Attacking FHE-based applications by software fault injections

Ilaria Chillotti[1], Nicolas Gama[2,1], Louis Goubin[1]

[1] Laboratoire de Mathématiques de Versailles, UVSQ, CNRS, Université Paris-Saclay, 78035 Versailles, France
[2] Inpher, Lausanne, Switzerland

Abstract. The security of fully homomorphic encryption is often studied at the primitive level, and a lot of questions remain open when the cryptographer needs to choose between incompatible options, like IND-CCA1 security versus circular security or search-to-decision reduction. The aim of this report is to emphasize the well known (and often underestimated) fact that the ability to compute every function, which is the most desired feature of Homomorphic Encryption schemes, is also their main weakness. We show that it can be exploited to perform very realistic attacks in the context of secure homomorphic computations in the cloud. In order to break a fully homomorphic system, the cloud provider who runs the computation will not target the primitive but the overall system. The attacks we describe are a combination between *safe-errors attacks* (well known in the smart cards domain) and *reaction attacks*, they are easy to perform and they can reveal one secret key bit per query. Furthermore, as homomorphic primitives gets improved, and become T times faster with K times smaller keys, these attacks become KT times more practical. Our purpose is to highlight the fact, that if a semantically-secure model is in general enough to design homomorphic primitives, additional protections need to be adopted at a system level to secure cloud applications. We do not attack a specific construction but the entire idea of homomorphic encryption, by pointing out all the possible targets of this attack (encrypted data, bootstrapping keys, transciphering keys, etc.). We also propose some possible countermeasures (or better precautions) in order to prevent the loss of information.

1 Introduction

Nowadays, more and more data is stored in the cloud, sometimes without the owner even being aware of it. This includes all websites and databases hosted on public servers, but also data gathered by public e-mail providers, from posts on social networks, research patterns in major search engines, phone records, GPS information on mobile phones. All this information has a huge economical value. The use of secure protocols like https, imaps prevent man-in-the-middle attacks, but for these protocols, the cloud provider itself remains the legitimate recipient, so he keeps an unrestricted access to this valuable information. It is in this case natural to consider an honest but curious model.

With homomorphic encryption, and more generally with end-to-end encryption, the setting is completely different, because the sensitive data is encrypted with a key that the cloud does not control. Moreover, if the underlying cryptosystem is semantically secure, it is natural to imagine that some cloud providers will stop offering their services for free, and some others may even switch to more powerful attacks, that could include data tampering, in order to regain access to that information. To capture the second scenario, the cloud will therefore mainly be considered untrusted and malicious in this paper. However, the cloud will do his best to ensure that his attacks remain undetected: indeed, visible attacks would break the client's trust, and would incite him to stop using the cloud services, which is obviously not the goal. Thus, in our security model, the cloud[3] is a discreet and cautious adversary.

Security model. The aim of this report is to prove that the security of an homomorphic scheme cannot be assessed at the primitive level. The broader construction which uses the scheme must also be taken into account in the analysis.

Cryptographic primitives are usually classified according to a standard hierarchy of security notions: IND-CPA security, or semantic security, is generally viewed as the standard security requirement for homomorphic encryption. State of the art homomorphic primitives, either over the integers or based on LWE or on principal ideal problems are usually proved semantically secure by a direct reduction to an intractable problem. IND-CCA2 security is one of the strongest security level, which essentially means that an attacker who chooses two messages μ_1,μ_2 and receives the encryption c of one of them, cannot decide which message was encrypted, even if he is granted an oracle that can decrypt everything but c. No homomorphic primitive can be IND-CCA2, since their main feature is to be malleable. In this case, choosing feature over security seems to be the only possible choice. Interestingly, the same question arises with the IND-CCA1 security notion, which differs by the fact that the decryption oracle may only be used before the challenge c is published. Some recent technical papers [37], [47], [14] prove that almost all the proposed somewhat homomorphic schemes until now are vulnerable to non-adaptative chosen ciphertext attacks (CCA1), which all reveal the decryption key. In fact, all the somewhat homomorphic primitives which are based on LWE or its ring variant, and also those based on integers, rely on a *search-to-decision* reduction. This basically means that, being able to decrypt an homomorphic primitive is equivalent to recovering the private key, which itself reduces to solving some intractable problem in the worst case. If this feature allows elegant IND-CPA security proofs, it also means that any (CCA1) decryption oracle totally breaks the scheme, just by following the steps of the search-to-decision reduction. A similar remark also applies to the bootstrapping principle and the circular security assumption, which also contradicts IND-CCA1 security. Reciprocally, it appears that IND-CCA1 somewhat

[3] In this report, the term cloud denotes not only a cloud server, but also any possible malicious actor that can access the data stored/processed, inject errors in order to perform an attack and observe all the communications between the cloud server and the clients.

homomorphic schemes, like [37] rely on ad-hoc security assumptions, which are not easy to compare with more traditional LWE or SIS assumptions, or worst-case lattice problems. At a primitive level, it seems that we are faced with an impossible choice between security requirements (IND-CCA1) or features (FHE, bootstrapping, worst-case assumption).
To make a choice, we need to see beyond the homomorphic primitive and consider a broader view of the overall system. Namely, are there realistic situations where the cloud would have access to some (possibly weak) decryption oracle?

