TFHE and FHEW [5], the error is sampled from a "Gaussian distribution". Therefore, the error values towards the *tails* of this distribution are less likely to appear when randomly drawn. In other words, since during encryption, the errors are sampled from a Gaussian distribution, some error values have a higher probability of being sampled than others. Also, since the final ciphertext of a homomorphic gate computation is a linear combination of the input ciphertexts, the error in the final ciphertext is also a linear combination of errors in the input ciphertexts and thus follows a Gaussian distribution. We exploit this observation to further reduce the probable range of errors by creating *timing buckets* containing error values and performing a *bucket-matching analysis* for unknown ciphertexts. To summarize, if the adversary can identify the range $(\pm e_{min}, \pm e_{max})$ such that the error value for a particular ciphertext has a high probability of lying within that range, it can reduce the overall error search space. While this approach will require perturbing more ciphertexts, it will reduce the overall number of queries required to recover errors for $\Omega(k)$ ciphertexts. The reason for requiring to perturb more ciphertexts is that, for certain ciphertexts that are encryption of 1 and have $+ve$ error value, i.e, the ones we are targeting (cf. Section V-B), the error value might not lie in the range $[+e_{min}, +e_{max}]$. Our proposed algorithm will detect such a case and will not proceed to recover the error, the reason for which has been explained later in this section. Let us understand the scenario with an example of a ciphertext C_r with error $+e_r$. Based on the value of $+e_r$, one of the following three conditions will arise:

❶ $\underline{+e_r < +e_{min}}$**:** Referring to Fig. 3, $+e_r$ will lie to the left of $+e_{min}$. If we add $e_{diff} = e_{th} - e_{min}$ to the ciphertext C_r, the overall error $e_{diff} + e_r$ will be less than e_{th}, and thus will not cause decryption error. We will not proceed with such ciphertexts and simply ignore them. This situation is similar to a ciphertext with $-e_r$ and consumes only 1 query.

❷ $\underline{+e_{min} \leq +e_r \leq +e_{max}}$**:** We have already covered this case in Section VI in detail. The number of queries required in this case will be $\lceil \log (e_{max} - e_{min}) \rceil$, which is lesser than $\lceil \log (e_{th} - 0) \rceil$ given $(e_{max} - e_{min}) < (e_{th} - 0)$.

❸ $\underline{+e_{max} < +e_r}$**:** In this case, if we add $e_{diff} = e_{th} - e_{min}$ to the ciphertext C_r, the overall error $e_{diff} + e_r$ will become greater than e_{th}, and thus will result in decryption error. The server will treat this as the case of a ciphertext with

[5]We have observed that the errors in LWE equation in FHEW also follow Gaussian distribution. Please refer Appendix **??** for further details.

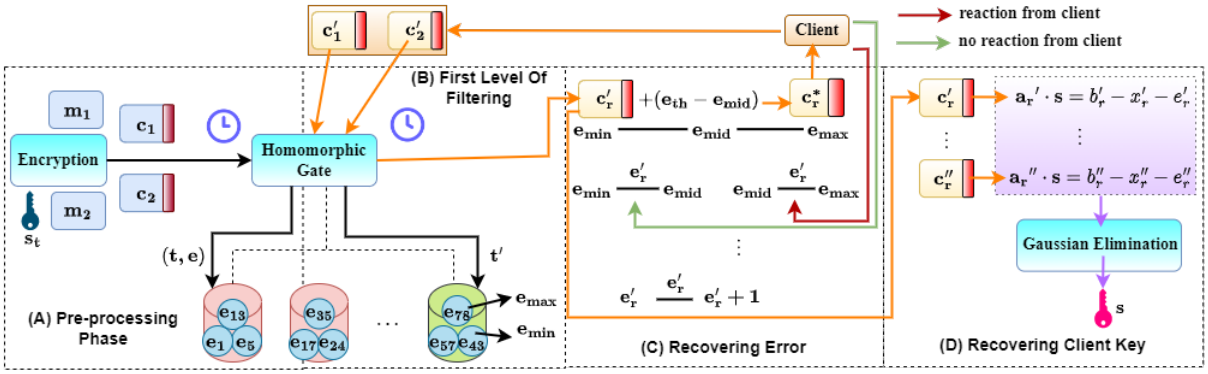

Fig. 4: End-to-End attack process showing (A) how buckets are generated, (B) how timing information is used as first level of filtering, (C) how reaction from client is used to reduce search space which ultimately leads to recovery of error, and (D) how recovered errors along with original ciphertexts are used to form a system of `exact` equations which are then solved using Gaussian Elimination to recover client key.

$+e_r$, and will thus proceed with recovering the error using Algorithm 1. However since the search space is limited to $[+e_{min}, +e_{max}]$ while $+e_r > +e_{max}$, the recovered error will be incorrect as the final error value returned will be $+e_{max}-1$. This is because the value of $start$ (cf. Algorithm 1), which is initially e_{min}, will keep on increasing until it becomes equal to $+e_{max}-1$. At this point, $start = end - 1 = +e_{max} - 1$ and thus the recursion will stop and the current value of $start$ will be returned. To prevent this from happening, we will perform an additional query by adding $e_{diff} = e_{th} - e_{max}$ to the ciphertext C_r. If $+e_r > +e_{max}$, then the total error $e_{temp} = e_r + (e_{th} - e_{max}) > e_{th}$ and will cause a decryption error. We will simply discard such ciphertexts and not proceed with recovering its error. However, this additional query will have to be performed for both cases ❷ and ❸ in order to differentiate among them. We would like to highlight the fact that this additional query will not be performed when the timing information is not used and the entire range of errors is considered. The reason for the same is that in such a scenario, the final error will always lie in the range considered and hence will always be recovered.

We note that our algorithm will *always be successful in recovering the correct error*. This is because cases ❶ and ❸ ensure that ciphertexts with error $+e_r < +e_{min}$ or $+e_r > +e_{max}$ are dropped and the server does not proceed with recovering the error for them. On the other hand the server will always be successful in recovering the error from ciphertexts falling under case ❷ as the underlying error will always follow the relationship $+e_{min} \leq +e_r \leq +e_{max}$, for which our algorithm will always converge to the correct error.

