Survey: Attacks on Fully Homomorphic Cryptography

Pascal Perrenoud

June 2025

Teacher : Alexandre Duc





Abstract

We show how attacks against FHE schemes, shown by Li and Micciancio in 2021 at Eurocrypt, is a real threat against FHE usages in real use-cases. The countermeasures proposed to mitigate it work for passive attackers but add an important performance overhead and can fail in practice.

Through two real example of FHE usage, we show how an active attacker can still attack FHE schemes and perform a full key recovery. The countermeasures are tailored for the new IND-CPA-D security notion that takes into account passive attackers, which does not fully represents reality.

We also highlight how circuit privacy becomes an issue, as countermeasures rely on the definition of the circuit to effectively protect FHE schemes. The exact definition of this circuit is never considered in countermeasures and often summarized as *a function f that outputs a correct ciphertext*.

Contents

I Introduction **2**

II Preliminaries **3**

1 FHE **3**

2 RLWE **3**

3 CKKS **4**

4 IND-CPA security notion **5**

III On the Security of Homomorphic Encryption on Approximate Numbers **6**

5 The attack **6**

6 IND-CPA-D **7**

7 (q, l)-IND-CPA-D **7**

8 Results reproduction **8**

IV Countermeasures **9**

9 Noise flooding **9**

10 Output sanitization **10**

11 Circuit privacy and security **10**

V Practical impacts **11**

12 GWAS and databank **11**

13 Confidential computing in the cloud **15**

14 Discussion **17**

VI Alternatives **19**

15 MPC **19**

16 ZKP **19**

VII Conclusion **20**

VIII Appendices **C**

17 Attack source code **C**

List of Figures

1 Sequence diagram of the GWAS scenario 11
2 Sequence diagram of the confidential cloud computing scenario 15
3 Confidential cloud computing with noise flooding 16

List of Tables

1 Summary of the two scenarios, the impact of the attack and of the countermeasures 18

List of Algorithms

1 IND-CPA$^A(\lambda)$ 5
2 IND-CPA-D : $\mathbf{Encryption}_{pk}(m_0, m_1)$ 7
3 IND-CPA-D : $\mathbf{Evaluation}_{evk}(g, J)$ 7
4 IND-CPA-D : $\mathbf{Decryption}_{sk}(j)$ 7

Part I
Introduction

Fully Homomorphic Encryption (FHE) is a field of cryptography that allows to evaluate mathematical operations on encrypted data, in an homomorphic way. It is, in theory, a powerful tool as it allows to tackle a wide range of security problems in confidential computing.

State-of-the-art FHE schemes rely on the RLWE problem. This works by adding a *noise* to protect the confidentiality. In exact FHE schemes, this noise is not leaked after decryption, as correctness is ensured, but for approximate FHE schemes, like CKKS, this noise is leaked and ends up in the decrypted plaintext. If this has no consequence on correctness, we will see how this affects security.

Li and Micciancio showed how to exploit a ciphertext and its decryption to perform a complete key recovery in expected polynomial time. By using linear algebra, they show how to combine a ciphertext and its decryption to retrieve the secret key that was used to encrypt the data. After that, Cherci et al. extended the attack in 2024 to exact FHE schemes. Using thousands of ciphertexts and decryption, and lattice reduction technique, they managed to attack exact FHE schemes in the same way, recovering the secret key.

As shown by Li and Micciancio, IND-CPA security notion makes an important assumption : the decryption is exact, which does not hold for *approximate* FHE schemes nor for exact FHE schemes when pushed over their limit. To take this into account, they proposed a new security notion, called IND-CPA-D, which is an extension and strictly stronger version of IND-CPA. In this security notion, they add decryption and evaluation oracles to better model the real behavior of FHE schemes.

After the attack was disclosed, various countermeasures were proposed. They work with certain assumptions and add a performance overhead to already computationally heavy FHE schemes. Those mitigations consider the noise as confidential, and protect it by either trimming it away or flooding it in more noise.

Our contribution is to confront those countermeasures to real use-cases through two examples, evaluating their security and performance overhead. We will see how they can fall short in practice if the attacker is considered active.

The rest of the paper is organized as follows. In Section 2, we provide some background about FHE, CKKS and IND-CPA. In Section 3, we explore the results of Li and Micciancio, and try to reproduce their result. In Section 4, we explore two countermeasures proposed to mitigate the attack. In Section 5, we analyze the practical security and impact of those countermeasures through two real use-cases. In Section 6, we highlight some alternatives that could extend or replace FHE for those scenarios. Section 7 concludes the article with remarks about why the countermeasures fall short in practice and how we could mitigate the attack against active attackers.

Part II
Preliminaries

1 FHE

A *Fully Homomorphic Encryption* (FHE) [RAD+78] scheme is a *public-key encryption* scheme with *homomorphic evaluation* capabilities. An FHE scheme, with a plaintext space $\mathcal{M}$, a ciphertext space $\mathcal{C}$, is defined by four algorithms:

- **KeyGen** : With a security parameter k, generates a private key sk, a public key pk and an evaluation key evk
- **Enc** : $\mathcal{M} \to \mathcal{C}$, encryption of a message, using the public key
- **Dec** : $\mathcal{C} \to \mathcal{M}$, decryption of a ciphertext, using a private key
- **Eval** : $\mathcal{C}^* \to \mathcal{C}$, homomorphic evaluation of a circuit, using an evaluation key and a circuit g, where C^* is a set of ciphertexts

The scheme is homomorphic as computing $g : \mathcal{C}^* \to \mathcal{C}$ then decrypting the resulting ciphertext is the same as computing its homomorphic equivalent $g : \mathcal{M}^* \to \mathcal{M}$. Another definition is :

$$\mathrm{Dec}_{sk}(\mathrm{Eval}_{evk}(g, \mathrm{Enc}_{pk}(m))) \approx g(m) \tag{1}$$

2 RLWE

The current main FHE schemes are based on the RLWE [LPR12] problem, an adaptation of the LWE [Reg10] problem over polynomial rings. It works on a polynomial ring $\mathcal{R} = \mathbb{Z}_p[X]/(X^N+1)$, having its coefficients in a field $\mathbb{Z}_p$ and the polynomials reduced by the modulus $2^N + 1$, where $N = 2^n$, which makes it a cyclotomic ring.