We would also like to mention the notion of Indistinguishability against (Chosen) Ciphertext Verification Attack (IND-CVA), which is closer to what we will study in the rest of the report. This notion is usually defined at the primitive level, as an intermediate security notion stronger than IND-CPA security but weaker than IND-CCA security[4]. The idea of CVA dates back to the "Reaction Attacks" of Hall, Goldberg and Schneier [33] in 1999. Instead of targeting the hard underlying problem, their idea was to observe the "reaction" of the private key owner, when he decrypts a tampered ciphertext. Knowing whether it still decrypts into a valid plaintext, can sometimes provide the attacker with some information on the message or on the secret key. The attack by Bleichenbacher [3] in 1998 against the RSA-PKCS#1 padding uses similar principles. These notions have been later regrouped and re-defined in 2009 by Hu, Sun and Jiang in [34] as Indistinguishability under Ciphertext Verification Attack (IND-CVA). At a primitive level, IND-CVA is usually achieved by requiring a strong padding condition on the message space, like the RSA-OAEP padding. In the context of homomorphic encryption, IND-CVA does not seem to make sense at the primitive level, especially when ciphertexts are full-domain, or if the message space is too small to encode an intrinsic constraint (like the $\{0,1\}$ message space). However, this notion should be extended at the system level, where many individually valid ciphertexts are combined together to form some possibly meaningful information. In this case, an attacker could replace a few ciphertexts with other valid ciphertexts, and see if the overall information looks still meaningful to the recipient. As already said in [37] and [46], this oracle exists in practice, whenever the client asks the cloud for computations. If the client thinks that the data returned by the cloud is incorrect, especially in an economical model where the client pays the cloud per running times, he will certainly ask the cloud for a (free) re-computation of the result, otherwise he will just accept it. This natural behaviour can be viewed as the response of a CVA oracle, and used as an instrument to retrieve sensitive information. As we show in following sections, it leaks one bit of information per "query".

Safe-Errors and Reaction. The point we want to stress is that, even if the primitives at the base of homomorphic encryption schemes are secure, the whole construction around presents several failures, exploitable by a malicious attacker. Previous attacks against somewhat homomorphic encryption schemes usually targeted mathematical weaknesses inside the primitives. As in [37] and [46], the

[4] Figure 1 of [20] summarizes the connections between the different notions of security.

attacks we describe are more generic. They will be presented as if the primitive was a perfect semantically secure and plaintext-aware black-box homomorphic scheme. The attacks we will present are strongly inspired by side-channel attacks in the smart-card domain, but without the need for particular equipments (lasers, probes, etc.). In the cloud scenario, they are simple software attacks. Overall, we perform *safe-error attacks* [45], whose principle is the following: the malicious actor changes a few bits during a computation, and then, he checks if this modification triggers an error later in the process. In the cloud setting, the semantic security of the black box primitive, prevents the attacker from directly obtaining a non-trivial information on the input. This is the second main difference with the smart card domain, where the result can instead be measured immediately after the attack. This is the very reason why in the cloud domain, the attacker needs an observable *reaction* from the client. If no one is aware of this attack, the model is quite realistic: for instance, if the attacker's modification impacts the response at a point that the decrypted text looks gibberish, the natural reaction of the client is to notify the cloud and ask him to re-run the computation. We also want to emphasize the contrast between the simplicity of these attacks, which may be qualified as "trivial" (they require almost no mathematical background at all), and the fact that they can be conducted in practice for any use-case of the cloud until now. Furthermore, if heavy countermeasures may be deployed for somewhat homomorphic schemes, they seem impractical and useless for fully homomorphic schemes, underlining an additional antagonism between efficiency and security.

Our contribution. This paper is a technical report on homomorphic encryption. The goal is to highlight a (well known) weakness of homomorphic encryption schemes, that seems to be often ignored or underestimated. The class of attacks we describe targets the broader constructions of such schemes. The attack is easy to perform and it is dangerous for real world applications. We are not the first ones talking about reaction attacks, side-channel safe-error attacks or about the malleability of homomorphic encryption schemes. Our purpose is rather to stress on the fact that these weaknesses are practically exploitable by a malicious adversary and that the attack is easy to perform and realistic. Fully homomorphic encryption will not be a secure solution for cloud applications, until a valuable solution against this kind of attacks will be found. In our opinion, this family of attacks should be taken more into account for future security analysis. Previous papers such as [37] and [46] present similar attacks. Our contribution is a slightly easier (but as dangerous) attack, the analysis of a larger spectrum of targets and the proposition of some possible precautions and countermeasures.

