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. Present research shows that while there are possibilities of side channel exploitations on the client side targeting the encryption or key-generation processes, the encrypted data on the cloud is secure against practical attacks. 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 underlying core methodology of the attack is to exploit the dependence of the error in the ciphertexts to the timing of homomorphic computations. These correlations can lead to timing templates which when used in conjunction with the error-induced decryption errors as reported by the client can lead to an accurate estimation of the ciphertext errors. 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, where we need 7, 23 and 28 queries to the client for each error recovery respectively. We mounted full key recovery attack on TFHE (without and with bootstrapping) and FHEW with key sizes 630 and 500 bits with 1260, 703 and 1003 correct errors and 31948, 21273 and 9073 client queries respectively.**

***Index Terms*—FHE, LWE, timing attack, template attack, ciphertext verification attack, key recovery**

I. Introduction

Fully Homomorphic Encryption (FHE) schemes allow computations on encrypted data while ensuring the final result remains encrypted as well. In other words, it allows transformation of a collection of ciphertext messages for some plaintexts $\pi_1, \pi_2, \ldots \pi_n$ into a ciphertext for some function or circuit evaluation on plaintexts $f(\pi_1, \pi_2, \ldots \pi_n)$, without the knowledge of the underlying secret key. Such schemes are particularly useful in constructing privacy-preserving protocols in cloud computing scenario, where a user (client) can store its confidential data in encrypted form to a remote server and allow the server to process on the encrypted data without revealing the original form. The client is anyone who wants to access the services of the cloud and has a secret key with which it can encrypt its data. It then sends this encrypted data to the untrusted server to perform some computations on it. The server, in general, evaluates a known function on this encrypted data with the help on a publicly available bootstrapping key (BK) or evaluation key (EK), which is an encrypted form of the secret key of the client. The server then sends back the result of the evaluation, still in encrypted form, to the user in possession of the secret key, who in turn decrypts it to obtain the underlying plaintext result. Thus, once the data leaves the client machine, it remains in encrypted form during transmission as well as during computation. 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 this function.

The basic foundational assumption of FHE is that the server, where the computations are taking place, is untrusted, and shall not be allowed to obtain any information regarding client's data. Moreover, apart from providing computational services, these servers can store data, albeit in encrypted form, acting as a data storage service and also for future computational requirements of the client. The assumptions stem from the fact that the sensitive data stored in the cloud is encrypted with a key that the cloud does not have access to. Therefore, considering the underlying crypto-primitive being mathematically secure, the cloud could undertake unscrupulous activities that include tampering with the data in order to gain private information. Therefore, in modern-day cloud service settings, the server itself is considered as untrusted and malicious. It must be mentioned in this context that the objective of the attacker is to retrieve information regarding the client's data

while keeping the attack undetected, such as not to lose client's trust which would incite it to stop using the services. Therefore, the security evaluation of FHE schemes must take into account the practical aspects and provide a system-wide attack model, instead at the primitive level. Although well-known FHE schemes being mathematically and implementationally robust, the capability of the server to introduce perturbations in the client data (either while storage or during computation) provides a unique security challenge that becomes the crux of this paper. As observed by authors in [8], to make an informed choice on the security guarantees of homomorphic schemes, one needs to consider a broader view of the overall system, instead of focusing on the primitive only.

Quite interestingly, if the encrypted data gets corrupted in the server or the final computational result appears erroneous to the client (while decryption), the client, in general, can inform the server about the faulty data and request for re-computation. This happens in a pay-per-computation model, where the client pays the cloud for each correct computation. If it receives a wrong result, the client will ask the cloud for a free re-computation. Although such a situation is pretty common *in modern cloud services domain, where intentional or unintentional alteration of encrypted data is touted as a rather benign data integrity issue, we , for the first time in literature, analyse the security implications of such "feedback" procedures*. In this paper, we show that the seemingly simple integrity issue can be used to learn crucial information from the client, when the untrusted server is acting as an active adversary. We show that a malicious server can purposefully perturb client's data, during homomorphic computations and use the feedback from the client regarding the correctness of the output, to leak the entire secret key, thereby breaking the FHE scheme.

The security of FHE schemes relies on mathematically hard problems such as Learning With Errors (LWE) [2], or its ring variant Ring-LWE [3]. The intractability of these schemes depends upon the idea of *noise*, 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 once the noise reaches or is about to reach the threshold, 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 [37] 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 [4]. Both these schemes operate under binary plaintext space and performs bootstrapping immediately after evaluating a gate.

A. Motivation

The early attempts that tried to break the security of FHE schemes mostly targeted the underlying mathematical hard problem. Majority of the works either reported the asymptotic complexity of solving these problems [13]–[15] or reported the security level of the schemes built using these problems [16]. One of the advantages of using LWE or its variants as the foundational hard problem is that they are computationally secure and robust both in the classical and quantum world. Therefore, the FHE schemes that are implemented using these mathematical primitives are considered to be post-quantum secure as well. Although, these FHE schemes and their underlying mathematical primitives are found to be theoretically secure in both the worlds, the security evaluation of practical implementations of these schemes opens up a new facade. As established through decades of research [39]–[43], a mathematically secure cryptographic scheme could be jeopardised in practice due to implementational hindsight. The passive information leakage due to execution of a cryptographic scheme on a real-world device could lead to extraction of the underlying secret. Such passive attacks, collectively known as *side channel attacks* have been successfully used in compromising many real-world cryptographic schemes, inspiring crypto designers to incorporate various side-channel protection mechanisms while building any new primitive. However, the side channel implications in popular FHE libraries like FHEW [37] and TFHE [4] has not been widely explored in literature. Recently, a side-channel attack [6] has been shown on the client side running the encryption operation of a popular HE library [18] to recover the plaintext message, that is being encrypted, using a single power trace. One must note that the above-mentioned attack targets the error sampling phase at 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. Moreover, the essence of FHE lies in the fact that the cloud server is untrusted and thus, can attempt to snoop sensitive information from client's operations. 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.

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 as a side channel source.

- We provide a novel attack scheme using the error threshold for decryption and a reduced error range to induce faults in the ciphertext
- We, for the first time, demonstrate that the error values in the input ciphertext has a relationship with the execution time of homomorphic gate operation in popular FHE libraries like TFHE and FHEW. We utilize this error-timing relationship to perform template attacks in order to ascertain a range of error that bounds the original unknown error in the computed ciphertext.
- Unlike prior attack, we target the server side where the secret key is not directly involved in any computation and show that side-channels can still 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 as side-channel source. Once the errors are recovered, the secret key is extracted by forming a system of equations withn the ciphertexts and solving them using Gaussian Elimination method.