The RLWE problem has the following construction :

- A random but known polynomial $a \in \mathcal{R}$
- A random and unknown gaussian noise $e \in \mathcal{R}$
- A small, relative to a bound b, and secret polynomial $s \in \mathcal{R}$
- The pair $(as + e, a)$

In its decision version, the RLWE problem is to distinguish, in a list of polynomials and with a non-negligible advantage, a pair of the form $(as + e, a)$ against random pairs in $\mathcal{R}^2$.

3 CKKS

The FHE scheme CKKS [CKKS16] is the only *approximate* scheme, as $\text{Dec}_{sk}(\text{Enc}_{pk}(m)) = m+e \approx m$. It is based on the hardness of the RLWE [LPR12] problem. It is the only FHE scheme that encrypts and performs homomorphic operations on real number instead of integers.

3.1 Encoding

As CKKS is based on the RLWE problem, it works with a polynomial ring. The plaintexts that can be encrypted with CKKS are polynomials, and the ciphertexts are polynomials. The messages that can be encrypted are vectors of complex numbers with a maximum size of $\frac{N}{2}$. Therefore, we need to encode those messages before encrypting them, and to decode them after decryption. This encoding is complex, and composed of multiple steps, using projections, expansions and isomorphisms between complex vectors and polynomials over a finite field. An important step of the encoding is the multiplication with a *scale factor* that ensures correctness.

3.2 Key generation

The secret key used in CKKS is a small polynomial s. The public key is $(-as + e, a)$, with a a random polynomial and an *noise* e, a random polynomial with a gaussian distribution. Several evaluation keys are also derived from the secret key and can be shared with the public key.

3.3 Encryption

To encrypt a polynomial m, we sample a random polynomial v and two noises e_0, e_1, we then compute c follows.

$$\begin{aligned} c &= v \cdot pk + (m + e_0, e_1) \\ &= (v(-as + e), va) + (m + e_0, e_1) \\ &= (-vas + ve, va) + (m + e_0, e_1) \\ &= (m - vas + ve + e_0, va + e_1) \\ &= (c_0, c_1) \end{aligned} \tag{2}$$

To decrypt a ciphertext, we output $m' = \text{Dec}_{sk}(c) = sc_1 + c_0 = m + e' \approx m$, as we consider e, e_0 and e_1 negligible.

$$\begin{aligned} m' &= c_0 + sc_1 \\ &= (m - vas + ve + e_0) + s(va + e_1) \\ &= m - vas + ve + e_0 + vas + se_1 \\ &= m + ve + e_0 + se_1 \\ &\approx m \end{aligned} \tag{3}$$

3.4 Addition

To add two ciphertexts C_0 and C_1, we simply add them component-wise :

$$\begin{aligned} d &= c + c' \\ &= (c_0, c_1) + (c'_0, c'_1) \\ &= (m_0 - a_0 s + e_0, a_0) + (m_1 - a_1 s + e_1, a_1) \\ &= ((m_0 - a_0 s + e_0) + (m_1 - a_1 s + e_1), a_0 + a_1) \\ &= (m_0 + m_1 - s(a_0 + a_1) + e_0 + e_1, a_0 + a_1) \end{aligned} \tag{4}$$

The decryption function is the same as seen before and will output $m_0+m_1+e_0+e_1 \approx m_0+m_1$. We can see here that the noise of the resulting plaintext is the addition of the noise of the two plaintexts.

3.5 Multiplication

The multiplication of two ciphertexts requires an evaluation key. The multiplication of $c = (c_o, c_1)$ and $d = (d_0, d_1)$ is $cd = (c_0d_0, c_0d_1 + d_0c_1, c_1d_1)$. As the decryption takes a ciphertext with two components as an input, we need to *relinearize* this result, using the evaluation key. This advanced technique is based on key-switching [KPZ21]. This technique will consume what is called a *level* of multiplication. The number of levels is defined when generating the keys.

3.6 Bootstrapping

It is important to note that noise builds up as homomorphic operations progress. It accumulates until it covers the underlying, encoded message. As we saw with multiplication, relinearization also consumes a level of the multiplication depth, until there is no level left. This is where the most important operation of FHE schemes comes to help : bootstrapping. Without this function, CKKS would be a *somewhat homomorphic encryption* scheme, as its multiplicative depth would be limited. Bootstrapping is another advanced technique, and the slowest function of all the functions in FHE schemes, but is central as it transforms SHE schemes in FHE schemes.

4 IND-CPA security notion

When it comes to encryption, a scheme can be tested against a security notion called *indistinguishable under chosen plaintext attack* or IND-CPA [BR01]. It consists of a single decryption oracle that the attacker can use. After we generate a fresh keypair with the security parameter λ, the attacker can first perform calls to the encryption oracle, before it returns two different plaintexts : m_0 and m_1. A random bit is drawn uniformly at random, the corresponding plaintext is encrypted and the ciphertext is given to the attacker that must return which bit was picked. If the attacker has a non-negligible chance to distinguish which plaintext was encrypted, it breaks the IND-CPA security notion.

Algorithm 1 IND-CPA$^A(\lambda)$

$(sk, pk) \leftarrow \mathbf{Gen}(1^\lambda)$
$(m_0, m_1) = \mathbf{A}(\lambda, pk)$
$b \leftarrow\!\!\$ \{0, 1\}$
$c \leftarrow \mathbf{Enc}(m_b, pk)$
$b' \leftarrow \mathbf{A}(c)$
Returns $(b == b')$

Part III

On the Security of Homomorphic Encryption on Approximate Numbers

In 2021, Li and Micciancio [LM20] presented a passive attack against CKKS [CKKS16] at Eurocrypt. They highlighted the weakness of the IND-CPA security game when it comes to encryption scheme with *approximate* decryption. Their attack is efficient, running in expected polynomial time, and very practical as it leads to complete key recovery.

As stated by Li and Micciancio, IND-CPA does not capture an important property of CKKS : the decryption is approximate. When we challenge AES against IND-CPA, we consider that $\text{Dec}_k(\text{Enc}_k(x)) = x$, but this does not hold with CKKS. In the case of approximate FHE schemes, we rather have $\text{Dec}_{sk}(\text{Enc}_{pk}(m)) = m + e \approx m$, with e a small gaussian that we *thought* was negligible. This approximation is considered negligible for the correctness of the computation, but the security implications of this was not considered. The leak of this noise is what leads to the attack. This noise was not considered confidential but now it must be.