Paper organization. The paper is composed by two main Sections 3 and 4, preceded by a small background on Homomorphic Encryption (Section 2). In Section 3 we describe the safe-error attack, detailing both the attack targeting the data encrypted and stored in the cloud, and the attack against the algorithm used to do the computations. We propose a few countermeasures, analyzing their usefulness. In Section 4 we apply the same type of attack on schemes which use bootstrapping (and trans-ciphering), and we also study some countermeasures.

2 Background : Homomorphic Encryption

The notion of Homomorphic Encryption dates back almost 30 years, when Rivest-Adleman-Dertouzos [40] introduced the notion of *privacy homomorphisms*. The idea is to be able to perform homomorphic operations on encrypted data, without the needing to decrypt.
Let $\mathcal{M}$ be a valid message space and $\mathcal{C}$ a valid ciphertext space. A homomorphic encryption scheme is composed of four algorithms:

– The **key generation** algorithm KeyGen: given a security parameter λ, it generates a private key for the client and, if necessary, a corresponding public key.
– A deterministic **decryption** algorithm Dec: given a ciphertext $c \in \mathcal{C}$ and a private key, it outputs a message $m \in \mathcal{M}$.
– A randomized **encryption** algorithm Enc: given a message $m \in \mathcal{M}$ and a key (private, or public depending on the case), it outputs a ciphertext $c \in \mathcal{C}$ such that $\mathsf{Dec}(c) = m$.
– A (possibly randomized) **evaluation** algorithm Eval: it takes in input the description of a k-variable function $\varphi : \mathcal{M}^k \to \mathcal{M}$ and k ciphertexts $c_1, \ldots, c_k \in \mathcal{C}$ of unknown messages $m_1, \ldots, m_k \in \mathcal{M}$. It outputs a ciphertext $c \in \mathcal{C}$ such that $\mathsf{Dec}(c) = \varphi(m_1, \ldots, m_k)$.

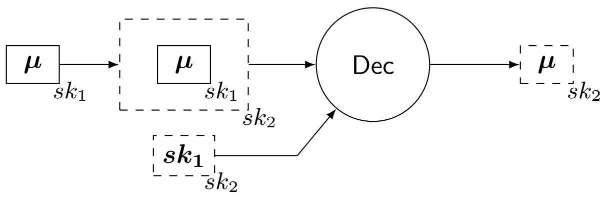
We suppose that the primitive is always semantically secure, or even IND-CCA1 or plaintext-aware in the sense of [37].

Depending on the evaluation algorithm, we can define different types of homomorphic schemes:

– *Somewhat (Partially) Homomorphic*: are the schemes able to evaluate a (possibly limited) set of Boolean functions. Known examples are RSA [41], ElGamal [22], Pailler [39], Boneh, Goh and Nissim [4], and all other Homomorphic schemes (described later).
– *Leveled Homomorphic*: are the schemes such that for all finite set of Boolean functions, there exists a parameter set that allows to successfully evaluate these functions. The choice of the parameters is done at the beginning. In general, the client chooses the parameters corresponding to the depth of the circuit he wants to evaluate.
– *Fully Homomorphic*: are the schemes for which there exists a parameter set able to evaluate all functions, without any limitation on the depth of the circuit to be evaluated. In particular, FHE schemes are characterized by their unique ability to evaluate their own decryption function [24].

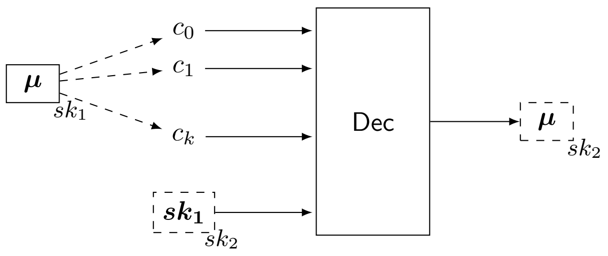
In the following, we will use the abbreviation HE for Homomorphic Encryption, SHE for Somewhat Homomorphic Encryption, LHE for Leveled Homomorphic Encryption and FHE for Fully Homomorphic Encryption. The main interest in HE schemes is that the evaluation can be done publicly without decrypting the messages, and thus without revealing any secret information.

At the time of writing, various semantically secure FHE schemes have already been proposed, and they are all constructed from specific noisy SHE/LHE schemes with a very efficient decryption algorithm, using the bootstrapping technique introduced by Gentry in 2009 [24]. The presence of noise inside ciphertexts is necessary for security purposes, but each time an operation is performed, it grows. Once the noise level reaches a certain amount, a correct decryption cannot be guaranteed. The number of operations is then limited. In 2009, Gentry proposed a technique, called bootstrapping [24], that can be used to “make Fully” an SHE/LHE scheme. The scheme contains an additional Refresh procedure, which takes as input a fixed bootstrapping key BK, and a ciphertext $c \in \mathcal{C}$ with a high amount of noise, and returns a ciphertext $c' \in \mathcal{C}$ of the same message, with less noise. Initially, the refreshing procedure was generically described as the homomorphic evaluation of the decryption circuit on each bit of the noisy ciphertexts. In order to protect the decryption, an additional layer of encryption is added to the ciphertexts. Furthermore, the secret key is needed in order to decrypt. So the evaluation circuit takes in input an additional entry, the Bootstrapping Key BK: this latter is the list of encryptions of each individual private key bit.