A. Identifying Outliers using Timing Analysis

We now explain in details how the server utilizes timing value of the homomorphic operations to reduce the error search space. The attack works in two phases - *pre-processing* and *actual attack*. Fig. 4 shows the overall process of our attack by utilizing the timing information. We now explain each of these phases.

Pre-processing Phase: The adversary starts by sampling a random secret key $\mathbf{s_t}$ by locally running the key generation process of the library. Once generated, the adversary chooses two messages x_0 and x_1 and obtains polynomially many encryptions of the same under the chosen secret key $\mathbf{s_t}$. It is easy to note that each of these encryptions will result in different ciphertexts, albeit containing different error values. In other words, for a given plaintext message x_j, multiple distinctive ciphertexts, $C_i = \mathbf{a_i} \cdot \mathbf{s} + x_j + e_i$ with different values of $\mathbf{a_i}$ and e_i, can be produced. The adversary can create any arbitrary number of such ciphertexts using the two messages. Once it generate sufficient number (empirically determined) of ciphertexts, it starts the next part of this phase to generate timing buckets, which is different for FHEW and TFHE schemes, owing to the way both these libraries are implemented.

For FHEW: The adversary does not modify the library and simply injects timing hooks at the beginning and end of the complete homomorphic gate evaluation which includes bootstrapping as well. It obtains the result of the entire gate computation $c = (\mathbf{a}, b)$ encrypting a message x_r and the time t required to perform this computation. Thus the adversary treats the entire gate computation operation as a black-box and does not tamper the underlying operations in any way. On the other hand, disabling the bootstrapping operation will cause the result to remain in the ciphertext space, which will result in incorrect decryption. Running the modulus switching operation alone is not sufficient to prevent this as the presence of high amount of error interferes with this operation. Thus it is not a choice but a necessity for an adversary to work in the black-box setting as opposite to TFHE where the ciphertext obtained from a gate computation running without bootstrapping will still decrypt correctly.

For TFHE: The adversary is free to run both the original as well as a modified homomorphic gate to obtain the result of the gate computation of $c = (\mathbf{a}, b)$ and the time t required to perform this computation. The situation is similar to that in FHEW when the original, unmodified gate is being run. On the other hand, the modified gate is obtained by making a copy of

the original gate into a new function, disabling the refreshing operation, and injecting timing hooks around the point where ciphertext addition or subtraction is taking place. We note that the adversary (server) has complete control of the library and can choose to make modifications in the source code. We performed our attack on both these versions of the library, i.e., one where the gate is modified to remove the refreshing step and the timing is observed around the point where ciphertext addition or subtraction is taking place, and other where the gate is not modified and the timing is observed for the entire homomorphic gate computation which includes the refreshing step. We show that disabling bootstrapping requires even lesser number of queries to recover the error from a single ciphertext as compared to when bootstrapping is enabled.

Now, as the adversary knows the secret key $\mathbf{s_t}$, it can simply extract the error e_r in the ciphertext by evaluating $e_r = b - \mathbf{a} \cdot \mathbf{s_t} - x_r$. This error e_r along with the execution time t obtained during homomorphic gate computation forms a timing trace (t, e_r). We obtain a varied range of timing values, say from t_{start} to t_{end}, for n ciphertexts, t_{start} and t_{end} being the smallest and largest timing values in the entire trace profile. We empirically select a timing interval, say δ, and segregate the entire timing range into $\lceil \frac{t_{end} - t_{start}}{\delta} \rceil$ number of buckets. For simplicity, let's denote a particular bucket as:

$\mathcal{B}_{t_{min}}^{t_{max}}(t)$: t_{min} and t_{max} represents the minimum and maximum timing values for this particular bucket.

t: a timing value such that $t_{min} \leq t \leq t_{max}$.

Next, for each timing value $t \in \mathcal{B}_{t_{min}}^{t_{max}}(t)$, we put the corresponding error value e_r into the bucket $\mathcal{B}_{t_{min}}^{t_{max}}(t)$. Therefore, at the end of the segregation process, all buckets contain a number of error values that correspond to the timing t where $t_{min} < t < t_{max}$ for a particular bucket.

It is worth mentioning that the size of the bucket δ is chosen on the basis of the timing values obtained during generation of traces and is independent of the range of error in the final ciphertext. In other words, the adversary is free to choose any value for δ based on the t_{start} and t_{end}, obtained during traces generation. The objective of the attacker is to choose an optimal value for δ in order to maximize the number of errors in a single bucket, while also ensuring that it does not encompass the whole timing range. In other words, the value of δ is chosen such that a single bucket contains a higher fraction, say .75 to .90, of the n traces, while the rest belongs to some other bucket(s).

Since it is this bucket that we are targeting to reduce the error range required to launch our CVO-based attack, it is necessary that the bucket size is chosen optimally so that the targeted bucket neither contains too high (which will increase the range of error and thus will require more number of queries per ciphertext) or too low (which will increase the chances of error of a sample lying outside the range and so will require perturbing more ciphertexts) number of errors. We also highlight that the bucket size δ might not be the same for every gate as the operations corresponding to each gate is different. Once such bucket is identified, we sort the errors in that bucket and set the lowest and highest positive error values in this bucket as $+e_{min}$ and $+e_{max}$ respectively. We also note down the values of t_{min} and t_{max} for this bucket. In this work, we majorly focus on the NAND gate and thus create buckets for that particular gate only.

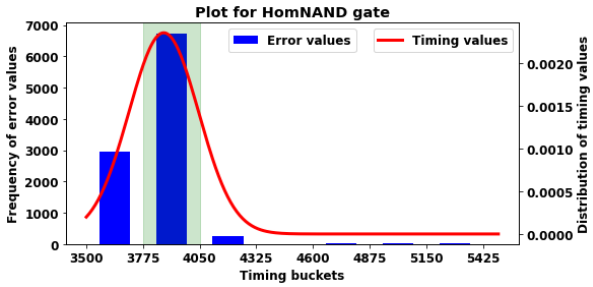

Fig. 5: Plot of timing distribution vs error distribution in timing bucket. The shaded region highlights the timing values between 3775 and 4050 that correspond to the timing bucket containing highest number of errors.