5 The attack

Using only linear algebra, Li and Micciancio showed that with only a ciphertext and its decryption, or a pair (c, m'), an attacker can recover the secret key as follows :

$$\begin{aligned} s' &= \frac{m' - c_0}{c_1} \\ &= \frac{(m+e) - (m - as + e)}{a} \\ &= \frac{m + e - m + as - e}{a} \\ &= \frac{as}{a} \\ &= s \end{aligned} \tag{5}$$

This simple equation is the attack that breaks CKKS. It was later extended, by Checri et al. [CSBB24] to all FHE schemes, even the exact ones. If on CKKS, a *correct* ciphertext is enough to recover the secret key, the attack on exact FHE scheme is a bit harder, as it requires *wrong* ciphertext, in the sense of correctness, as we have to accumulate a lot of noise through homomorphic evaluations, until the correctness of the schemes does not hold anymore. Along with this, an attack on an exact FHE scheme also requires thousands of queries to the decryption oracle, and to use lattice reduction techniques to retrieve the secret key. Even if the attack is a bit more complex, it is still very practical.

6 IND-CPA-D

In the IND-CPA security notion, the attacker has only access to an encryption oracle. To take into account approximate decryption, Li and Micciancio proposed the new security notion *indistinguishability under chosen plaintext attacks with decryption oracles* (IND-CPA-D). This gives to the attacker an access to an evaluation oracle and to a restricted decryption oracle. This security notion is extending IND-CPA to a strictly stronger version. For encryption schemes with *exact* decryption, the scheme is IND-CPA secure *iff* it is IND-CPA-D secure.

6.1 IND-CPA-D game

The three oracles of the IND-CPA-D game share a common state $S \in (M \times M \times C)^*$, with M the message space and C the ciphertext space. This definition restricts attacks to passive attacks, as the attacker cannot decrypt any abritrary ciphertext.

Algorithm 2 IND-CPA-D : $\mathbf{Encryption}_{pk}(m_0, m_1)$

$c \leftarrow \mathbf{Enc}_{pk}(m_b)$
$S \leftarrow [S, (m_0, m_1, c)]$
Returns c

Algorithm 3 IND-CPA-D : $\mathbf{Evaluation}_{evk}(g, J)$

$c \leftarrow \mathbf{Eval}_{evk}(g, S[J_0].c, S[J_1].c, \ldots, S[J_{n-1}].c)$
$S \leftarrow [S, (g(S[J_0].m_0, S[J_1].m_0, \ldots, S[J_{n-1}].m_0), g(S[J_0].m_1, S[J_1].m_1, \ldots, S[J_{n-1}].m_1), c)]$
Returns c

Algorithm 4 IND-CPA-D : $\mathbf{Decryption}_{sk}(j)$

Ensure: $j < |S|$, $S[j].m_0 = S[j].m_1$
Returns $\mathbf{Dec}_{sk}(S[j].c)$

7 (q, l)-IND-CPA-D

The new security notion IND-CPA-D does not totally reflect reality and might be too restrictive as attacks against exact FHE schemes require thousands of queries to evaluation and decryption oracles, while real use-cases of FHE often only require a few calls to decryption. To fill this gap, Li and Micciancio also proposed an intermediate security notion called (q, l)-IND-CPA-D. In this version, the attacker is limited to $q(k)$ decryption queries and $l(k)$ evaluation and encryption queries, with k the security parameter. This security notion will allow to describe defenses where the secret key owner can restrict the number of decryption performed. This leads to more efficient countermeasures.

8 Results reproduction

To show the practicality of their attack, Li and Micciancio implemented it using multiple well-known programming libraries that correctly implement CKKS. They released the source code[1] of their attack. Before disclosing, they contacted the developers of those libraries who quickly responded to the attack and applied fixes, ranging from warning the user to implementing real countermeasures.

In an attempt to reproduce their attack, we focused on the library `Palisade`, now called `OpenFHE` [ABBB+22]. Unfortunately for us, and fortunately for the FHE world, the attack as it was presented does not work anymore. It is easily explained as all the main FHE libraries reacted quickly to their disclosure and worked on implementing various countermeasures[2]. The library `OpenFHE` implemented noise flooding[3], among other defenses.

Even if the API of this library offers to tweak the countermeasures, we could not reproduce the result of Li and Micciancio using a recent version of `OpenFHE`. Appendix 17 shows the modified code of the attack, adapted to the new version and with defenses tweaked. The comments through the code show what was modified compared to the original version.

[1] `https://github.com/ucsd-crypto/CKKSKeyRecovery`

[2] `https://openfhe.org/security-of-ckks/`

[3] `https://github.com/openfheorg/openfhe-development/blob/main/src/pke/examples/CKKS_NOISE_FLOODING.md`

Part IV
Countermeasures

After the release of the publications about the attacks on first approximate then exact FHE, two main countermeasures were proposed : *noise flooding* [LMSS22] and *output sanitization* [BCM+24].

Under certain assumptions, they provide IND-CPA-D or (q, l)-IND-CPA-D security for FHE schemes. The following sections show how the defenses work, the assumptions they do, the precision loss they introduce and the performance overhead they add.

9 Noise flooding

A first countermeasure, called *noise flooding* [LMSS22], was proposed in 2022 to counter the attacks against FHE. Using *differential privacy*, it consists of adding a gaussian noise, where the parameters of its distribution are computed as the *worst-case noise bound* of the circuit g evaluated on the data. With noise flooding applied, the work of Li et al. allows to reduce the security analysis of CKKS to the case of exact FHE, where IND-CPA and IND-CPA-D are equivalent.

To compute the noise bound, this construction requires to run the circuit g on the input data, or a representative subset of the data, with a fresh keypair. This means the circuit g has to run twice : once to estimate the bound and once to run the actual computations on the data. This is an important performance overhead, as homomorphic circuits are quite slow and we now have to run them twice.

Even if it achieves IND-CPA-D security, *noise flooding* suppose something important : the decryption oracle can know exactly what circuit g was applied. As shown in the article, the bounds computed on the worst-case are nearly tight, meaning that any reduction in the parameters of the Gaussian distribution would be enough to attack the construction.