II. Background

In this section, we provide a brief background on the Learning With Error 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 Error problem

The idea of Learning With Error was introduced by Regev in 2005 [2]. Since its inception, LWE has been used as a foundation of multiple cryptographic constructions 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 $(\mathbf{a}, b) \in \mathbb{Z}_{q}^{\ k} \times \mathbb{Z}_{q}$, where $\mathbf{a} \in \mathbb{Z}_{q}^{\ k}$ is chosen uniformly and $b = \mathbf{a} \cdot \mathbf{s} + e \in \mathbb{Z}_{q}$. 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 $s \in \mathbf{S}$.
- *Decision problem*: distinguish between LWE samples and uniformly random samples drawn from $\mathbb{Z}_{q}^{\ k} \times \mathbb{Z}_{q}$.

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. The Message Space

1) Torus Domain For TFHE: Torus [4] 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 which results in a torus element, while external product is defined as a product between an integer and a torus element which results in a torus element. Product between two torus elements is not defined.

In the CPU implementation of TFHE library [5], the Torus elements are defined as 32-bit unsigned integers and all the Torus-based operations are performed modulo 2^{32}. The plaintext messages 1 and 0 are represented as μ and $-\mu$ in the library, which are the 32-bit unsigned representations of the Torus equivalent of these messages.

2) Integer Domain for FHEW: The plaintext and ciphertexts as well as the underlying operations in the FHEW library [36] are defined over Integers modulo 512. The plaintext space is divided into two halves with each half either representing a 0 or 1. 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 [36] and TFHE [5]. The overall working principle of FHE schemes can be broadly broken into three stages - the *encryption stage* that takes place in client side and involves the encryption key, *homomorphic gate evaluation stage* on the server and finally *bootstrapping* to reduce the overall noise both of which does not directly involve the secret key, also at the server. 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 $e \in \mathbb{Z}_{q}$ 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 $\mathbf{a} \in \mathbb{Z}_{q}^{\ k}$ 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.

While FHEW and TFHE schemes operate in the secret key setting as shown above, the schemes can be converted to a public key setting. The owner of the secret key generates its public keys 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 product between a matrix and a vector. This matrix-vector pair $(\mathbf{A}, \mathbf{b})$ acts as its public key. To encrypt a message $x \in \mathbb{Z}_{\shortparallel}$, it randomly selects a row $(\mathbf{a}, b)$ from the public key and then adds x to b to obtain b'. The pair $(\mathbf{a}, b') \in {\mathbb{Z}_{\shortparallel}}^{k} \times \mathbb{Z}_{\shortparallel}$ 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: While the idea of homomorphic gate implementation and bootstrapping is identical for both FHEW and TFHE libraries, there are implementational differences, which we elaborate in this subsection.

FHEW: For server side computation, 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 $(2q, b_{gc})$, where the first part is a vector having each coefficient set to $2q$ and $q = 512$ denotes the modulus under which all the computations are carried out. The result $c = (\mathbf{a}, b)$ of the gate computation is evaluated by computing $\mathbf{a} = 2q - (\mathbf{a_1} + \mathbf{a_2})$ and $b = b_{gc} - (b_1 + b_2)$ where b_{gc} is defined differently for each of the 4 homomorphic gates. The process is same for evaluating any of the 4 gates, apart from NOT-gate, that are implemented in FHEW library. The bootstrapping operation takes place in the ciphertext domain, i.e., the modulus under which the ciphertext is defined, and the noise is reduced without a change in this domain. Once the noise is reduced, a key-switching procedure is carried out to switch back to the original key as refreshing operation changes the underlying secret key. Finally a modulus switching operation is carried out to switch the modulus from the ciphertext domain to plaintext domain.

TFHE: The server receives input ciphertexts in the same format described in case of FHEW, which are $c_1 = (\mathbf{a_1}, b_1)$ and $c_2 = (\mathbf{a_2}, b_2)$. The gate constants are defined as a pair $(\mathbf{0}, b_{gc})$ where the first part is a vector having each coefficient set to 0 and all the computations are carried out under modulo 2^{32}. The result $c = (\mathbf{a}, b)$ of the gate computation is evaluated by computing $\mathbf{a} = \mathbf{0} \pm (\mathbf{a_1} \pm \mathbf{a_2})$ and $b = b_{gc} \pm (b_1 \pm b_2)$ where the ordering of $+$ or $-$ depends on the homomorphic gate being evaluated. Similar to FHEW, the second part of the gate constant b_{gc} is defined differently for each of the 10 homomorphic gates (apart from NOT-gate) supported in TFHE library. The bootstrapping operation takes place in the ciphertext domain, similar to FHEW. However it reduces the noise in the ciphertext and switches back the modulus to plaintext domain from the ciphertext domain. Thus TFHE does not require a separate modulus switching operation. The only similarity between the implementation of the two schemes is that they both require a key-switching procedure to switch back to the original secret key.

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 $\langle \mathbf{a} \cdot \mathbf{s} \rangle$ 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, which it extracts by performing a rounding operation on the same. However the rounding 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 the papers [4] and [37], 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 [13] 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 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 the FHE libraries FHEW and TFHE. Interestingly, authors in [14] 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, [15] 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 [16] proposed a Dual attack on TFHE but the attack does not practically break the security of the scheme. Rather, the authors only reported a drop in the security level of the scheme from the one reported in the original TFHE scheme [17]. It must be noted that all the above mentioned attacks are generic and does not work well practically 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 targets the implementational aspects of FHE schemes. The first side-channel attack on HE has been demonstrated recently in [6], targeting the client side that is running the encryption operation of SEAL scheme [18]. The attack targets the conditional statements executed in the Gaussian Sampler routine to obtain the coefficients for the error polynomial. In the context of homomorphic encryption, attacking a client system poses realistic challenges and one must assume a stronger attacker model where the attacker has access to the client's machine. Whereas, the server, itself being untrusted, presents a more realistic scenario where it can try to decipher the data stored and computed at the server. However, to the best of our knowledge no reported attack successfully exploited the

server side computations due to non-involvement of the secret key directly in the computations done in the cloud.