Original bootstrapping idea. *The straight line boxes represent the first encryption layer, with respect to a secret key sk_1, while the dashed line boxes represent the second encryption layer, with respect to a secret key sk_2. In order to reduce the noise, the decryption circuit with respect to the key sk_1 is evaluated homomorphically. The security is guaranteed by the second layer of homomorphic encryption. The result will be a new ciphertext encrypting the initial message, but with less noise. The encryption of sk_1 with the new key sk_2 is called bootstrapping key.*

Many improvements to Gentry’s bootstrapping principle consisted in removing the re-encryption phase by delaying as much as possible the operations depending on the secret key, and to make them as linear as possible. Instead of re-encrypting the noisy ciphertexts with an additional layer of homomorphic encryption, new bootstrappings evaluate the decryption circuit directly on the bits of the noisy ciphertext by using the encryption of the secret key (i.e. the bootstrapping key), as schematized in next figure.



The bootstrapping proposed in [6] has an appealing asymptotic polynomial complexity, refinements of [1] make the decryption algorithm almost quartic in the security parameter, and drops to quasi-quadratic in [21], allowing to refresh a ciphertext in 0.6s between two NAND gates. Further improvements of the [21] scheme are proposed in [2] and in [15] (~ 30 times faster with ~ 40 times smaller keys). FHE schemes base their security on hard problems, in general hard problems in lattices (and ideal lattices) [24], [26], [42], [43], [27], [28], [25], [29], [30]. Recently, the most promising FHE schemes rely on the Approximate GCD problem [44], [18], [19], [17], or the Learning With Errors (LWE) problem (and its variants) [6], [7], [8], [5], [32], [9], [21], [2], [15].

3 Safe-errors and reaction attacks in the cloud

In this section, we establish a parallel between smart-cards circuits and homomorphic computations in a cloud. Both are circuits which operate on hidden data. As instance, the most frequent purpose of a smart card is to compute digital signatures, using a private key which should never be extracted, even by its legitimate owner. Similarly, the client may provide the bitwise homomorphic encryption of a private key to the cloud, and let the cloud homomorphically compute (encrypted) signatures of encrypted data. It has long been known that smart-cards are vulnerable to side-channel attacks. Some of the attacks are passive, where an attacker gets information on the computation by measuring the running time, the power consumption, some electromagnetic field, or any other side channel information. For example, if the computation of an RSA [41] or (EC)DSA signature [36], [35] uses a naive square-and-multiply/double-and-add algorithm, the sequence of operations strongly depends on the number of "ones" and on their positions in the secret key. In smart-cards, these attacks are usually prevented by making the circuit data-independent or *oblivious*, if possible even SIMD parallel. Other perturbations may also be introduced, like adding other arbitrary computations that are not used in the sequel. In homomorphic computations, the above countermeasures are mandatory by design: all circuits must be oblivious, since the semantic security of homomorphic primitives prevent the cloud from getting any information on the data. In practice, it means that the number of iterations of all for loops are publicly known in advance, there is no while loop, no data-dependent jump, and the standard way of evaluating an if-then-else block is to fully evaluate both possibilities, and in the end, to pick

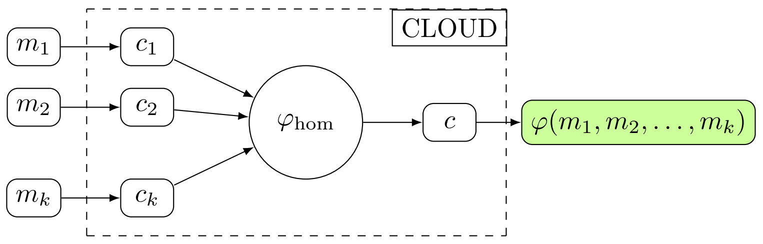
the right result. It also means that simple power or running-time analysis are irrelevant in the cloud, however, fault attacks are still meaningful.
In an active attack on smart cards, the attacker tampers the computation and observes the result in order to gain information on the hidden data. For smart cards, the hardest part is to introduce the fault at a precise moment of the computation, because this usually requires a dedicated physical device. For this reason, and also the fact that most of these attacks require the card pin code, the attacker is in general the legitimate owner of the card, who just wants to retrieve the hidden key. Furthermore, such attacks have variable success probabilities, because some parts of a circuit are easier to overheat than others, and all this must be taken into account in their analysis.
By contrast, the cloud provider has a direct and unlimited software access to the whole circuit which is evaluated. He may tamper the result of any gate of his choice with probability 1, at any time. The physical protection of the circuit is replaced by the mathematical shield, which usually consists in saying that every bit is encrypted with a semantically secure scheme. This indeed guarantees that, without an external feedback, no attacker may conduct, on his own, any attack (active or passive) that can reveal sensitive data.