9.1 Worst-case noise bound estimation

Using *differential privacy* [MTZ19], the idea is to process the decrypted result by adding additional noise to it in a way to limit the leakage of the original noise that was in the ciphertext. This noise is randomly drawn from a gaussian distribution. To ensure the security of the scheme, the parameters of this distribution has to be computed on the worst-case bound of the circuit applied. This estimation is done by running the circuit on the dataset, but encrypted with a fresh keypair. With every homomorphic operation, we *tag* the resulting ciphertext with a noise estimation, based on the previous tag, the circuit and the tag of the ciphertexts involved by the circuit. At the end of the computation, we end up with a worst-case estimation of the noise generated by the circuit *and* the data. We now can use this bound to post-process the decryption of ciphertexts.

10 Output sanitization

Similar to noise flooding, *output sanitization* [BCM+24] is a countermeasure to protect from the latest attack against FHE. In a similar fashion, it consists of computing a bound on the noise generated by a circuit g. Instead of adding a random Gaussian noise on decryption, they propose to trim the r less significant bits of the decryption, which creates an exact version of CKKS, called (δ, r)-CKKS, with probability δ, for a maximum of q calls to the decryption and evaluation oracles. The construction is (q, l)-IND-CPA-D secure.

Instead of running the whole circuit g with a fresh keypair, as for noise flooding, they proposed a way to compute the noise bound based on the homomorphic operations composing the circuit. This lowers the performance overhead but still implies that the decryption oracle needs to know the exact circuit g that was applied on the data.

Once we have this noise bound estimation, we can again post-process the resulting decryption, but this time by trimming the r less significant bits of the plaintext. This allows to reduce leakage of the noise that was in the ciphertext and thus make (δ, r)-CKKS (q, l)-IND-CPA-D secure.

10.1 Composable average-case noise bound

The approach of this defense when it comes to noise bound estimation differs from noise flooding. Instead of computing the noise bound on the data, they propose formulas to estimate the noise of every homomorphic operation. Having a circuit g, we can compose those formulas to obtain the average-case noise bound of the circuit. From this bound, we can compute the number r of least significant bits to trim to obtain a (δ, r)-exact CKKS scheme that is (q, l)-IND-CPA-D secure ,with probability δ.

11 Circuit privacy and security

As seen in the two previous Sections, current countermeasures require to compute a noise bound generated by the circuit. In practice, this poses two issues : (i) if the owner of the secret key is not the party that evaluates homomorphic operations, we do not yet have any way to cryptographically prove what the circuit was, and (ii) the circuit computed by another party cannot be private, as they have to share it with the secret key owner to apply decryption. We will see in a later Part that this poses serious issues when applying those countermeasures in real use-cases.

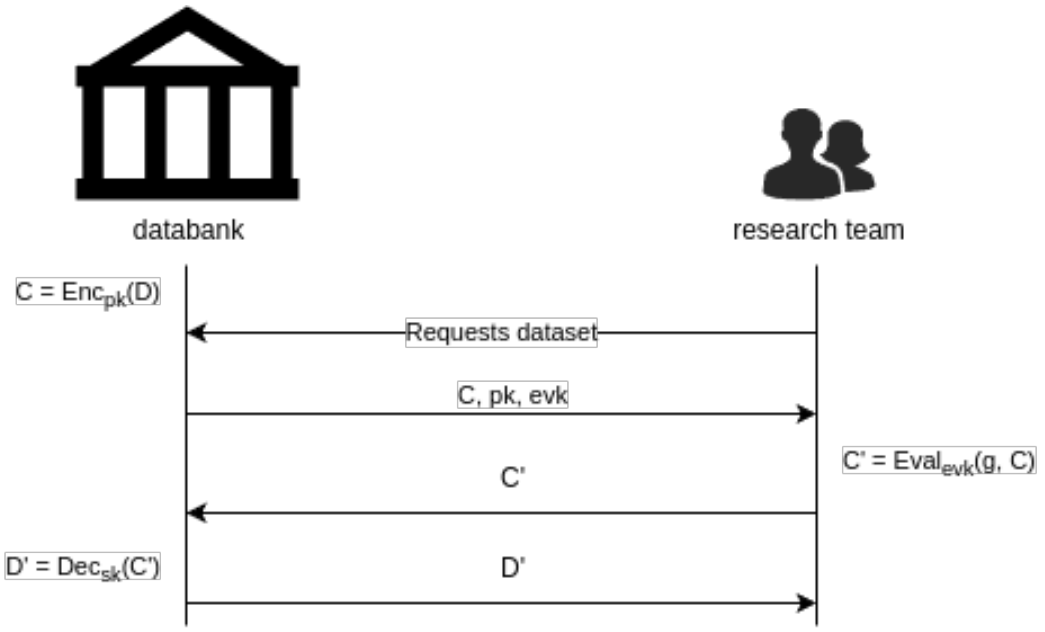


Figure 1: Sequence diagram of the GWAS scenario

Part V
Practical impacts

To understand the impact of this attack in real world cryptography, we will consider two scenarios of confidential computing using FHE. We will then analyze if the attack applies to those scenarios, then we will analyze the impact of the two countermeasures seen in Part IV. To understand the impact, we will consider the loss of confidentiality (key recovery, full or partial data recovery) and the performance overhead it adds.

12 GWAS and databank

A common usage of FHE is in *Genome-wide association studies* (GWAS). Using a large dataset of DNA samples with a control group of healthy patients and a test group having a disease, research teams look for statistical differences between them to possibly identify one or multiple variations in the DNA, called Single Nucleotide Polymorphism (SNP), that could be related to the disease.

DNA is considered to be the source code of Humans. It contains information about someone's height, weight or predisposition to certain diseases. Even anonymized, with the DNA of someone, we can identify them. Those data are therefore inherently highly confidential.

Databanks are organisms collecting a large quantity of DNA samples and providing them to research teams on request for their studies. As the dataset is highly confidential, the process to obtain it is complicated. The databank pre-processes the dataset and make sure to anonymize it as much as possible. As an alternative, with FHE, as the dataset is encrypted, the requirements on the protection of the dataset are lower and databank are more prone to share those dataset for studies.