Actual Attack Phase: During the actual attack, the adversary obtains a fresh ciphertext pair from the client, encrypted under an unknown key, $\mathbf{s_k}$, that it is trying to recover. The adversary then runs the same gate operation for which the buckets were built (NAND in our case), on this new ciphertext pair. It receives as output a new ciphertext that encrypts the result of this computation with an increased error value in case of the modified TFHE, or which is further reduced due to bootstrapping in case of FHEW or the unmodified TFHE library. It also receives as output the corresponding timing value t'. Since the adversary does not know the secret key, it cannot recover the error e' of this new ciphertext [6] unlike it was able to do during pre-processing step. Finally, this timing value t' is compared with the timing range, $[t_{min}, t_{max}]$, of the bucket obtained previously, which contains the maximum number of errors. We only consider those ciphertext samples for which $t_{min} \leq t' \leq t_{max}$, and reject others. Once such ciphertexts are identified, we perform the error and key recovering process, as explained in section VI, by using the reduced range $[+e_{min}, +e_{max}]$ obtained using the bucket instead of the original $[0, +e_{th}]$.

B. Gaussian nature of Timing and Error distributions

The homomorphic gate operations in FHE libraries are essentially linear operations of LWE equations. Therefore, it is counter-intuitive why shall timing-based buckets would work in such a scenario. In this section, we provide the intuition on why and how the bucketing process helps in reducing the error search space using empirical results on TFHE. Fig. 5 shows the distribution of timing values which follows a Gaussian distribution (shown in red). It also shows count of errors in different buckets (shown in blue bars), where the (second) bucket with the highest number of errors represents the peak of this distribution, while the rest represents the tails. The timing and error values are obtained after running homomorphic NAND operation 10000 times on a pair of ciphertexts, in

[6] It is this error that we are trying to recover first which will then be used to recover the secret key.

TFHE. The highlighted (in green) portion shows the timing range, which is 3775 to 4050 in our case, that corresponds to the bucket with highest number of errors. We can clearly see that both these (Gaussian) distributions coincides, which implies that we can utilize the peak of the timing distribution to find the peak of the error distribution. Finally, Fig. 7 shows the frequency of errors obtained from the 10000 ciphertexts obtained as a result of the above homomorphic NAND computations. The shaded regions highlight the error ranges $(-e_{max}, -e_{min})$ (in green and to left) and $(+e_{min}, +e_{max})$ (in red and to right) obtained from the highest bucket. One can observe that the highlighted portions, obtained using the timing information, have reduced the search space for the error. In case of FHEW, the distribution of errors in the final ciphertext obtained after homomorphic NAND operation follows a Gaussian distribution, as evident from Fig. 6 which shows the plot of frequency of errors in 10000 ciphertexts, even though the error is sampled from a Chi distribution during encryption. Since the underlying timing distribution also follows a Gaussian distribution, the same can be used to infer the peak of the error distribution, as in the case of TFHE, and thus can be utilized to reduce the error range. Since this timing and error distribution is independent of the secret key s, the plaintext message x_r and the mask a_r, errors in the ciphertexts encrypted using any random key will also follow this distribution.

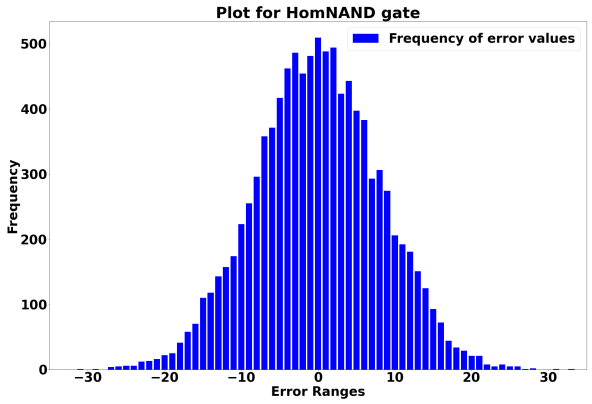

Fig. 6: Plot of frequency of errors for *10k* ciphertexts in FHEW.

VIII. Experimental Results

In this section, we provide the experimental results of the attack for both FHEW and TFHE. We first show the results for when the entire range of errors is considered, which is taken to be $[0, 63]$ and $[0, 10200547327]$, and then show the results when the range of error is reduced using timing information. Here 63 and 10200547327 are the $+ve$ error threshold of homomorphic NAND gate for FHEW and TFHE, respectively. During experiment, we generated 3000 pair of random plaintext bits in both the cases, i.e., without and with timing information, except for the case where bootstrapping is disabled in TFHE in which we generated 6000 pair of random

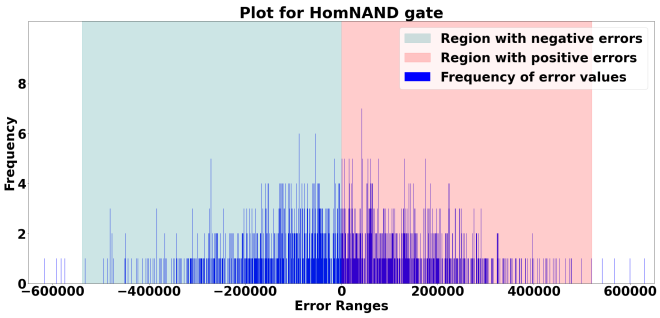

Fig. 7: Plot of frequency of errors in intervals of 100. The shaded regions on the left and right highlights the bound of negative and positive errors in the targeted bucket.

plaintext bits. 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 pair to obtain the corresponding computation result, upon which we ran our attack. Table I shows the count of ciphertexts out of these 3000 samples (6000 in case of TFHE without bootstrapping) for which we were able to successfully recover the error term and the total number of CVO queries made for the same. For a single ciphertext, we required 8 and 33 queries, when timing information is not used, and 8, 29 and 23 queries, when timing information is used, to recover the underlying error term for FHEW with bootstrapping, TFHE with bootstrapping and TFHE without bootstrapping, respectively. From the table, we can observe that our improved attack has reduced, on average, the number of CVO queries by almost 10000, 20000 and 4000 when using the timing information with bootstrapping enabled and disabled for TFHE, and with bootstrapping for FHEW, respectively. However, in both instances the number of samples for which the errors were successfully recovered has also reduced, which was expected due to our two level filtering based on the timing value and error range. For our improved attack using the timing information, we started with creating the timing buckets. We fixed the input plaintext pair as $x_0 = 1$ and $x_1 = 0$, wlog, and generated a secret key $\mathbf{s_t}$ using the key generation function of the libraries. We then obtained timing traces for 10000 ciphertext pairs of the above message pair under the same key. The traces were obtained for FHEW, TFHE with bootstrapping and TFHE without bootstrapping. Once obtained, we divided these traces into timing buckets of sizes 2500000, 500000 and 275 across the above three cases, respectively.