However, some composition of schemes may grant the attacker a weak version of a decryption oracle. For instance, the cloud provider can observe the reaction of a person who owns the private key and who will process the result afterwards, and hope that his behaviour reveals information on a part of the data.

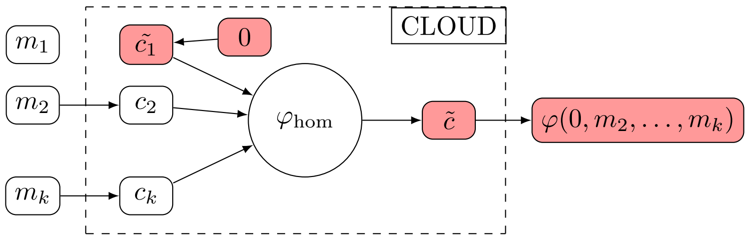
3.1 Attacking the data

The idea of *safe-error* attacks is that during the execution of an algorithm, a fault is injected in a precise point. For some secret values, this fault has no effect on the final result, and for some other values, it changes the result completely. So, by observing the correctness of the final result (or in our context the reaction of the legitimate receiver), the attacker can retrieve the secret value targeted by the attack.

By using the techniques from smart-cards on the cloud, we can recreate a more realistic scenario than those involving a universal decryption oracle. Suppose that we want to evaluate a function φ on k ciphertexts $c_1, \dots, c_k$, encrypting k messages $m_1, \dots, m_k \in \{0, 1\}$ respectively. If the evaluation is performed without any error, the result will be a ciphertext c encrypting $\varphi(m_1, \dots, m_k)$.



But if an error is introduced, the final result could be wrong. As instance, suppose the cloud wants to retrieve the value encrypted in ciphertext c_1. He could just replace this latter with a different ciphertext $\tilde{c_1}$ encrypting 0. Then he performs the rest of the computations correctly. The result will be a faulted ciphertext $\tilde{c}$ encrypting $\varphi(0, m_2, \ldots, m_k)$.



If the decrypted results $\varphi(m_1, m_2, \ldots, m_k)$ and $\varphi(0, m_2, \ldots, m_k)$ are different, it means that an error occurred, so m_1 is different from 0. Otherwise, the guess of the cloud was correct. The cloud cannot verify this equality, because it can access only semantically secure encrypted data.
It is here that the *reaction* of the client plays a crucial role. In our model, upon reception of the ciphertext computed by the cloud, the client decrypts the message and applies some likelihood test. Then, the cloud provider may observe two possible reactions[5] from the client: if the likelihood test failed, the client may directly complain that the result is wrong, and ask for a free re-computation. If the likelihood test succeeds, the client will simply accept the result. In both cases, the cloud has received the information he needed to understand if the safe-error was correct or not, and so retrieve the value of the encrypted bit. By repeating the same procedure k times, the cloud will retrieve all the k bits.

As a toy example, we can think of a client asking the cloud to compute homomorphically some signatures. He stores on the cloud the encrypted signature secret key and asks to sign some data. If a safe error attack is performed on the signature secret key, the verification of the signature ("likelihood test") will detect immediately if an error was introduced.

3.2 Attacking the algorithm

The spectrum of targets of safe-error attacks is wide. As well as an attack on data, the attacker could target the algorithm itself. As instance, he could temper the temporary variables used in the algorithm. In this section we will use the RSA square-and-multiply always procedure (algorithm 1) as a concrete example.

[5] Actually there exists some situations where the cloud could observe multiple reactions from the client, but this case will be treated later in the paper.

Algorithm 1 RSA square-and-multiply always

```
1: Input : a message $m$, a secret key $d = (d_0, \ldots, d_{k-1})$, a modulus $N$
2: Output : $m^d \mod N$
3: $t \leftarrow m$
4: for $i = k - 2$ to 0 do
5:     $t_0 \longleftarrow t^2 \mod N$
6:     $t_1 \longleftarrow t_0 \cdot m \mod N$          ▷ A safe error could target this line
7:     $t \longleftarrow t_{d_i} \mod N$
8: end for
9: return $t$
```

RSA square-and-multiply always. *The RSA square-and-multiply always algorithm as it would be implemented on a smart card, or homomorphically in the cloud. In particular, it contains no data-dependent* if *conditions. By introducing an error in the execution of line 6, an attacker learns whether the variable t_1 is used or not (line 7), and therefore, he learns the key bit d_i. This attack is particularly efficient, since the attacker only needs to know that he modifies the value t_1, but he does not need to control the replacement value.*

In the process of making an algorithm oblivious, all conditional structures like "*if a then B else C*" are replaced by "*evaluate B, evaluate C, and output $aB + (1 - a)C$*". This strategy prevents timing attacks on smart cards, because the set of operations that are executed do not depend anymore on a. However, it is a natural target for safe-errors attacks. Indeed, if an attacker may tamper the execution of the block B, and see if it has a consequence in the following. If it is so, it means that the condition a was true.