12.1 Scenario

Let's consider the following scenario:
A research team wants to apply a circuit g on a dataset D (or C when encrypted) to find a statistical difference between two groups. The study goes as follow :

1. A research team requests a dataset to a databank
2. Based on some requirements (multiplicative depth, transformations to apply on the dataset, ...), the databank encrypts the dataset D with an FHE scheme
3. The databank sends the encrypted dataset C along with a public key pk and various evaluation keys evk, required to apply computations on C
4. The research team applies the circuit g on C and obtains $C' = \text{Eval}_{evk}(g, C)$, using the evaluation keys
5. The research team sends the resulting cipher C' to the databank, asking for decryption
6. The databank decrypts C' and sends the result to the research team

12.2 Threat model

In this scenario, we consider that the databank is the party that wants to protect its dataset from leakage. We will consider two security level : a *semi-honest* and a *malicious* research team.

In a real scenario, the research team can be a passive adversary against their will if they are victim of an attack. The research team can be the target of an attack where everything the they have can leak. The adversary would end up with the encrypted result and its corresponding decryption.

As the databank serves confidential and valuable data, a malicious adversary can be interested in this dataset to commercially exploit or sell it. In this case, the adversary would be more offensive in the attacks they perform.

Semi-honest adversary (honest-but-curious model) : This is the weakest security model. The attacker is an entity of the protocol. It will follow the protocol exactly but can read everything that is transmitted and use it to *passively* gain an advantage [NM21].

Malicious adversary : A stronger security model where the attacker can deviate from the protocol to gain an advantage. This adversary will *actively* try to gain an advantage, even if it needs to deviate from the protocol or send malformed messages in the protocol [MAL23].

12.2.1 Semi-honest context

We consider the research team is *honest-but-curious*. They want to compute a circuit on a dataset, they are willing to follow the exact protocol that was defined by the databank. The databank bases the security of its dataset on the trust they have toward the research team who they give their data to.

We consider that the research team might be the target of an external attack and that all the data they have might leak. An external attacker might get access to everything the research team has and receive from the databank: the public key, the evaluation keys, the encrypted dataset, the circuit they run, the resulting ciphertext and its decryption, sent by the databank.

This is a *passive* attack, as the attacker will not modify the data or the circuit that runs on the dataset, but they will exploit all the data they have.

12.2.2 Malicious context

In this context, we consider that the party requiring access to a dataset is malicious and will perform *active* attacks. They will announce a noise bound they computed, but that can be lower than what it is in reality. In this case, the only trustable party is the databank. Only what they control is safe, the rest must be considered malicious and not trustable.

12.3 Li and Micciancio's attack

As seen in Part II, the attack applies when the attacker has a ciphertext and the result of its decryption. We can see that, in this scenario, the research team, who should not have access to the whole dataset, ends up with a ciphertext and its decryption. Thus, the research team can apply the attack and recover the secret key, resulting in a complete loss of confidentiality. As they have the secret key, they can completely decrypt the dataset.

Even in a semi-honest context, this attack applies, as the attacker would get access to a ciphertext and its decryption, and thus could decrypt the whole dataset.

12.4 Noise flooding

To apply noise flooding to the decryption, the databank needs the worst-case noise bound of the circuit. In this scenario, the evaluation oracle is controlled by the research team, and the decryption oracle is controlled by the databank. Unfortunately, the latest has no way to be cryptographically sure about the exact circuit that was applied to the dataset nor if the announced noise bound is the real one. The research team does not have to announce the exact circuit they will use, and even if so, the databank has no cryptographic way to ensure it is the on really computed. Thus, the databank cannot compute the worst-case noise bound of the circuit with confidence, required to apply noise flooding on the decryption, and any reduction would make the scheme insecure.

12.4.1 Semi-honest context

In the semi-honest context, this protection is enough to protect the dataset from leaking. As the research team has to announce the noise bound computed, and as we consider the attack as passive, the databank can apply a gaussian noise, with parameters based on the noise bound, to protect the resulting plaintext. An attacker, getting access to everything the research team has, could not apply the attack seen, so the dataset is safe.

12.4.2 Malicious context

In the malicious context, this protection gives no proof about the circuit that was applied. In this case, the databank cannot cryptographically trust the research team and has no proof about the circuit. This protection does not give enough confidence to a conservative databank to protect its dataset.

As shown in [LMSS22], the noise estimation is tight, and even a small imprecision in the estimation is enough to perform the attack again. To perform the attack, the attacker has to announce a lower noise bound than the reality. In this case, the noise will not be fully protected from leakage and the attacker, using thousands of decryption, can run lattice reduction techniques to retrieve the secret key.

12.5 Output sanitization

In a similar fashion to noise flooding, output sanitization requires a bound on the noise of the circuit applied to the dataset to trim the r less significant bits of the decryption. Once again, the databank, which decrypts the result of the research team, has no way to known what was the exact circuit applied.

12.5.1 Semi-honest context

In the semi-honest context, the research team has to announce the noise bound of the circuit they used. This way, the databank can compute security parameters on the bound and trim the plaintext. This way, an attacker cannot use the attack to retrieve the secret key.

12.5.2 Malicious context

In the malicious context, as there is no proof about what circuit was used, the databank cannot be confident about the noise bound and cannot safely decrypt the result without risking a leakage. This protection does not work in this case.

In order to perform the attack, the attacker has to announce a lower noise bound than the real one. Then, using thousands of decryptions, the attacker can use lattice reduction techniques to perform a secret key recovery, which completely breaks confidentiality.