Fig. 8 shows the plot of total number of CVO queries made across the three cases of TFHE for 5 different keys. From the plot, we can observe that the total number of CVO queries made has reduced when timing information is used, which is even further reduced when bootstrapping operation is disabled. For the third case, i.e., with timing and without bootstrapping, we considered 6000 samples instead of 3000 samples used for the other two cases. The reason for doing so is that the bucket size in this case is quite small (275) and thus there is a higher chance for a timing trace generated during the attack phase to lie outside the range of the target bucket. However,

TABLE I: Total number of CVO queries made (and total number of samples for which errors were successfully recovered) for the two schemes, for 5 different, random keys.

		Key 1	Key 2	Key 3	Key 4	Key 5
FHEW	**Without Timing**	10737 (1071)	10865 (1090)	10576 (1042)	10289 (1009)	10630 (1054)
	With Timing (bucket size = 2500000)	7565 (766)	7744 (790)	5285 (524)	5664 (573)	7541 (771)
TFHE	**Without Timing**	36437 (1032)	37308 (1058)	38887 (1110)	37487 (1065)	38128 (1086)
	With Timing & With Bootstrapping (bucket size = 500000)	29200 (930)	29991 (959)	30991 (997)	25227 (806)	27714 (887)
	With Timing & Without Bootstrapping (bucket size = 275)	19838 (739)	17500 (646)	18624 (677)	17569 (650)	18206 (663)

we would like to emphasize that while we need to perturb more ciphertexts in the third case, we still require the lowest number of CVO queries which is one of our goals.

Once we recover the original error from ciphertexts, a system of equations is formed and then the same is solved to recover the entire key. We perform full key recovery for FHEW and TFHE with key size 500 and 630 bits on two different systems, a Desktop computer running Intel Xeon Silver 4210R @ 2.4GHz powered by Ubuntu 20.04, and a Desktop computer running Intel i7-7567U @ 3.5GHz powered by Ubuntu 18.04. We obtained similar results on both these systems, which implies that our attack is not machine specific and can be carried out using any machine. However we only report the results we obtained by running the attack on the Xeon-based machine as the processor is of server grade. In case of TFHE without timing, with timing and bootstrapping enabled and with timing with bootstrapping disabled, we perturbed 3000, 3000 and 6000 ciphertexts, respectively, and out of those, 1032, 930 and 739 ciphertext were suitably faulted to recover their error values. In case of FHEW with and without timing, we perturbed 3000 ciphertexts in both cases, out of which 1071 and 766 ciphertexts were suitably faulted to recover their error values. Finally, we ran Gaussian Elimination from SageMath9.0 and Python 3.8 to recover the entire secret key. In case of TFHE without timing information, the overall attack took around 3 hours and required 36437 CVO queries. With timing information, it took around 10 and 6 hours, starting from bucket matching to key recovery, and required 29200 and 19838 CVO queries, in case of TFHE with bootstrapping enabled and disabled, respectively. In case of FHEW, we required 0.5 and 1.5 hours and required 10737 and 7565 CVO queries, in case of without timing and with timing, respectively.

IX. Discussion and Future Direction

In this section, we discuss about few critical questions that can arise when one considers the practical implication of the attacks and the role of client in aiding the attack. We follow it up with some of the possible countermeasures against our proposed attack.

A. Some Practical Questions

Why would the client decrypt a modified ciphertext when it already has obtained a correct decryption previously?: To answer this question, we would like to state that the client has no way of knowing whether it has received a modified version of some previous ciphertext or a new ciphertext that is the result of a fresh computation. In other words, the server may perform a replay attack by re-sending a modified version of a previous ciphertext. One might argue that the client can simply check the first part, i.e., $\mathbf{a}$ of the ciphertext pair $(\mathbf{a}, b)$ and check whether it was part of any previous ciphertext or not. However this requires the client to store the results of all the previous computations, which requires both storage and processing, and is thus not practical.

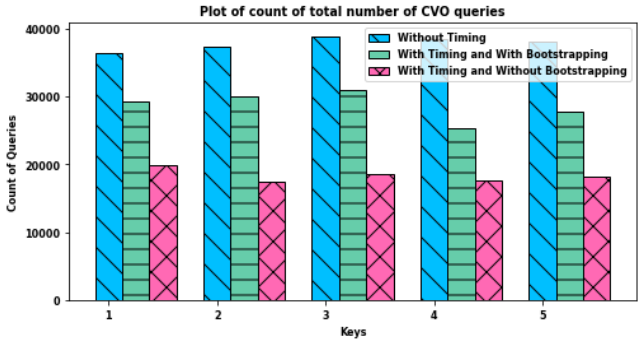

Fig. 8: Comparison of total number of CVO queries made for 5 different, randomly generated secret keys across the three cases in TFHE.

What if the user does not react and instead ask for a re-computation at a later time?: To answer this question, we would like to highlight first that the encrypted inputs are already stored on the server and the client has knowledge of their location to identify them uniquely. One may think of this location as the row index of a table that stores these ciphertexts. Whenever the client wants the server to evaluate a function on certain inputs, it simply informs the server of the function and the indices of the input ciphertexts. The server is free to maintain a log of these function along with its inputs and outputs. In case the user decide to ask for a re-computation at a later stage on the same inputs, it will mention their corresponding indices to the server along with the function to be re-evaluated. At this point the server can simply check its logs to obtain the previous results and resend the same along with another perturbation of previously perturbed ciphertext. To prevent this attack, the user may encrypt the inputs again to generate a new set of ciphertexts and re-transmit them, but then it defeats the whole point of storing the data in the cloud in the first place.