IV. Attacker Assumptions and Threat Model

In this section, we present the basic security assumptions, valid for FHE and related cryptographic primitives. 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 cryptographic 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 [8]. On the other hand, it has also been established that no FHE schemes can be either IND-CCA [9], [11], [50] or IND-CCA2 secure, which implies that an adversary can break such schemes if it has access to a decryption oracle [8]. Our attack operates under the notion of IND-CVA (Indistinguishability against Ciphertext Verification Attack) security [10], which is based on the idea of "reaction" attack from [24]. 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. 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 [9], [11]. In such a model, the client could ask for a free re-computation in case the result returned by 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, 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 cloud setting where the server or cloud [1] 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. Therefore, in this work, we assume that the client can choose either of the variants. 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, *we target the decryption key, which is the secret key in both the cases.*

[1]We use the terms cloud and server interchangeably in this paper.

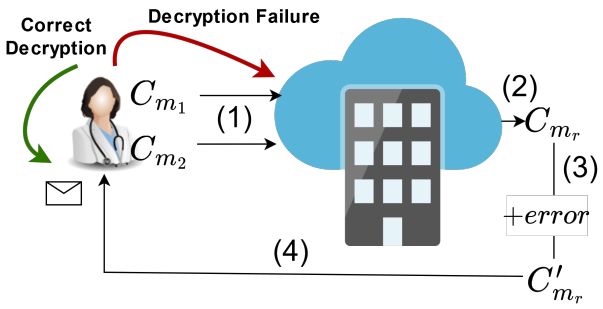

Fig. 1. Receiving client feedback works by (1) receiving two input ciphertexts from client, (2) computing a homomorphic gate, (3) introduce carefully crafted perturbation, and (4) sending this modified ciphertext to the client. The client sends a feedback if decryption fails, otherwise keeps the message.

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 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 schemes is based on the intractability of both the secret key and the random error, and

leakage of the noise can trivially leak the secret key. Also, for a key size of k bits, atleast 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. 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 a 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) 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) [51]. 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.

In the following subsection, we explain the idea of the attack wlog. when the client intends to perform homomorphic NAND computations on the cloud. Our choice of the NAND gate is motivated by the fact that the NAND is a universal gate. However, before describing 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 attacks, 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.

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, e 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).

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 in the output 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 its reaction.

B. Introducing perturbations in 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 and observe its feedback. Without loss of generality, we assume 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 encryption 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 depends on the size of the key used and 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 operation at the server, the total error in the computed result increases which is then brought down below

a pre-defined threshold e_{th} using bootstrapping operation. Without bootstrapping, the overall error in the final computed ciphertext would result in decryption failure at the client end. 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 [2]. 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, e_{min}, and a maximum value, e_{max}. However, the server does not have the knowledge about the sign of e_r, and therefore, the value 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 $+e_{max}$, where all the three quantities 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. $-e'_{th} < -e'_{max} < -e_r < -e'_{min}$ and $+e_{min} < +e_r < +e_{max} < +e_{th}$ 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 ciphertext outputs to instigate reaction from the client, there exist two major challenges that the server needs to deal with. ① the knowledge of the plaintext value for the corresponding ciphertext and ② the 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, denoted by C_r^0, or encryption of 1, denoted by C_r^1. 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. Now, depending on the input plaintext (r) of the perturbed ciphertext , one of the following four conditions will take place.

1) ***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*.
2) ***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.
3) ***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.
4) ***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 $+e_{th}$. But once again, 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 a feedback only when the sign of the error is $+ve$ and does not receive a feedback when the error is $-ve$. Thus presence or absence of feedback from the client leaks the sign of the error with probability 1.

Next, to recover the underlying plaintext message, we introduce another perturbation in the original ciphertext C_r. While the perturbation is almost of similar nature for these two cases, their effect on the underlying plaintext message is different for the two libraries. 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

[2]In the following sections, we will show that this error range can be precisely determined using timing side channel. Note that this does not impose any assumption on the threat model as in the FHE setting the server can do arbitrary computations with any library even with timing leaks.

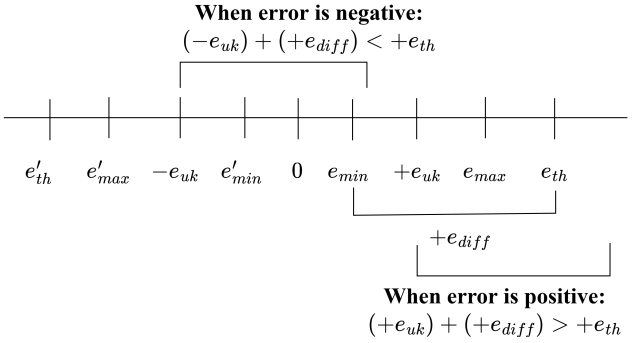

Fig. 3. Different bounds of errors plotted on a number line.

keeping 0 to remain same. This follows from the decryption function which is 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. Thus this observation reveals the underlying plaintext message to be 1.

In case of FHEW, we obtain a new ciphertext $C'_r = AND(C_r, NOT(C_r))$ which will always be an encryption of 0 irrespective of whether C_r was 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. Thus this observation reveals the underlying plaintext message to be 1. We would like to highlight here the fact that FHEW library [36] does not allow homomorphic gate evaluations on related pair of ciphertexts, i.e., when both the inputs are 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, which does make sense as the homomorphic gate evaluation is itself performed on the server. Since the server itself is an adversary, it can simply disable this validation.

Therefore, the plaintext message r will always be 1 for both TFHE and FHEW and the error $+ve$ for all the perturbed samples. For both TFHE and FHEW, the occurrence of $r = 1$ for all perturbed samples help us to capitalise on the output bias of NAND gate. 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 Range of Error in Ciphertext through Timing Channel

In the previous section, we discussed how a malicious server can act as an adversary and incite the innocent client to send feedback by carefully perturbing homomorphically computed ciphertext results. However, we assumed that the server knows the range, $[-e'_{min}, -e'_{max}]$ or $[e_{min}, e_{max}]$, where the actual error e_r, for each ciphertext, belongs to. Although it is intuitive that e_r would always belong in the range $[-e'_{th}, e_{th}]$ (after bootstrapping), a further reduction in the probable range makes the exact error recovery process efficient and real-time. In this section, we demonstrate how the exact error range, i.e, the values $-e'_{min}, -e'_{max}$ or e_{min}, e_{max} can be retrieved using timing channels during the homomorphic gate operations.