In Algorithm 1, in order to homomorphically evaluate $m^d \bmod N$ via some oblivious Horner variant, each intermediate step computes $t_0 = t^2 \bmod N$ and $t_1 = m \cdot t^2 \bmod N$, where t is initially set equal to m, and then chooses which value to select depending on the current key bit (0 or 1 respectively). If an attacker tampers the multiplication by m, replacing the value of t_1 with any other random (plaintext) data, this safe-error will affect the final result with almost certainty if and only if the current private-key bit is equal to 1. Indeed, if the key bit was null, this tampered data would simply have been discarded. Once again, the attacker learns one key bit by observing the reaction of the recipient to safe errors, but this time, he does not target the data directly, but a portion of the algorithm he is able to understand.

3.3 Countermeasures

In this subsection, we study possible countermeasures to these simple attacks, and show that the client must be very strict concerning its trust model towards the cloud.

Add plaintext awareness (asymmetric vs symmetric encryption)? One possible countermeasure we may think of, is to try to use symmetric plaintext-aware schemes. Indeed, the direct attack on the hidden data requires the cloud to replace a ciphertext with a valid ciphertext of 0 (or 1), and the idea of the countermeasure is to prevent him from being able to generate such ciphertext. Unfortunately, this countermeasure does not really apply to homomorphic constructions. The universality[6] of these schemes allows them to evaluate any function, including the constant 0 or the constant 1 functions, and thus, to forge valid ciphertexts of 0 or 1 at will. For example, if the homomorphic scheme includes the (homomorphic) XOR or the (homomorphic) NAND gates, a valid ciphertext of 0 may be obtained by computing $XOR(c, c)$ for any valid ciphertext c, and a valid ciphertext of 1 can be obtained as $NAND(c, NAND(c, c))$.

Work on large blocks (non-bitwise encryption)? A second proposal would be to increase the length of the message space, so that homomorphic primitives operate on larger blocks (for instance on 128-bits blocks) instead of just zeros and ones. In this case, an attacker would only be able to tamper a whole block of data, and, unless he guesses the right 128-bit value, it would always produce a reaction from the client, which does not bring much information. But once again, this approach is bound to fail for two reasons: it does not prevent the case where the attacker targets a value which is not used in the sequel, which may be replaced by any random (plaintext) value. And most importantly, once again, the universality of homomorphic primitives allows the attacker to homomorphically evaluate any function on a 128-bit block, including the function ϕ_i which takes a block as input and sets its i-th bit to zero. The attacker can therefore replace a hidden block Hom.Enc(m) of data with Hom.Enc($\phi_i(m)$) and perform the attack. In this case, the reaction of the client reveals the value of the i-th bit of m, as if the homomorphic primitive was just operating on binary messages.

Add obfuscation? The nature of the two attacks strongly suggest that the location of important data bits must be hidden from the cloud, and the function must be impossible to reverse-engineer or to understand. Achieving these goals is usually called obfuscation. In order to prevent the attack on the data bits, an idea would be to add some error correcting code. For instance, instead of computing $f(x)$ where x represents n bits of data, the algorithm would first compute $x = g(y)$ where g is some random error correcting code of large distance $d > n$, and the hidden data y has at least $n + 2d$ bits. In order to successfully invalidate the result, an attacker would then need to flip at least d bits of data, *i.e.* to fix $2d$ bits to arbitrary values. Of course, this is more a hint than an actual workaround, for the following reasons:

- The code should not be perfect, and if possible non linear. Else, the reaction attack of [33] against the McEliece cryptosystem can be adapted. Basically, fix bits of data one by one to arbitrarily values until the client reacts. At that point, if the code is perfect, there are exactly $d + 1$ errors, so the last bit set is wrong, and since there are already d errors among the others bits fixed,

[6] Universality intended as the property of FHE schemes to evaluate all functions, due to their large malleability.

each additional error induces a reaction from the client. So the classical safe error can be conducted on the remaining data bits one by one.
- The composition between the homomorphic decoding $x = g(y)$ and the function computation $f(x)$ must be obfuscated to the cloud. Else, the cloud provider may just run the code part homomorphically, and once it has the bits of x, do the regular attack on them.
- For the same reason, the part of the circuit which computes f must be obfuscated, else the attack on the function can be directly performed.
- If g represents the decoding function of a non-linear random looking code, its complexity will have a serious impact on the size of the overall circuit, and therefore on the parameter sizes of the homomorphic scheme. Besides, the fact that the whole composition $f \circ g$ must be obfuscated worsens the situation.