What if the server re-sends a modified ciphertext as part of the result of some later computation and gets no reaction from the client?: In this case, the server will not be sure whether it is due to correct decryption or that the decryption was incorrect but the output that it gave was what the user was expecting in the first place. To put this into an example, say a previously modified ciphertext c is an encryption of 1, which the server is aware of. The server modifies c, without knowing whether it will decrypt correctly or not, and replaces a ciphertext c' of some later computation with c which it then

sends to the client along with the other unmodified ciphertexts. The client decrypts the same and obtains 0 as a result, which implies that the decryption is not correct. But it so happens that the user was expecting 0 when it decrypts this ciphertext. Thus the user accepts the result even though there was an incorrect decryption and does not send any reactions to the server. One way to prevent this from happening is that the server can identify a gate or a set of gates (e.g., NAND, OR) in the final level of the circuit that has a high probability to output encryption of a certain bit (say 1) irrespective of the input values, and then can always send an encryption of that bit as the output of this gate.

B. Potential Countermeasures

The authors in [9] utilized the IND-CVA security model to attack the input data and underlying homomorphic function in a similar setting and proposed certain possible countermeasures. We will revisit two of their countermeasures to analyse the relevance in context to our proposed attack, as the authors have already refuted the other countermeasures. At last we also propose a possible countermeasure against our attack.

Obfuscate function and distrust server on decryption failure: First, the authors have proposed to obfuscate the underlying function that is being evaluated so that the server neither understands the function nor it can locate the position of important data bits. However, function obfuscation will be irrelevant in the context of our attack as we are only targeting the output of the function based on the circuit used to implement this function. Since an obfuscated function will still be implemented using these homomorphic Boolean gates, our attack will still be relevant. Finally, they have proposed to distrust the server immediately when a decryption failure occurs. However the decryption can still fail with a small probability even when the ciphertext has not been tampered with. Also this is not a good countermeasure in practice, as the client will have to look for a trusted server that does not tamper with its data thus nullifying the whole purpose of homomorphic encryption.

Using Authenticated Encryption: A natural solution to this perturbation-based attack could be to use authenticated encryption [60], [61]. In such schemes, the tag verification fails if the accompanied ciphertext is perturbed. The user will not decrypt such ciphertexts, and thus will always react irrespective of whether the ciphertext was supposed to decrypt correctly or not. However classical AE schemes do not allow homomorphic computations on the ciphertexts or the associated tags. To overcome this drawback, the ciphertexts and their corresponding tags are encrypted using FHE schemes, possibly TFHE or FHEW as they support fastest bitwise homomorphy. To put the above into perspective, let us have two pairs of AE based ciphertexts (C_1, T_1) and (C_2, T_2) upon which we want to perform an operation "$\circ$" to obtain a final pair (C_3, T_3), such that $C_3 = C_1 \circ C_2$. The input AE ciphertexts are first converted into homomorphic ciphertext pairs (C_1^F, T_1^F) and (C_2^F, T_2^F), upon which the operation "$\circ$" is evaluated homomorphically to obtain the resultant pair (C_3^F, T_3^F) which is homomorphic encryption of (C_3, T_3). In case the ciphertext C_3^F is not perturbed, the pair $(C_3^F,\ T_3^F)$ will decrypt correctly to the pair $(C_3,\ T_3)$, which is a valid ciphertext-tag pair. However in case the ciphertext C_3^F is perturbed, two cases will arise. If the error in the perturbed ciphertext does not cross the threshold, then the pair $(C_3^F,\ T_3^F)$ will decrypt correctly to the pair $(C_3,\ T_3)$, which is a valid ciphertext-tag pair. On the other hand, if the error in the perturbed ciphertext crosses the threshold, then the pair $(C_3^F,\ T_3^F)$ will decrypt to a different pair $(C_3^*,\ T_3)$, which is an invalid ciphertext-tag pair. The client will react in the latter case, while it will not react in the former case. This shows that the reaction-based attack will work even if authenticated encryption is used. Additionally, the server will get a reaction even without the client decrypting the received ciphertext as the client need not know the result of the decryption beforehand for it to react or not.

Countermeasure with reaction restriction: Since our attack is based on the reaction of the client, it makes sense to limit the number of such reactions over a pre-determined period of time. However, our proposed attack does not demand consecutive reactions or re-computation requests. Server can induce erroneous computations with series of correct computations and collect and store the ciphertexts from the client feedback over a period of time. So, it is difficult to define the time range for which the restriction on the number of re-computation requests can be imposed. Second, as explained in the plaintext recovery step of the attack, the cloud can forcefully perturb the ciphertext to an encryption of 0 and observe the client's reaction. It may happen the client will simply accept the message as it was expecting a 0 without any reaction. This no reaction (or passive reaction) is also a leakage about the original ciphertext generated from the homomorphic evaluation. Hence, only restricting the number of recomputation requests may not fully alleviate this attack possibility. However, modifying the decryption step with threshold cryptosystem [62] can be promising against this attack. In this case, the secret key is divided among N users as their shares such that any subset of t or more shares can be used to decrypt a ciphertext encrypted under the original secret key, but any subset of $t-1$ shares or less cannot be used to do so. Verifying this countermeasure will be taken as a future work.

X. Conclusion

In this paper, we have shown that access to a CVO can result in 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 error from a single ciphertext for the libraries FHEW and TFHE (with bootstrapping), respectively. Using timing information, we require 8, 29 and 23 such queries for the libraries FHEW and TFHE, with and without bootstrapping, respectively. Thus timing information is not necessary to perform our attack, however it aids in reducing the number of queries which helps in keeping the overall number of incorrect decryptions to be low. While CVO-based attacks exists in literature, in this paper

we showed such an attack to recover the full secret key on practical schemes that are being used in 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* [9] 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.

REFERENCES

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption library," August 2016. https://tfhe.github.io/tfhe/.

[2] L. Ducas and D. Micciancio, "FHEW: A fully homomorphic encryption library," May 2017. https://github.com/lducas/FHEW.

[3] "Microsoft SEAL (release 4.0)." https://github.com/Microsoft/SEAL, Mar. 2022. Microsoft Research, Redmond, WA.

[4] S. Halevi and V. Shoup, "Design and implementation of helib: a homomorphic encryption library." Cryptology ePrint Archive, Paper 2020/1481, 2020. https://eprint.iacr.org/2020/1481.

[5] S. Halevi and V. Shoup, "homenc: An implementation of homomorphic encryption," November 2019. https://github.com/homenc/HElib.