Exploiting the timing channel: For both TFHE and FHEW libraries, a homomorphic gate takes two LWE ciphertexts, say $C_{x_1} = \mathbf{a_1}\cdot\mathbf{s} + x_1 + e_1$ and $C_{x_2} = \mathbf{a_2}\cdot\mathbf{s} + x_2 + e_2$ as inputs and produces another LWE ciphertext as output, say $C_{x_r} = \mathbf{a_r}\cdot\mathbf{s} + x_r + e_r$, which is an encryption of the resultant message x_r under the same secret key $\mathbf{s}$. Thus the secret key remains the same across the inputs and output. On the other hand, while the secret key in the input ciphertexts remains the same during multiple homomorphic gate computations, the input error values change across these computations as they are sampled uniquely during each encryption. For example, n such computations will result in n different $\mathbf{a_r}$, x_r and e_r values but will have the same $\mathbf{s}$. Out of these, $\mathbf{a_r}$ is known as it is part of the ciphertext and x_r is being controlled by the adversary. Thus the only unknowns are $\mathbf{s}$ which is same for each ciphertext and the collection of e_r which varies across each ciphertext. Since these errors are results of addition or subtraction of input errors during a homomorphic gate computation, we observed during our experiments (more details in Section. VIII), that the execution time of the homomorphic gate computation varies with the value of the errors in the input ciphertext. In other words, changing the error in the input ciphertexts causes a change in the overall execution time of the homomorphic gate as well as the error in the output ciphertext. Since the timing value depends on the input errors, transitively it also depends on the output error. However, one cannot utilize these timing values to profile the error in the input ciphertexts as, for each timing value t, two possible error values e_1 and e_2 in the input ciphertexts C_{x_1} and C_{x_2} can be mapped to. On the other hand, the same timing value t also corresponds to the error e_r of the final ciphertext C_{x_r}. This relationship between the known timing value t and the unknown error e_r can be used to create the timing templates, which can be further used to infer the range of the error value in the final ciphertext.

Overview of the attack: We now provide a detailed overview of our proposed attack to recover the error in a ciphertext message which will be further used to recover the secret key in the later sections. More precisely, we perform the well-known template attack [7], which works in two phases - *offline* and *online*. In the *offline* phase, the adversary builds templates using a randomly chosen secret key. It must be

noted that since we aim to template the errors using timing channel, the choice of secret key does not have any influence in this case, thereby making the attack more practical. Once the templates are prepared using sufficient number of ciphertext and their corresponding homomorphic operations, then starts the *online* phase where the ciphertexts are encrypted with an unknown key. The objective of the attacker is to ascertain the template that could possibly match with the unknown error after the homomorphic computation. We would like to highlight that the biggest advantage an adversary can have in this case is that it only needs to *build templates using a single key of its own choice and the same templates can be used to mount attacks on ciphertexts generated using any random key*. The outcome of the template attack is a range of error values corresponding to each ciphertext, essentially signifying the upper and lower bounds of the actual error e_r, which will later be used to query the client to uniquely determine the original error value. Next, we explain each of these steps in further details.

A. Template Building Phase

The adversary starts by sampling a random secret key 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 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 of ciphertexts, it starts the next part of the template building phase which is different for FHEW and TFHE schemes, owing to the way both these libraries are implemented.

For TFHE: The adversary runs 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 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. Moreover, it can maintain two copies of the library where one version works as intended while the other one is manipulated for template attack. The server can decide whether to choose the original or modified or both versions for any ciphertext. 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.

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 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.

We emphasise that we discard the bootstrapping phase in case of TFHE library (modified version) while keeping the FHEW unchanged. The removal of bootstrapping phase makes the template building phase faster and helps in reduction of system noise [3], thereby making the process of template creation and matching efficient. However, the same attack can be performed even with the bootstrapping method included in the computation process.

Coming back to our attack, as the adversary knows the template generation secret key $\mathbf{s_t}$, it can simply extract the error e_r in the ciphertext by evaluating $e_r = b - \mathbf{a} \cdot \mathbf{s_t} - x_r$. Simultaneously, it executes homomorphic NAND operation (either in modified or original library for TFHE and original for FHEW) using the input ciphertexts C_{x_0} and C_{x_1} and records the execution time t. We denote the tuple (t, e_r) as a timing trace value. Due to the dependence of execution time of homomorphic gate computation on the input errors, a varied range of timing values, say from t_{start} to t_{end}, is obtained 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 $\frac{t_{end}-t_{start}}{\delta}$ 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_i into the bucket $\mathcal{B}_{t_{min}}^{t_{max}}(t)$. Therefore, at the end of the segregation process, all buckets must 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 can be chosen on the basis of the timing values obtained during template generation 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

[3]Not to be confused with the *noise* in the context of LWE. "System noise" refers to the disturbances in the timing measurement due to other processes (including kernel) running on the target device.

during template generation. The objective of the attacker to choose an optimal value for δ in order to obtain maximum accuracy during the template matching phase. If a higher value of δ is chosen, the chances of a ciphertext with an unknown secret key lying in the correct bucket, based on its timing value, will be higher. But that will cause some buckets to have more number of errors and some to have less, i.e., the distribution of errors will not be uniform across the buckets. Conversely, if δ is reduced, the uniformity of errors in various buckets will increase but the accuracy will decrease. 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. In this work, we majorly focus on the NAND gate and thus create templates for that particular gate only.

B. Template Matching Phase

In this phase 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 template was 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 [4]. Finally, this timing value t' is compared with the timing range, $[t_{start}, t_{end}]$, of each of the buckets $\mathcal{B}_{t_{min}}^{t_{max}}$ in the template to find the bucket whose timing range bounds this value t'. In other words, we are trying to find a bucket $\mathcal{B}_{t_{min}}^{t_{max}}(t')$ such that $t_{min} < t' < t_{max}$. Once such a bucket is identified, we infer that the error e' is bounded by the minimum and maximum error values e_{min} and e_{max} corresponding to this bucket, which we eventually utilize to launch the final attack.

Fig. 4 shows the overall process of template building and matching. The template building process (in the blue shaded area) starts with the adversary generating a random secret key and getting encryption c_1 and c_2 of two messages m_1 and m_2 of its choice. It runs the homomorphic gate on these two ciphertexts and observes the execution time t of this operation. It also extracts the error e from the final ciphertext generated as a result of this gate computation. The error e is then placed into the corresponding bucket $\mathcal{B}_{t_{min}}^{t_{max}}(t)$ satisfying $t_{min} < t < t_{max}$. The template matching phase (in the yellow shaded area) starts with the adversary running the homomorphic gate on two ciphertexts c_1' and c_2' received from the client and observing its execution time t'. It then finds its corresponding timing bucket $\mathcal{B}_{t_{min}}^{t_{max}}(t')$ and extracts the minimum and maximum error values from that bucket. These values serves as the bound on the error in the resultant ciphertext of the above gate computation.