Overall, homomorphic encryption was often presented as the systematic way of obfuscating a computation. But now, we see that in order to resist against simple safe-errors, the computation has to be obfuscated again with something larger. Obviously, taking another homomorphic scheme as the larger obfuscation seems to create a vicious circle. And if the computation is protected with another ad-hoc obfuscation, then what is the purpose of the first homomorphic scheme?

Do not forgive any mistake It seems that the safest approach for the client is to distrust the cloud immediately whenever he receives a ciphertext which does not correspond to any realistic result. For instance, if the goal of the homomorphic scheme is to compute signatures and the received signature is invalid, or if the data looks like a random binary sequence instead of a plaintext message. This also implies that the underlying homomorphic primitive must be error-free. Indeed, many LWE-based homomorphic schemes which were recently proposed, including [1], [21] and [15] were able to drastically reduce the parameter sizes by requesting that the homomorphic operations are randomized, but with the counter-effect that even an honest homomorphic computation has a small error-probability per gate in the circuit. Namely, the parameter set proposed in [21] and [15], which allows to bootstrap the scheme in less (or much less) than a second, has an error probability of 2^{-32}, which means that the result of an homomorphic computation may be wrong even if no safe-error was introduced. Due to the nature of the attack we are pointing out, allowing errors can be really devastating, since the client would not be able to distinguish between an attack and an honest error due to the homomorphic primitives.

4 Attacking the bootstrapping principle

In the previous section we showed how to attack a secret value stored in the cloud and encrypted with a LHE scheme: our target was not the scheme itself. Instead, in this section we apply the attacks directly to the bootstrapping principle to target the secret keys of the HE scheme, highlighting that a safe-error and reaction attack is particularly efficient in this case. There are two use-cases of the bootstrapping mechanisms: the first one is used to proxy-re-encrypt a message, encrypted with any scheme, into a bit-wise HE encryption of the same message. The second notion is the original bootstrapping proposed by Gentry in order to refresh noisy ciphertexts, and to turn a somewhat/leveled homomorphic scheme into a fully homomorphic one. The first notion is a technique that we denote as *trans-ciphering*, which is a commonly proposed optimization in order to drastically reduce the communications from the client to the cloud.

4.1 Trans-ciphering

Homomorphic schemes are known to have a very high ciphertext vs plaintext expansion rate. For instance, the FHEW implementation of [21] (and the variant of [15]), which is one of the most compact schemes, uses about 16000 bits of ciphertext to encrypt one bit of message for 128-bit security. This rate is particularly annoying when the client has to upload his input data to the cloud over the network. A solution that has been proposed in [38] is to encrypt the data using a traditional symmetric algorithm like AES-CBC, and to send it together with the bitwise-homomorphically encrypted AES key. That way, the data is sent using an amortized 1-1 encryption ratio, which saves a considerable amount of network bandwidth.
In order to perform homomorphic operations on the data, the cloud needs to convert the AES ciphertext into an homomorphic ciphertext, and this is the goal of the trans-ciphering phase: it locally re-encrypts each bit of the AES ciphertext, and runs the AES decryption homomorphically. This principle can be applied to any output SHE scheme with message space $\mathcal{M}$, as long as ciphertexts, plaintexts, keys, and the internal state of the input scheme can be viewed as words in $\mathcal{M}^*$, and its decryption algorithm is parallelizable, for instance ECB, CBC or CTR modes. Homomorphic evaluation of the AES circuit has already been intensively studied, and for instance, [31] takes a particular care optimizing the running time of the algorithm[7].

This whole construction has the huge drawback, that the secret key of the first scheme becomes a secret data encrypted with the secret key of the output HE scheme. This allows to apply the attack presented in previous section, with the difference that, targeting the secret key reveals much more than just the 128 AES key bits. It allows the attacker to decrypt the whole input data. The main security properties of symmetric ciphers guarantees that any modification of a single key bit of a secret key renders the whole data gibberish, and will certainly trigger the expected reaction from the client. In a practical attack scenario, if the input data is AES-encrypted with a 128-bit AES key, the attacker may prefer to perform the attack on first 80 key bits, which only triggers about 40 negative feedbacks from the client. Then, the attacker may simply brute-force the remaining 48 bits off-line and decrypt all the client's data.

[7] Recent work from [10], show that different schemes (some stream ciphers for instance) can be more "FHE friendly" than AES for trans-ciphering.