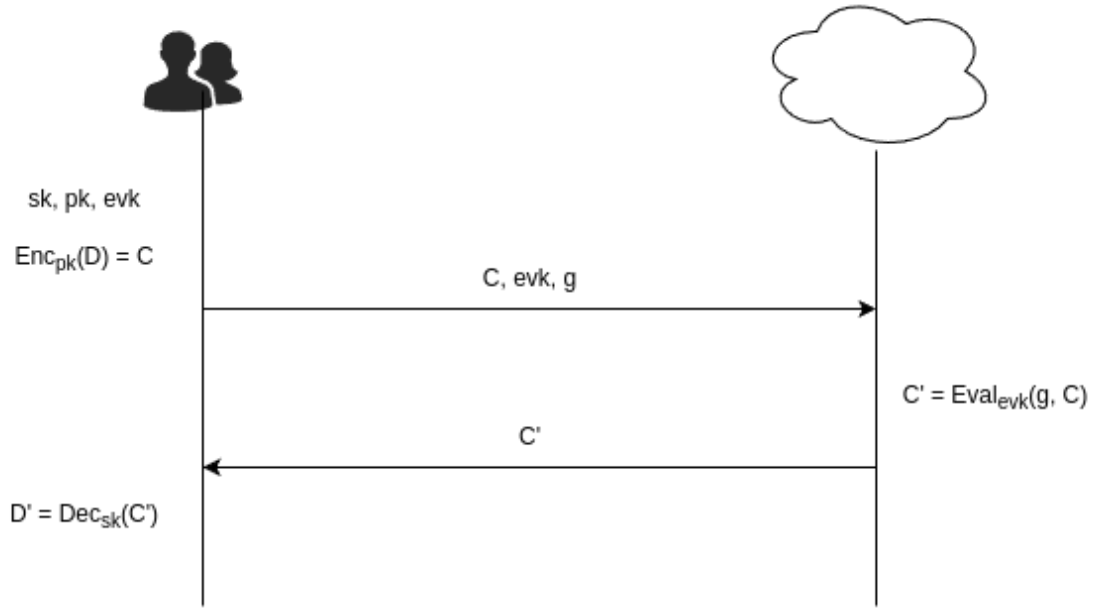


Figure 2: Sequence diagram of the confidential cloud computing scenario

13 Confidential computing in the cloud

Another common use of FHE is *confidential computing*. It consists of using cheaper computation power, often cloud-based, or at least of a not fully trusted third-party, to run heavy computations. In some cases, the data used in the computation is highly confidential, as we saw with DNA samples used in GWAS. Confidential computing is a solution, as encrypted data is no longer considered confidential, we can use cloud resources to compute calculations. FHE is a good fit for this use case, as we can encrypt data, so they are no longer confidential, but still evaluate operations on the ciphertexts.

This scenario, illustrated in Figure 2, shows how someone can use FHE to encrypt a dataset, send the ciphertext and some keys to a third-party cloud and apply a circuit on an encrypted dataset. In this scenario, we consider that the third-party cloud is *honest-but-curious* : it will try to read and decrypt the data, but it will not interfere with the computation or the data itself.

13.1 Threat model

The valuable asset here is the dataset that the user encrypts before sending to a third-party server. We consider this server *honest-but-curious*. The server will read and store everything it receives, and try to exploit it to retrieve the dataset. This is a *passive* attack.

13.2 Li and Micciancio's attack

As seen in Section 5, an attacker needs a ciphertext and its corresponding decryption to perform the attack. We can see in Figure 2 that the cloud, that we consider malicious here, only obtains a ciphertext, the circuit, the evaluation keys and the resulting ciphertext, which is not enough to apply the attack shown by Li and Micciancio. In this case, usage of FHE is safe without the need of additional defenses.

Yet, it is important to treat the resulting decryption as confidential. If the decrypted result is shared publicly, namely if the server gains access to it, the third-party will be able to exploit the

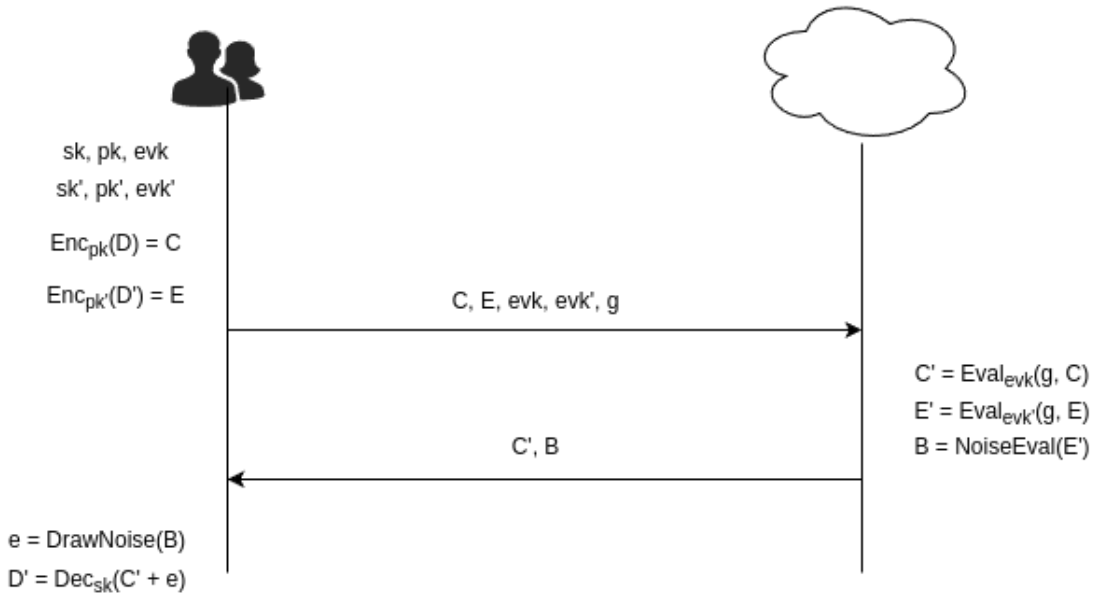


Figure 3: Confidential cloud computing with noise flooding

attack and recover the secret key, thus decrypt the whole dataset that was stored on the server. For this matter, it is interesting to consider how the countermeasures would mitigate this issue.

13.3 Noise flooding

As the attacker, the third-party cloud server is passive, and as the user controls the evaluation and the decryption oracles, we can apply noise flooding as a countermeasure. In parallel of the whole computation on the real dataset, the user will send a fresh public key and evaluation key on the server, along with a subset of the data, encrypted with the fresh public key. The user will need to compute the same circuit on this dataset to compute the worst-case noise bound, which is not confidential. Once the computation is done, the user can retrieve the result of the computation and the noise estimation from the cloud. The user will need to draw a random gaussian noise, with parameters based on the computed bound, and add this noise to the computation result before decrypting it. Doing this, the resulting plaintext will not be confidential anymore, and can be safely shared with anyone. The modified scenario is shown in Figure 3.