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

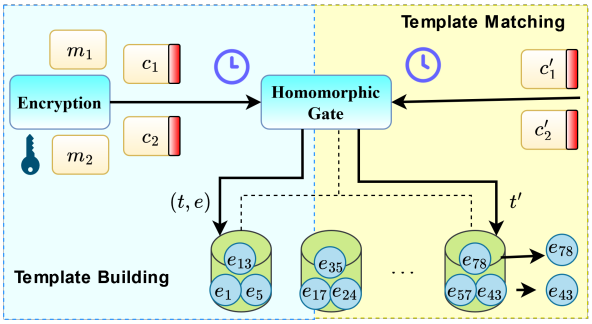

Fig. 4. Template building and matching process

Algorithm 1 Error Recovery using Binary Search

```
1:  e_th := positive error threshold
2:  e_min := minimum bound of error
3:  e_max := maximum bound of error
4:  c := ciphertext with the original error e_r
5:  start ← e_min
6:  end ← e_max
7:  e_temp ← 0
8:  function GETERRORPOSITIVE(c, start, end)
9:      if start == end − 1 then return e_temp
10:     else
11:         mid ← ⌊(start+end)/2⌋
12:         e_diff ← e_th − mid
13:         c ← c + e_diff = a · s + x_r + e_r + e_diff
14:         feedback ← CVO(c)
15:         c ← c − e_diff = a · s + x_r + e_r + e_diff − e_diff
                = a · s + x_r + e_r
16:         if feedback = "correct decryption" then
17:             e_temp ← mid
18:             GETERRORPOSITIVE(c, start, mid)
19:         else
20:             GETERRORPOSITIVE(c, mid, end)
21:         end if
22:     end if
23: end function
```

VII. Recovering The Original Error Value

Once the error bound has been recovered using template attack, the adversary proceeds to use active perturbations in the computed ciphertext result and iteratively sends faulty ciphertext to the client, while awaiting its reaction. In Section V, we explained how the adversary can uniquely ascertain the plaintext message of the homomorphically computed ciphertext and its corresponding error's sign ($+ve$ or $-ve$) 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 as obtained from the template attack phase, respectively. With the knowledge of error sign and a possible range, the server now recursively perturbs the original 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 $end - 1$, that means that there is only one error value left in the range, which is the original error e_r. 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 additional 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 in 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 $(e + e_{diff})$ to cross the positive threshold e_{th}. In such 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. The number of ciphertext with known error value 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 atleast k ciphertext with correct error values for solving the equations and retreive 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, $\Omega(k)$.

VIII. Experimental results

In this section, we provide the experimental results of the attack, starting with the template attack. To build the template, we chose $x_0 = 1$ and $x_1 = 0$, without any loss of generality. We first generated a random secret key $\mathbf{s_t}$ and obtained $10,000$ different ciphertexts of the above message pair under the same key. In order to verify the consistency of the template building phase across all types of gates, we independently executed the modified version of all the ten homomorphic gates (without bootstrapping) defined in the TFHE library, which resulted in $10,000$ timing traces (t, e). For FHEW, as already mentioned earlier, we do not modify the gates and acquire timing measurements for the entire process (gate operation and bootstrapping collectively). We collect traces for all the four gates supported in FHEW. All the gates were run independently such that the data is not stored in the cache and hamper the templating process due to caching. We further chose the bucket size as 500 and segregated the error values into these buckets based on their timing values.

In order to validate the accuracy of the templates, we further generated $1,000$ ciphertexts of the above message pair using another randomly generated secret key $\mathbf{s_k}$, different from $\mathbf{s_t}$. Once again, we independently execute the modified version of all the ten homomorphic gates defined in the TFHE library, which resulted in $1,000$ timing traces (t', e'). Likewise, we executed all four gates in the unmodified FHEW library. Once the traces are acquired, we checked whether the error value e' for each ciphertext lies in the correct bucket for its corresponding timing value t' or some other bucket. In short, we compare the error value with those present in individual buckets and find the error value which is closest to e'. Next, we also determine the bucket where this error value e' actually maps to, irrespective of its associated timing t'. This can be done by comparing e' with e_{min} and e_{max} for all buckets. If the error e' falls in the same bucket in both the cases, i.e, when compared w.r.t it's associated timing t' and also according to the range e_{min} and e_{max}, we consider that the template matching is successful, else it's an incorrect match.

Fig. 5 and 6 show the count of occurrence in the same bucket and in different buckets for all the ten and four homomorphic gates of TFHE and FHEW schemes respectively. The bucket size is 500 for TFHE and 23750000 for FHEW. In both cases, the key used to generate these ciphertexts is different and independent from the one used to generate ciphertexts for template data. In the figures, the blue bar represents the number of ciphertexts out of $1,000$ whose error lies in the correct bucket (correct matching), defined by the timing value of the corresponding gate operation. The orange bar represents the number of ciphertexts out of $1,000$ whose error lies in some other bucket (incorrect matching). It is worth mentioning that the keys used for template generation and template matching are different. However, the template attack, in our case, does not depend on the secret key, which is evident by the high range of correct matches with the template. We also highlight that there is a considerable difference between bucket sizes for the two libraries, as in case of TFHE, we disabled the bootstrapping step when performing the experiment, whereas the timing for FHEW included both the gate operation and bootstrapping.

To further validate that the templates work for any random key, we repeated the experiment for five different randomly generated keys and the results for ciphertexts lying in the correct bucket is shown in Fig. 7. The results are shown for AND gate for a bucket size of 500 in case of TFHE and 23750000 in case of FHEW. We observe that the count of ciphertexts matching in a correct bucket is similar in all five cases for TFHE, as shown by an almost straight line; while there is a slight variation in case of FHEW as shown by a zig-zag pattern. The reason for such results is that in case of TFHE we have injected the timing hooks deep into the library and thus are obtaining accurate timing values, while in case of FHEW we are obtaining the timing value of the entire gate operation including bootstrapping. These results show that a template created with a single key will work for any randomly generated keys as well, both in case of FHEW and TFHE.