[6] D. Cousin, K. Rohloff, and Y. Polyakov, "Palisade homomorphic encryption software library," December 2019. https://gitlab.com/palisade/palisade-release.

[7] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library." Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[8] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "HEAAN: Homomorphic encryption for arithmetic of approximate numbers," May 2016. https://github.com/snucrypto/HEAAN.

[9] I. Chillotti, N. Gama, and L. Goubin, "Attacking fhe-based applications by software fault injections." Cryptology ePrint Archive, Paper 2016/1164, 2016. https://eprint.iacr.org/2016/1164.

[10] J. Loftus, A. May, N. P. Smart, and F. Vercauteren, "On cca-secure somewhat homomorphic encryption," in *In Selected Areas in Cryptography*, pp. 55–72, 2011.

[11] Z. Zhang, T. Plantard, and W. Susilo, "Reaction attack on outsourced computing with fully homomorphic encryption schemes," in *Information Security and Cryptology - ICISC 2011* (H. Kim, ed.), (Berlin, Heidelberg), pp. 419–436, Springer Berlin Heidelberg, 2012.

[12] C. Hall, I. Goldberg, and B. Schneier, "Reaction attacks against several public-key cryptosystem," in *Information and Communication Security* (V. Varadharajan and Y. Mu, eds.), (Berlin, Heidelberg), pp. 2–12, Springer Berlin Heidelberg, 1999.

[13] Z. Hu, F. Sun, and J. Jiang, "Ciphertext verification security of symmetric encryption schemes," *Sci. China Ser. F Inf. Sci.*, vol. 52, no. 9, pp. 1617–1631, 2009.

[14] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Tfhe: Fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, p. 34–91, jan 2020.

[15] L. Ducas and D. Micciancio, "Fhew: Bootstrapping homomorphic encryption in less than a second," in *Advances in Cryptology – EUROCRYPT 2015* (E. Oswald and M. Fischlin, eds.), (Berlin, Heidelberg), pp. 617–640, Springer Berlin Heidelberg, 2015.

[16] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical gapsvp," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings* (R. Safavi-Naini and R. Canetti, eds.), vol. 7417 of *Lecture Notes in Computer Science*, pp. 868–886, Springer, 2012.

[17] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, p. 144, 2012.

[18] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012* (S. Goldwasser, ed.), pp. 309–325, ACM, 2012.

[19] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I* (T. Takagi and T. Peyrin, eds.), vol. 10624 of *Lecture Notes in Computer Science*, pp. 409–437, Springer, 2017.

[20] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Annual Cryptology Conference*, pp. 75–92, Springer, 2013.

[21] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, STOC '05, (New York, NY, USA), p. 84–93, Association for Computing Machinery, 2005.

[22] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010* (H. Gilbert, ed.), (Berlin, Heidelberg), pp. 1–23, Springer Berlin Heidelberg, 2010.

[23] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, (New York, NY, USA), p. 169–178, Association for Computing Machinery, 2009.

[24] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *ASIACRYPT (1)*, pp. 3–33, Springer, 2016.

[25] R. Lindner and C. Peikert, "Better key sizes (and attacks) for lwe-based encryption," in *Topics in Cryptology – CT-RSA 2011* (A. Kiayias, ed.), (Berlin, Heidelberg), pp. 319–339, Springer Berlin Heidelberg, 2011.

[26] S. Bai and S. D. Galbraith, "Lattice decoding attacks on binary lwe," in *Information Security and Privacy* (W. Susilo and Y. Mu, eds.), (Cham), pp. 322–337, Springer International Publishing, 2014.

[27] M. R. Albrecht, "On dual lattice attacks against small-secret lwe and parameter choices in helib and seal," in *Advances in Cryptology – EUROCRYPT 2017* (J.-S. Coron and J. B. Nielsen, eds.), (Cham), pp. 103–129, Springer International Publishing, 2017.

[28] L. Bi, X. Lu, J. Luo, K. Wang, and Z. Zhang, "Hybrid dual attack on LWE with arbitrary secrets," *Cybersecur.*, vol. 5, no. 1, p. 15, 2022.

[29] Q. Guo and T. Johansson, "Faster dual lattice attacks for solving LWE with applications to CRYSTALS," in *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* (M. Tibouchi and H. Wang, eds.), vol. 13093 of *Lecture Notes in Computer Science*, pp. 33–62, Springer, 2021.

[30] S. Bai, S. Miller, and W. Wen, "A refined analysis of the cost for solving LWE via usvp," in *Progress in Cryptology - AFRICACRYPT 2019 - 11th International Conference on Cryptology in Africa, Rabat, Morocco, July 9-11, 2019, Proceedings* (J. Buchmann, A. Nitaj, and T. Rachidi, eds.), vol. 11627 of *Lecture Notes in Computer Science*, pp. 181–205, Springer, 2019.

[31] K. Laine and K. E. Lauter, "Key recovery for LWE in polynomial time," *IACR Cryptol. ePrint Arch.*, p. 176, 2015.

[32] S. Arora and R. Ge, "New algorithms for learning in presence of errors," in *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I* (L. Aceto, M. Henzinger, and J. Sgall, eds.), vol. 6755 of *Lecture Notes in Computer Science*, pp. 403–415, Springer, 2011.

[33] M. R. Albrecht, C. Cid, J. Faugère, R. Fitzpatrick, and L. Perret, "Algebraic algorithms for LWE problems," *ACM Commun. Comput. Algebra*, vol. 49, no. 2, p. 62, 2015.

[34] M. R. Albrecht, C. Cid, J. Faugère, R. Fitzpatrick, and L. Perret, "On the complexity of the BKW algorithm on LWE," *Des. Codes Cryptogr.*, vol. 74, no. 2, pp. 325–354, 2015.

[35] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *J. Math. Cryptol.*, vol. 9, no. 3, pp. 169–203, 2015.

[36] M. R. Albrecht, F. Göpfert, F. Virdia, and T. Wunderer, "Revisiting the expected cost of solving usvp and applications to LWE," *IACR Cryptol. ePrint Arch.*, p. 815, 2017.

[37] M. Chenal and Q. Tang, "On key recovery attacks against existing somewhat homomorphic encryption schemes," in *Progress in Cryptology - LATINCRYPT 2014* (D. F. Aranha and A. Menezes, eds.), (Cham), pp. 239–258, Springer International Publishing, 2015.