Even if this countermeasure successfully mitigates the attack, it has a serious performance overhead. The user needs to run the circuit twice, at least on a subset of the data, which can up to double the computation time. The user also needs to generate a fresh keypair, encrypt the dataset twice and transmit everything, which is not negligible as data encrypted with FHE and keypairs can have of size up to gigabytes [Per23].

13.4 Output sanitization

In this passive context, it is possible for the user to compute a noise bound based on the circuit applied, which means it is possible to use output sanitization as a mitigation against attacks. In this case, the user will use the cloud server like before to compute to circuit on the dataset, then retrieve the result. Before decrypting it, the user will compute a noise bound based on the circuit description. After that, the user will be able to trim the less significant bits of the result, which will make CKKS an exact scheme with overwhelming probability, and mitigates the attack. This

defense is lighter, as we do not need to run the circuit twice on the data, nor to encrypt and transmit the dataset twice.

14 Discussion

As we saw in the previous sections, the attack Li and Micciancio disclosed has a real impact on FHE security in many contexts. Simple things like decryption of ciphertexts become confidential, and leaking it could result in a complete key recovery.

The two countermeasures seen here have a significant performance overhead, especially noise flooding, that has to run the circuit twice. They also have an impact on correctness, as they add noise, for noise flooding, or round values, for output sanitization.

The scenarios we explored also highlighted a practical issue : IND-CPA-D is a security notion that protects against passive attacks. In practice, the owner of the secret key, who protects its dataset, relies on the exact definition of the circuit that was computed on the data. A common use-case of FHE is to allow third-party to run computation on a dataset without having to share this dataset. In this case, the data owner is only protected in the case of a passive attacker. If the attacker becomes active and lies about the definition of the circuit, they can apply the attack again.

The countermeasures seen protect FHE schemes against passive attack, but as soon as we consider active attacks, like lying on the circuit, the FHE schemes are threatened again. This is outside the scope of IND-CPA-D, but in practice, it could happen.

Future security notions will need to consider the circuit as a part of the schemes, but this will lead to another issue : circuit privacy. If the circuit is part of an intellectual property or a patent, it cannot be shared with the data owner. In this case, the third-party wants to protect its circuit, and the data owner wants to protect its data. Future security notions will need to consider circuit privacy while giving to data owners confidence about what circuit was used.

Table 1 summarizes the impact of the attack and the countermeasures in the scenario seen earlier.

	GWAS	confidential computing
Li and Micciancio's attack	Complete key recovery	Complete key recovery if the result is shared
Noise flooding : protection	Only for passive adversary	yes
Noise flooding : impact	performance halfed, correctness lowers	performance halfed, correctness lowers
Output sanitization : protection	Only for passive adversary	yes
Output sanitization : impact	Light performance overhead, correctness lowers	Light performance overhead, correctness lowers

Table 1: Summary of the two scenarios, the impact of the attack and of the countermeasures

Part VI
Alternatives

In some use-cases, FHE is a promising solution to do confidential computing, but can be limited by its security and the way it is used. In this Part, we will see how another cryptographic field, called *multi-party computation* (MPC) [EKR+18], tries to tackle the problem. We will also see how *Zero-Knowledge Proofs* (ZKP) [VKH23] can be of use when it comes to protect the integrity of FHE schemes in confidential computing contexts.

15 MPC

Multi-party computation (MPC) [EKR+18] is a field of cryptography allowing multiple parties to jointly compute a function over private inputs. Considering the issue seen in the GWAS scenario, in Section 12, we can consider the special case of MPC called *two-party computation* (2PC) [Yao86].

As its name suggests, *multi-party* computation schemes require that every party in the system participate in the computation. The schemes are *synchronous* and require that the databank performs heavy computations on the data and transmits hundreds of gigabytes [CWB18] over the network so the research team can compute its circuit.

15.1 Secure GWAS

In 2018, Cho et al. [CWB18] proposed a protocol based on MPC to compute a secure GWAS with multiple computing resources[4]. They argue that for a million individuals and 500'000 SNPs[5], they have to share a dataset of 36 TB, then exchange up to 435 GB during the synchronous computation.

Their protocol is secure in the *semi-honest*, or *honest but curious*, security model. As seen in Section 12 about the GWAS scenario, this security model might be insufficient for a conservative databank. They conclude their paper saying that taking the approach of the SPDZ protocol [DPSZ12, DKL+13] would make their protocol secure against active attacker, at the cost of *doubling* computation time and communication size.

16 ZKP

Zero-Knowledge Proof (ZPK) is a field of cryptography describing protocols where a *prover* wants to convince a *verifier* that they have a certain knowledge, without having to disclose the knowledge to the prover, and without being able to lie. Those protocols allow the prover to protect its knowledge and allow the verifier to ensure the prover does not lie.

16.1 vFHE

Viand, Knabenhans and Hithnawi (2023) [VKH23] presented an augmented version of the programming library called `OpenFHE` [ABBB+22]. By composing a IND-CPA secure FHE scheme with a generic ZKP system, they propose a *Verifiable FHE* (vFHE) scheme.

In the context of the GWAS scenario seen in Section 12, the research team can now convince the databank about what circuit they used (i) without having to reveal it and (ii) without being able to lie.

Even if this approach is promising, its performance overhead is for now big. FHE schemes work with polynomial rings and state-of-the-art ZKP frameworks work with fields in $\mathbb{Z}_p$. This major difference makes vFHE slow, adding a big performance overhead. To make this approach usable, we will need to define a ZKP framework working with rings of the same kind that FHE uses.

[4] a party of an MPC scheme that computes the function over encrypted data

[5] Single Nucleotide Polymorphism : A small variation in a patient's DNA among a larger dataset

Part VII
Conclusion

Through the two real use-case scenarios, we saw how IND-CPA-D security notion can fall short in practice. As it is only a protection against passive attackers, any use-case where active attackers could operate is still not safe. The countermeasures seen use the circuit to make FHE schemes safe, but yet do not consider them as a part of the threat model. In this case, an attacker can use correct homomorphic operations to a point that breaks the correctness of FHE schemes and ends up leaking the noise embedded in encryption.

We saw how ZKP can ensure integrity of FHE schemes, including the circuit in the security definition. Even if the construction is promising, we saw how it is not efficient enough yet to be used in practice and that further work needs to be done in this field.