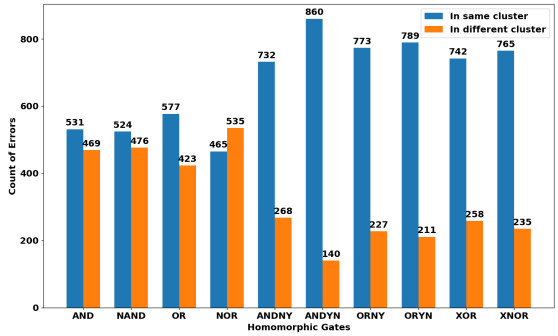

Fig. 5. Plot of gates with count of occurrence of error in same bucket vs different bucket. The size of buckets is 500. Key used to generate these ciphertexts is different from the one used to generate ciphertexts for template data.

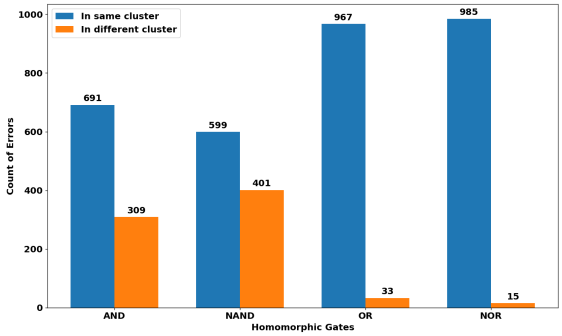

Fig. 6. Plot of gates with count of occurrence of error in same bucket vs different bucket. The size of buckets is 23750000. Key used to generate these ciphertexts is different from the one used to generate ciphertexts for template data.

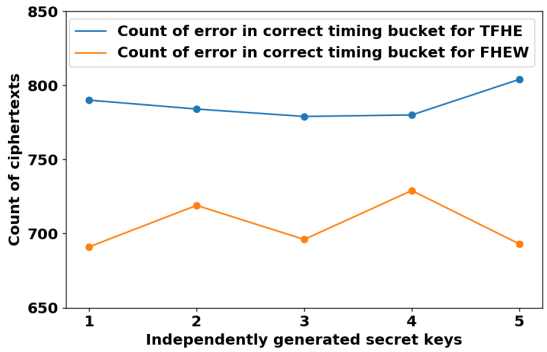

Fig. 7. 5 sets of ciphertexts with error in correct bucket. Each set is generated using different keys. The size of buckets is 500 and 23750000 resp.

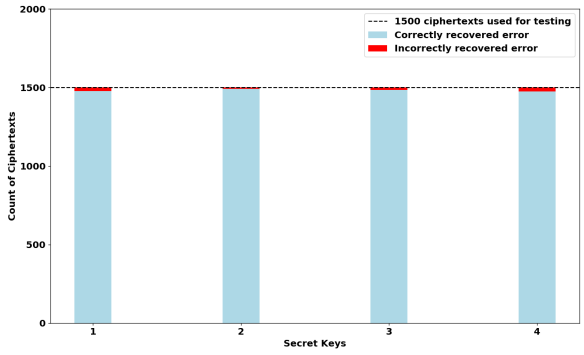

Fig. 8. Plot of count of ciphertexts with correctly and incorrectly recovered errors using HomAND gate. 1500 ciphertext pairs were generated to obtain this graph. Key used to generate these ciphertexts is different from the one used to generate ciphertexts for template data.

It also shows that generating one set of templates is enough for our attack to work, while template attacks usually require multiple templates with different keys.

Further to validate whether we can extract the error from the ciphertext, we generated $1,500$ ciphertext pairs using 4 different secret keys that are distinct from $\mathbf{s_t}$. We execute AND gate operation on these pairs and matched the results with their corresponding templates. Once the bounds were obtained, we performed the second part of our attack by adding perturbations and then sending queries to the client, in order to recover the error for each of these ciphertexts. Fig. 8 shows the plot of the count of ciphertexts with correctly and incorrectly recovered errors using homomorphic AND gate. The blue bar shows the count of ciphertexts for which we were able to successfully recover the correct error, while the red bar shows the count of ciphertexts where we could not recover the error.

We were successfully able to recover the error in most of the cases, as evident from Fig. 8 where all the blue bars being close to the black line that represents the count of ciphertexts used to perform this experiment. For the few ciphertexts for which we were not able to recover the error, the reason was that the timing value for them actually corresponds to the buckets that had either few error values or no error values at all. For these ciphertexts, we can either recompute this gate to get better timing values or ignore this particular ciphertext altogether. We used four different, independent keys to generate four different sets of ciphertexts to ensure that our attack works with any unknown key. We observe that the plot is similar for all the four keys, implying that this method works with ciphertexts generated using any key. Finally, to recover the error from a single ciphertext, we require two queries to the client to recover the sign of the error and the underlying plaintext message and then 20 to 21 queries (in case of TFHE without bootstrapping), 25 to 26 queries in case of TFHE with bootstrapping and 3 to 5 queries in case of FHEW to recover the actual error value. The reason for requiring this many queries is that the difference between the maximum and minimum errors in most of the timing buckets is between 2^{20} to 2^{21} (in case of TFHE without bootstrapping). Since we are using Binary Search to reduce the range of errors, the number of queries required is $\log_2 2^{20} = 20$ or $\log_2 2^{21} = 21$. Thus we can recover the error with only 22 or 23 queries in case of TFHE without bootstrapping, 27 or 28 queries in case of TFHE with bootstrapping and 5 to 7 queries in case of FHEW to the CVO. Once we recover the original error from ciphertexts,

we form a system of equations and then solved the same to recover the entire key. We perform full key recovery for TFHE with key size 630 bits on a Desktop computer running Intel i7-7567U @ 3.5GHz, powered by Ubuntu 18.04. In case of TFHE without bootstrapping, we perturbed 5000 ciphertexts and out of those, 1260 ciphertext were suitably faulted to recover their error values. In case of TFHE with bootstrapping, we perturbed 2000 ciphertexts and out of those, 703 ciphertext were suitably faulted to recover their error values. In case of FHEW, we perturbed 3000 ciphertexts and out of those, 1003 ciphertext were suitably faulted to recover their error values. Finally, we ran Gaussian Elimination from SageMath9.0 and Python 3.8 to successfully recover the entire secret key. The overall attack process, from template matching to key recovery took around 6, 8 and 2 hours and required 31948, 21273 and 9073 CVO queries, in case of TFHE without bootstrapping, TFHE with bootstrapping and FHEW respectively.