[38] T. Espitau, A. Joux, and N. Kharchenko, "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*, (Berlin, Heidelberg), p. 440–462, Springer-Verlag, 2020.

[39] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology — CRYPTO '96* (N. Koblitz, ed.), (Berlin, Heidelberg), pp. 104–113, Springer Berlin Heidelberg, 1996.

[40] C. Ashokkumar, R. P. Giri, and B. L. Menezes, "Highly efficient algorithms for aes key retrieval in cache access attacks," *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 261–275, 2016.

[41] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (K. Fu and J. Jung, eds.), pp. 719–732, USENIX Association, 2014.

[42] B. Timon, "Non-profiled deep learning-based side-channel attacks with sensitivity analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, p. 107–131, Feb. 2019.

[43] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, and A. Raychowdhury, "Practical approaches toward deep-learning-based cross-device power side-channel attack," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, pp. 2720–2733, dec 2019.

[44] F. Aydin, E. Karabulut, S. Potluri, E. Alkim, and A. Aysu, "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*, DATE '22, (Leuven, BEL), p. 1527–1532, European Design and Automation Association, 2022.

[45] F. Aydin and A. Aysu, "Exposing side-channel leakage of seal homomorphic encryption library," in *Proceedings of the 2022 Workshop on Attacks and Solutions in Hardware Security*, ASHES'22, (New York, NY, USA), p. 95–100, Association for Computing Machinery, 2022.

[46] "Microsoft SEAL (release 3.2)." https://github.com/Microsoft/SEAL, Feb. 2019. Microsoft Research, Redmond, WA.

[47] C. Gentry, C. Peikert, and V. Vaikuntanathan, "Trapdoors for hard lattices and new cryptographic constructions," in *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pp. 197–206, 2008.

[48] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key {Exchange—A} new hope," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 327–343, 2016.

[49] D. Micciancio and C. Peikert, "Trapdoors for lattices: Simpler, tighter, faster, smaller," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 700–718, Springer, 2012.

[50] S. Agrawal, D. Boneh, and X. Boyen, "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.

[51] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "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.

[52] V. Lyubashevsky, "Lattice signatures without trapdoors," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 738–755, Springer, 2012.

[53] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "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.

[54] A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 719–737, Springer, 2012.

[55] M. Chenal and Q. Tang, "On key recovery attacks against existing somewhat homomorphic encryption schemes," in *Progress in Cryptology - LATINCRYPT 2014 - Third International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014, Revised Selected Papers* (D. F. Aranha and A. Menezes, eds.), vol. 8895 of *Lecture Notes in Computer Science*, pp. 239–258, Springer, 2014.

[56] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced encryption standard (aes)," 2001-11-26 2001.

[57] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the aes circuit," in *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*, (Berlin, Heidelberg), p. 850–867, Springer-Verlag, 2012.

[58] L. E. B. Iii, "The advanced encryption standard algorithm validation suite (aesavs)," 2002.

[59] H. Herrn and R. Gauss, "Anwendung der wahrscheinlichkeitsrechnung auf eine aufgabe der practischen geometrie. von herrn hofrath und ritter gauss.," *Astronomische Nachrichten*, vol. 1, no. 6, pp. 81–86, 1823.

[60] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings* (T. Okamoto, ed.), vol. 1976 of *Lecture Notes in Computer Science*, pp. 531–545, Springer, 2000.

[61] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), p. 98–107, Association for Computing Machinery, 2002.

[62] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P. M. R. Rasmussen, and A. Sahai, "Threshold cryptosystems from threshold fully homomorphic encryption," in *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I*, (Berlin, Heidelberg), p. 565–596, Springer-Verlag, 2018.

APPENDIX A
CVO ATTACK PROOF-OF-CONCEPT CODE ON TFHE WITHOUT TIMING INFORMATION

As a proof-of-concept for our CVO-based attack on TFHE library, we wrote a set of scripts to emulate the behaviour of client and server during our attack. These set of scripts can be found at https://github.com/SEAL-IIT-KGP/CVO-TFHE, inside the directory `TFHE/Code 1`. This code does not rely on timing information and serves as a demonstration of how a CVO can be used to launch an attack on TFHE library.

To execute the PoC code locally, one may follow the detailed instructions provided inside the `readme.md` file. We recommend to use a Linux based machine, preferably Ubuntu based, as we used a Ubuntu based machine to develop this code. The necessary tools required to run the code can be downloaded and installed by running the `setup.sh` bash script. Once all the tools and libraries are installed correctly, three files, namely `user_testing.c` (containing code run by client to set up keys and input ciphertexts), `cloud_testing.c` (containing code run by server to perform homomorphic NAND operation on input ciphertexts) and `verify_testing.c` (containing our PoC code to perform perturbation and emulate CV Oracle), needs to be compiled using gcc compiler with the flag `-ltfhe-spqlios-fma`. It needs to be ensured that the output files for these compilations are set as `user_testing`, `cloud_testing` and `verify_testing`, respectively. The reason is that `user_testing.c` internally calls the executables `cloud_testing` and `verify_testing`, so changing the names of these output files will cause `segmentation fault`. Once the compilation is done, `user_testing` needs to be executed to run the PoC code. This execution will take around 3 hours to complete.

When the PoC code executes completely, we will be provided with a certain number of files, out of which we are only interested in the following two files:

- **gaussian_elimination.txt:** This will contain the samples whose errors were successfully recovered. We will run our key recovery script (equation_solver.py) on these ciphertexts.
- **count_of_oracle_accesses.csv:** This will contain the number of CVO queries made for each sample. These values were added up to find the total number of CVO queries made.