References

[ABBB+22] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *proceedings of the 10th workshop on encrypted computing & applied homomorphic cryptography*, pages 53–63, 2022.

[BCM+24] Jean-Philippe Bossuat, Anamaria Costache, Christian Mouchet, Lea Nürnberger, and Juan Ramón Troncoso-Pastoriza. Practical q-IND-CPA-d-secure approximate homomorphic encryption. Cryptology ePrint Archive, Paper 2024/853, 2024.

[BR01] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. *Lecture Notes*, 2001.

[CKKS16] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016.

[CSBB24] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. On the practical CPAD security of "exact" and threshold FHE schemes and libraries. Cryptology ePrint Archive, Paper 2024/116, 2024.

[CWB18] Hyunghoon Cho, David J Wu, and Bonnie Berger. Secure genome-wide association analysis using multiparty computation. *Nature biotechnology*, 36(6):547–551, 2018.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits. In *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*, pages 1–18. Springer, 2013.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[EKR+18] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2(2-3):70–246, 2018.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. Cryptology ePrint Archive, Paper 2021/204, 2021.

[LM20] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Paper 2020/1533, 2020.

[LMSS22] Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. Securing approximate homomorphic encryption using differential privacy. Cryptology ePrint Archive, Paper 2022/816, 2022.

[LPR12] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. Cryptology ePrint Archive, Paper 2012/230, 2012.

[MAL23] Daniel Morales, Isaac Agudo, and Javier Lopez. Private set intersection: A systematic literature review. *Computer Science Review*, 49:100567, 2023.

[MTZ19] Ilya Mironov, Kunal Talwar, and Li Zhang. R\'enyi differential privacy of the sampled gaussian mechanism. *arXiv preprint arXiv:1908.10530*, 2019.

[NM21] Bhawna Narwal and Amar Kumar Mohapatra. A survey on security and authentication in wireless body area networks. *Journal of Systems Architecture*, 113:101883, 2021.

[Per23] Pascal Perrenoud. Chiffrement de données génétiques. Bachelor's thesis, HEIG-VD, 2023.

[RAD+78] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Reg10] Oded Regev. The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204, 2010.

[VKH23] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. Verifiable fully homomorphic encryption. *arXiv preprint arXiv:2301.07041*, 2023.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.

Part VIII
Appendices

17 Attack source code

```
#include <cassert>

#include "openfhe.h"
#include "helpers.h"

using namespace lbcrypto;
using namespace std;

int main() {
    puts("Generate cryptocontext");
    constexpr uint32_t batchSize = 1 << 15;

    CCParams<CryptoContextCKKSRNS> parameters;
    parameters.SetMultiplicativeDepth(20);
    parameters.SetScalingModSize(40);
    parameters.SetBatchSize(batchSize);
    parameters.SetRingDim(batchSize << 1);
    parameters.SetSecurityLevel(HEStd_128_classic);
    parameters.SetScalingTechnique(FLEXIBLEAUTO);

    // Lines added to tweak the noise flooding parameters
    parameters.SetExecutionMode(EXEC_EVALUATION);
    parameters.SetDecryptionNoiseMode(FIXED_NOISE_DECRYPT);
    parameters.SetNoiseEstimate(0.);

    const CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);
    assert(cc->GetRingDimension() == 1 << 16);

    cc->Enable(PKE);
    cc->Enable(LEVELEDSHE);

    puts("Generate keys");
    auto keys = cc->KeyGen();
    cc->EvalMultKeyGen(keys.secretKey);

    puts("Generate random vector");
    vector<complex<double>> resVec;
    resVec.reserve(batchSize);
    // Generate a random complex vector with norm of maximum 1
    const auto xvec = get_random_complex_vector(batchSize, 1.);
    Plaintext ptxt = cc->MakeCKKSPackedPlaintext(xvec);

    puts("Encrypt data");
    auto c = cc->Encrypt(keys.publicKey, ptxt);

    puts("Evaluate circuit");
```

```
    Ciphertext<DCRTPoly> ctRes = c->Clone();
    ctRes = c * c;
    ctRes = cc->EvalMult(ctRes, 1.); // Force rescale

    puts("Decrypt result");
    Plaintext ptxtRes;
    cc->Decrypt(keys.secretKey, ctRes, &ptxtRes);

    puts("Decode result");
    shared_ptr<CKKSPackedEncoding> ptxtResEncoded =
    ↪  std::dynamic_pointer_cast<CKKSPackedEncoding>(ptxtRes);
    Poly ptxtResPoly = ptxtResEncoded->GetElement<Poly>();
    ptxtResPoly.SetFormat(COEFFICIENT);
    resVec = ptxtResEncoded->GetCKKSPackedValue();

    Plaintext ptxtReEnc = cc->MakeCKKSPackedPlaintext(
        resVec,
        ctRes->GetNoiseScaleDeg(),
        ctRes->GetLevel()
    );
    shared_ptr<CKKSPackedEncoding> ptxtReEncEncoded =
    ↪  std::dynamic_pointer_cast<CKKSPackedEncoding>(ptxtReEnc);
    ptxtReEncEncoded->Encode();
    DCRTPoly ptxtReEncCRT = ptxtReEncEncoded->GetElement<DCRTPoly>();

    // Retrieve the two components of the ciphertext
    DCRTPoly cb = ctRes->GetElements()[0];
    DCRTPoly ca = ctRes->GetElements()[1];

    ca.SetFormat(EVALUATION);
    cb.SetFormat(EVALUATION);

    puts("Attack key");
    DCRTPoly caInv =  ca.MultiplicativeInverse();
    ptxtReEncCRT.SetFormat(EVALUATION);
    DCRTPoly sGuess = (ptxtReEncCRT - cb) * caInv;

    puts("Extract secret key for comparison");
    PrivateKey<DCRTPoly> sk(keys.secretKey);
    size_t towersToDrop =
    ↪  sk->GetPrivateElement().GetParams()->GetParams().size() -
    ↪  cb.GetParams()->GetParams().size();
    auto s(sk->GetPrivateElement());
    s.DropLastElements(towersToDrop);

    if (sGuess == s) {
        puts("Attack successful");
        return 0;
    } else {
        puts("Attack failed");
        return 1;
    }
}
```