IX. Discussion and Conclusions

In this paper, we have shown that timing attack in combination with CVO access 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 7, 23 and 28 such queries for the libraries FHEW and TFHE (without and with bootstrapping), respectively. However, even with a constant number of queries to the oracle, a few critical questions can arise when one considers the practical implication of the attacks and the role of client in aiding the attack.

why would the client decrypt a re-modified ciphertext in the first place? Also, why would it react to a wrong decryption when it already has obtained a correct decryption with a previous modified version?: 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 the 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. This is not a practical solution as the client will have to store the results of all the previous computations and will have to check whether the new ciphertext has been received before or not, which requires both storage and processing. It also needs to be kept in mind that this final ciphertext is a result of a chain of operations where different input as well as intermediate ciphertexts contribute to these operations. So there is still a chance that the first part of final ciphertext evaluates out to be $\mathbf{a}$. The server also may not resend the modified ciphertext immediately, as it is free to replace a single ciphertext from the output of some later computation with this modified ciphertext. Again the user will not be able to tell the difference between a fresh ciphertext and a replayed one.

what if the user does not react immediately and decides to ask for a recomputation at a later time?: To answer this question, we would like to state that the user cannot ask us to recompute the value of a single bit and the whole function would need to be recomputed to obtain the result. If the user do decide to ask for a re-computation at a later stage, it will have to resend the input values to the server. The server has the necessary storage capacity to store the previous inputs along with information regarding the manipulated ciphertext from the result. It will store them anyway so that it can decrypt the same once it recovers the secret key. Once the server receives the input all it needs to do is to look up in the table to see whether it has received these inputs previously or not. To prevent this attack, the user may choose to encrypt the inputs again so that it receives a new set of ciphertexts. While it is a possible counter-measure, it is certainly not cheap as it requires multiple encryptions and transmission of the same inputs.

what happens if the server delays its query and re-sends the modified ciphertext as part of the result of some later computation and gets no reaction for 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 the modified ciphertext c is an encryption of 1, i.e., if decrypted correctly, it will give output as 1 otherwise 0. The server modifies c, without knowing whether it will decrypt correctly or not, and replaces a ciphertext c' from 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 ciphertext did not decrypt correctly. 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 in the final level of the circuit that has a high probability to output encryption of a certain bit irrespective of the input values. For example, say a gate outputs 1 with high probability irrespective of the input ciphertexts (e.g., NAND, OR), then the server can always send an encryption of 1 as the output of this gate.

Potential Countermeasures

The authors in [8] utilized the IND-CVA security model to attack the input data and underlying homomorphic function in a similar setting and proposed four possible countermeasures. We will revisit their last two countermeasures to analyse the relevance in context to our proposed attack. The first two have been refuted by themselves, so we are not considering them.

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. While they have shown how this can be achieved, 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 they have themselves refuted this countermeasure as that 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 in the first place.

Single-use database: The other solution proposed in [8] is for a single-use database, where the client will have to re-encrypt and resend the database to the cloud after each query. This also prevents the user from frequently changing the underlying secret key used to encrypt this database stored on the server after each decryption failure. Moreover, this solution is not practical in scenarios where the server is used both for processing and storage of data for future use.

Random computation and single-bit ciphertext to encrypt multi-bit plaintext: Random computations will also require random data to be generated and sent to the cloud which is certainly not cheap. This becomes more expensive when the client pays to the cloud per computation, which the authors themselves have acknowledged.

Countermeasure with reaction restriction: It is to highlight that the number of ciphertexts required to make the attack successful is in the order of key size k. Server can obtain that amount of ciphertexts from the reactions or recomputation requests from the client side. Hence, one possible countermeasure is to restrict the number of requests for recomputation, which should be less than k. However, there are two limitations of this solution. First, our proposed attack does not demand consecutive reactions or recomputation 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 recomputation requests can be imposed. Second, as explained in the plaintext recovery step of the attack, the cloud can forcefully perturb the ciphertext to 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 [28] concept can be promising against this attack. In this case, the secret key is divided into N shares and distributed among N users 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. Detailed implementation of this countermeasure will be taken as a future work.

REFERENCES

[1] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, (2001), Advanced Encryption Standard (AES), Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD.

[2] 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, pp. 84–93, STOC '05, Association for Computing Machinery, New York, NY, USA (2005).

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

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

[5] I. Chillotti, N. Gama, M. Georgieva, M. Izabachène, "TFHE: Fast fully homomorphic encryption library", (August 2016), https://tfhe.github.io/tfhe/, accessed March 2022.

[6] F. Aydin, E. Karabulut, S. Potluri, E. Alkim, 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, pp. 1527–1532, DATE '22, European Design and Automation Association, Leuven, BEL (2022).

[7] S. Chari et al, "Template attacks," in CHES, 2002, pp. 13–28.

[8] I. Chillotti, N. Gama, L. Goubin, "Attacking fhe-based applications by software fault injections", Cryptology ePrint Archive, Paper 2016/1164 (2016).

[9] J. Loftus, A. May, N. P. Smart, and F. Vercauteren, On cca-secure somewhat homomorphic encryption, in Selected Areas in Cryptography, pages 55–72, Springer, 2012.

[10] Z. Hu, F. Sun, and J. Jiang, "Ciphertext verification security of symmetric encryption schemes", Science in China Series F: Information Sciences, 52(9):1617–1631, 2009.

[11] Z. Zhang, T. Plantard, and W. Susilo, "Reaction attack on outsourced computing with fully homomorphic encryption schemes", in International Conference on Information Security and Cryptology, pages 419–436, Springer, 2011.

[12] K. Sinha, P. Majumder and S. K. Ghosh, "Fully Homomorphic Encryption based Privacy-Preserving Data Acquisition and Computation for Contact Tracing," 2020 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS), 2020, pp. 1-6.

[13] R. Lindner, C. Peikert, (2011), "Better Key Sizes (and Attacks) for LWE-Based Encryption", in A. Kiayias (eds) Topics in Cryptology – CT-RSA 2011, CT-RSA 2011, Lecture Notes in Computer Science, vol 6558, Springer, Berlin, Heidelberg.

[14] S. Bai, S.D. Galbraith (2014), "Lattice Decoding Attacks on Binary LWE", in W. Susilo, Y. Mu (eds) Information Security and Privacy, ACISP 2014, Lecture Notes in Computer Science, vol 8544, Springer, Cham.