We now focus on the values for each ciphertext result provided in `count_of_oracle_accesses.csv`. In certain rows we find that only 1 query has been made to the CVO. This is because the corresponding ciphertext contains an error with `negative` sign, which does not fetch a reaction from the client and is thus simply dropped. In certain rows we find that only 2 queries have been made to the CVO. This is because the corresponding ciphertext contains an error with `positive` sign but the underlying plaintext bit is 0. Thus while the first case fetches a reaction from the client, the second one does not, which causes the ciphertext to be simply dropped. At last, there are certain rows where we find that 32 or 33 queries has been made to the CVO. This is because the corresponding ciphertext contains an error with `positive` sign and the underlying plaintext bit is 1. Thus both the first and the second cases fetches reactions from the client. This implies that we have found the target ciphertext for which we perform additional queries to recover the underlying error. Thus apart from the 2 queries we required to identify the target ciphertexts to be perturbed, we required 30 to 31 queries additionally to recover the underlying error value. To reduce these additional queries, we took the help of timing information to reduce the range of our search space.

`gaussian_elimination.txt` contains the value of vector $\mathbf{a}$ and scalar b that together forms the ciphertext, apart from the underlying plaintext message m, which is always 536870912 (the Torus equivalent of 1) as we are targeting an encryption of 1, and `recovered_error` which corresponds to the error value that is recovered using our attack. These samples are used to form our system of equations, where each of the equation corresponds to $\mathbf{a}\cdot\mathbf{s} = b - m - recovered\_error$, which is then solved using Gaussian elimination to extract the secret key $\mathbf{s}$.

APPENDIX B
WORKING OF PROOF-OF-CONCEPT CODE

We now explain how our code works. We are not going into the details of how the keys are generated, and the encryption and homomorphic operations are performed. What we do like to highlight is that these operations run in the standard way under the parameters defined in the TFHE library. We will only explain the steps carried out after the result of a homomorphic NAND gate is obtained by the server, as these steps forms the part of our actual attack.

To begin with, we define two variables `pos_min` and `pos_max` and initialize them with our error bounds. We also decrypt the result of the computation and store it as `ans`. We assume that the client knows that this is the result it should be obtaining after it has decrypted the ciphertext obtained from the client, and will react if this is not the result it obtains. This forms the basis of our ciphertext verification oracle.

```
 long pos_min = 0;
long pos_max = 10200547327;

int ans = bootsSymDecrypt(answer, key);
```

Here bootsSymDecrypt() is the unmodified decryption function, as defined in TFHE library, that takes the ciphertext `answer` to be decrypted and the secret key `key` as arguments, and returns the underlying plaintext bit.

We now call a function `get_sign` which performs the first two perturbations and decides whether to proceed with error recovery or not.

```
 get_sign(answer, pos_min, pos_max,
key, ans);
```

We need the value of `ans` to verify whether the perturbation caused incorrect decryption or not.

Inside the function `get_sign()`, to perform the first perturbation, we define `distance_from_threshold` (which represents e_{diff}) as the difference of `pos_threshold` (which represents e_{th}) and `pos_min` (which represents $+e_{min}$). We add this perturbation value to `sample` (which represented `e12`), decrypt it to obtain the resulting plaintext bit `ai`, and then remove this perturbation to get back to the original ciphertext. At this point we have made a query to the CVO oracle and so we increment the value of `count` by one. We check whether the plaintext bits `ans` and `ai` are same or not. If they are different, then it implies that the underlying error was `positive` as the added perturbation caused the error to cross the positive threshold. If this is the case then we proceed with the next perturbation, else we ignore this ciphertext.

```
 long distance_from_threshold =
pos_threshold - pos_min;
sample->b += distance_from_threshold;
int ai = bootsSymDecrypt(sample, key);
count = count + 1;
sample->b -= distance_from_threshold;

if(ans != ai) { proceed with next
perturbation }
else { reject sample }
```

Here `sample` represents the target ciphertext `answer`.

For second perturbation, we subtract a value of 2μ to `sample`, decrypt it to obtain the resulting plaintext bit `bi`, which will always be 0, irrespective of whether the original ciphertext was an encryption of 0 or 1, and then add back the value of 2μ to `sample`. At this point we have made another

query to the CVO oracle and so we increment the value of count by one. We check whether the plaintext bits ans and bi are same or not. If they are different, then it implies that the original ciphertext sample was an encryption of 1 as we know that the perturbed ciphertext will always decrypt to 0, for the reason explained previously in the paper. If this is the case then we proceed with our error recovery process, else we ignore this ciphertext.

```
 sample->b -= (2*MU);
int bi = bootsSymDecrypt(sample, key);
count = count + 1;
sample->b += (2*MU);

if(ans != bi)
{
error_1 = get_error_if_positive(sample,
pos_min, pos_max, key, ans);
}
else { reject sample }
```

If both the perturbations causes decryption failure, and thus fetches reactions from the client, we proceed with calling a function get_error_if_positive performs our binary-search based algorithm to extract the underlying error.

Inside the function get_error_if_positive, we define mid as the average of start (which represents pos_min) and end (which represents pos_max). As the base case of recursion, we check whether start becomes equal to $end-1$, which implies that our search has converged to only one error. At this point we end our search and return the value in error, which will either contain the value of mid or will contain the default value of 0.

```
 long mid = (start + end)/2;
//base case for recursion
if(start == end-1) { return error }
```

Otherwise, we define distance_from_threshold (which represents e_{diff}) as the difference of pos_threshold (which represents e_{th}) and mid. We add this perturbation value to sample , decrypt it to obtain the resulting plaintext bit ci, and then remove this perturbation to get back to the original ciphertext. At this point we have made another query to the CVO oracle and so we increment the value of count by one.

```
 distance_from_threshold =
pos_threshold - mid;

sample->b += distance_from_threshold;
ci = bootsSymDecrypt(sample, key);
count = count + 1;
sample->b -= distance_from_threshold;
```

We now check whether the plaintext bits ans and ci are same or not. If they are same, then it implies that the result of decryption was correct which in turn implies that the underlying error lies between start and mid as the added perturbation did not caused the underlying error to cross the positive threshold. If this is the case then we reduce our search space to the bounds start and mid. If they are different, then it implies that the result of decryption was incorrect which in turn implies that the underlying error lies between mid and end as the added perturbation caused the underlying error to cross the positive threshold. If this is the case then we reduce our search space to the bounds mid and end.

```
 if(ci == ans) {//if decrypted
correctly
error = mid;
get_error_if_positive(sample, start,
mid, key, ans);
} else {
get_error_if_positive(sample, mid, end,
key, ans);
}
```

We would like to highlight that after each decryption of the perturbed ciphertext, we are removing the perturbation to restore back the original error as it is this error that we are trying to recover.