[15] M.R. Albrecht, (2017), "On Dual Lattice Attacks Against Small-Secret LWE and Parameter Choices in HElib and SEAL", in JS. Coron, J. Nielsen (eds) Advances in Cryptology – EUROCRYPT 2017, EUROCRYPT 2017, Lecture Notes in Computer Science(), vol 10211, Springer, Cham.

[16] T. Espitau, A. Joux, N. Kharchenko, (2020), "On a Dual/Hybrid Approach to Small Secret LWE", in K. Bhargavan, E. Oswald, M. Prabhakaran, (eds) Progress in Cryptology – INDOCRYPT 2020, INDOCRYPT 2020, Lecture Notes in Computer Science(), vol 12578, Springer, Cham.

[17] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryptionover the Torus", in Journal of Cryptology, volume 33, pages 34–91 (2020).

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

[19] R. L. Rivest, A. Shamir, and L. Adleman, 1978, "A method for obtaining digital signatures and public-key cryptosystems", Commun. ACM 21, 2 (Feb. 1978), pp. 120–126.

[20] Y. Abbar, P. Aubry, T. Barry, S. Carpov, S. Mallick, M. Krichen, D. Ligier, S. Shpak, and R. Sirdey, (2021), "Cloud-based Private Querying of Databases by Means of Homomorphic Encryption", in Proceedings of the 6th International Conference on Internet of Things, Big Data and Security - IoTBDS, ISBN 978-989-758-504-3, ISSN 2184-4976, pp. 123-131.

[21] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing", in 2018 IEEE Symposium on Security and Privacy, pp. 962–979, IEEE Computer Society Press, May 2018.

[22] M. Kim and K. Lauter, "Private Genome Analysis Through Homomorphic Encryption", BMC medical informatics and decision making, 15, December 2015.

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

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

[25] M. Fahr. Jr., H. Kippen, A. Kwong, T. Dang, J. Lichtinger, D. Dachman-Soled, D. Genkin, A. Nelson, R. Perlner, A. Yerukhimovich, D. Apon, "When Frodo flips: End-to-end key recovery on FrodoKEM via Rowhammer", ACM CCS 2022, 2022.

[26] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, D. Stebila, "Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE", ACM CCS 2016, 2016.

[27] P.W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer", SIAM Review 41(2), pp. 303–332, 1999.

[28] D. Boneh, R. Gennaro, S. Goldfeder, A. Jain, S. Kim, P.M.R. Rasmussen, A. Sahai, "Threshold cryptosystems from threshold fully homomorphic encryption", in H. Shacham, A. Boldyreva (eds.) Advances in Cryptology – CRYPTO 2018, pp. 565–596. Springer International Publishing, Cham (2018).

[29] A. D. Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely", in Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing (STOC '94), Association for Computing Machinery, New York, NY, USA, pp. 522–533.

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

[31] J. Fan, F. Vercauteren, "Somewhat practical fully homomorphic encryption", IACR Cryptol. ePrint Arch. p. 144 (2012).

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

[33] J.H. Cheon, A. Kim, M. Kim, Y.S. Song, "Homomorphic encryption for arithmetic of approximate numbers", in T. Takagi, T. Peyrin, (eds.) 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. Lecture Notes in Computer Science, vol. 10624, pp. 409–437. Springer (2017).

[34] HElib, https://homenc.github.io/HElib (Aug 2022).

[35] PALISADE, https://gitlab.com/palisade/palisade-release (Aug 2022).

[36] https://github.com/lducas/FHEW, accessed August 2022.

[37] L. Ducas, D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second", in E. Oswald, M. Fischlin, (eds) Advances in Cryptology – EUROCRYPT 2015, Lecture Notes in Computer Science(), vol 9056, Springer, Berlin, Heidelberg.

[38] R.L. Rivest, L. Adleman, M.L. Deaouzos, "On Data Banks and Privacy Homomorphism", in R.A. DeMillo, (eds) Foundations of Secure Computation, Academic Press, New York, 169-179.

[39] P.C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", in N. Koblitz, (eds) Advances in Cryptology — CRYPTO '96, Lecture Notes in Computer Science, vol 1109, Springer, Berlin, Heidelberg.

[40] C. Ashokkumar, R. P. Giri, B. Menezes, "Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks", IEEE European Symposium on Security and Privacy (Euro S&P), pp. 261-275, 2016.

[41] Y. Yarom, K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack", in Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14), USENIX Association, USA, pp. 719–732.

[42] B. Timon, "Non-Profiled Deep Learning-based Side-Channel attacks with Sensitivity Analysis". IACR Transactions on Cryptographic Hardware and Embedded Systems, 2019(2), pp. 107–131.

[43] A. Golder, D. Das, J. Danial, S. Ghosh, S. Sen, A. Raychowdhury, "Practical Approaches Toward Deep-Learning-Based Cross-Device Power Side-Channel Attack", in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 12, pp. 2720-2733, 2019.

[44] L. Bi, X. Lu, J. Luo, K. Wang, Z. Zhang, "Hybrid dual attack on LWE with arbitrary secrets", Cybersecurity 5, 15 (2022).

[45] Q. Guo, 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, Springer-Verlag, Berlin, Heidelberg, pp. 33–62.

[46] S. Bai, S. Miller, W. Wen, "A refined analysis of the cost for solving LWE via uSVP", Africacrypt 2019 - 11th International Conference on Cryptology in Africa, Jul 2019, Rabat, Morocco.

[47] K. Laine, K. Lauter, "Key recovery for lwe in polynomial time", IACR Cryptology ePrint Archive (October 2015).

[48] S. Arora and R. Ge, "New algorithms for learning in presence of errors", In Proceedings of the 38th international colloquim conference on Automata, languages and programming - Volume Part I (ICALP'11), Springer-Verlag, Berlin, Heidelberg, pp. 403–415.

[49] M. R. Albrecht, C. Cid, J.-C. Faugère, R. Fitzpatrick, L. Perret, "On the complexity of the BKW algorithm on LWE", Des. Codes Cryptography 74, 2 (February 2015), pp. 325–354.

[50] M. Chenal and Q. Tang, "On key recovery attacks against existing somewhat homomorphic encryption schemes", in Progress in Cryptology-LATINCRYPT 2014, pp. 239–258, Springer, 2014.

[51] E. L. Bassham III, "The advanced encryption standard algorithm validation suite (AESAVS)." NIST Information Technology Laboratory (2002).