4.2 Bootstrapping

The bootstrapping idea is to proxy re-encrypt data from a SHE scheme to another one. In this case, the noise of the output ciphertext only depends on the (fixed) noise of the bootstrapping key, and its (fixed) expansion during the homomorphic evaluation of the decryption function. Most importantly, it does not depend on the noise of the input ciphertext, which may reach the maximal level. Bootstrapping has been viewed as a way of bounding the noise growth, or of "refreshing" the ciphertexts after each (or after some) elementary homomorphic operation. The main problem, which was solved by Gentry in 2009, was to ensure that the parameter sizes of the output SHE scheme were not too large compared to the input SHE scheme. If possible, even take the second SHE scheme equal to the first one. To do so, the input SHE schemes must have a very efficient decryption procedure. For instance, integer schemes based on the approx-GCD, or on principal ideal lattices usually perform a single modular reduction plus an extraction of the least significant bit. Even then, the decryption circuit may still be too large. Gentry's first proposal, as instance, used a squashing technique, which added a lot of partial precomputed steps of the decryption algorithm as an additional hint, besides the encrypted private key, without weakening too much the security assumption. Recent schemes (like GSW [32],[1], or FHEW [21], [2], [15]) based on LWE use a quicker decryption algorithm, which only consists in a few modular additions plus an extraction of the most significant bit. This seems to correspond to the most complex decryption algorithm one can homomorphically evaluate within the scheme without increasing the parameter sizes, and without additional hints.

In all cases, the algorithm or circuit which is homomorphically evaluated during a bootstrapping procedure, or the refreshing procedure for schemes "without bootstrapping", is always equivalent to the official decryption function. Thus recovering the bootstrapping key via safe-error totally breaks the one-wayness of all schemes based on circular security. Schemes which do not rely on circular security but are still considered fully homomorphic provide instead, a long chain of distinct bootstrapping keys, where each one is encrypted with the next one. In this case, the safe error attack needs to be applied only on the last bootstrapping key of the chain, and once it is recovered, use it to decrypt the whole chain of keys. Again, this totally breaks the one-wayness.

4.3 Countermeasures?

Concerning the trans-ciphering attack, one could think that asking the client never to re-use the same AES key twice is enough. In particular, the encrypted AES key cannot be set as permanent key parameter, it must be generated and sent by the client over the network with every new data. In fact, the situation is much more complex than that. The actual requirement is that the cloud must never be allowed to re-use the same data twice. For instance, if the client used the AES trans-ciphering to transfer a whole database to the cloud, this database must only be used to answer a single client's query. Indeed, for the second query, the cloud may just silently re-run locally the trans-ciphering with another safe-error, and get another key bit, and so on until the 128-th query. Obviously, in most practical situations, the client will not be able to prevent the cloud from re-using the same data in many computations, so safe-errors should be considered as a huge threat against the trans-ciphering strategy.

The obfuscation counter-measure of the previous section may work for the trans- ciphering of LHE, but again, there exists no systematic obfuscation technique[8]. It must be done on a per-use basis, and it would considerably increase the trans-ciphering complexity. This means that the output LHE scheme must have extremely large parameters. In the bootstrappable FHE setting, obfuscation is simply impossible: indeed, within the noise overhead of practical bootstrapping, there is only enough room to express the decryption circuit as just a few additions.

Once again, the client should really stop trusting the cloud whenever he receives invalid data. Indeed, from what we saw with the AES attack, all his private data may already be lost before he even sees the 40-th error.

5 Conclusion and perspectives

The attacks we pointed out in this paper are very simple. Yet, they work even on perfect black-box primitives, and they have catastrophic consequences on data privacy if nothing is done at the system level to prevent them. Furthermore, in this report, we worked under the optimistic assumption that the requested computation had a unique solution, and that any error would be detected by the client's likelyhood test. Of course, the situation is much worse if the problem has multiple valid solutions, and each of those induces different visible reactions later in the process. This includes of course any white-box use of the cloud, where the final result is decrypted and published by the client. The cloud may hide some key-dependent ciphertexts inside the least significant bits of floating point statistics over encrypted medical data, or in the random bits of NTRU-like signatures. These attacks are even more dangerous than those we presented in the previous sections, because the client may hardly detect the attack, and in the mean time, the cloud provider is left with a universal decryption oracle.

Picturing the realization of a scenario similar to the one we described, we thought about some "more practical" countermeasures that can be added to the ones already proposed in Section 3 and 4.

Random computations. The first one will be to ask the cloud for random computations : for instance if we need to compute k signatures every day, we may ask the cloud for $2k$ signatures, of which k are random computations. If the cloud performs its attack on day 1, and the client detects an error, he will substitute the random computations of day 2 with a re-randomized ciphertext of day 1 and send to the cloud $2k$ computations as always. The cloud will not detect any

[8] At the time of writing, IO-obfuscation is neither practical, neither secure under any standard assumptions.

strange reaction from the client, and its attack will fail. But this countermeasure is probably really costly, especially in the economical model where the client pays per running time.

High entropy data. In the SHE or LHE settings, ensure that the data flow has high entropy at all times, and that changing one bit of a ciphertext is impossible, either because it exceeds the allowed noise bound (LHE), or because the operation itself cannot be expressed as a valid homomorphic operation (SHE). This implies that ciphertexts must encode multibit messages, and it may also require a secret key mode, where the attacker cannot easily obtain valid ciphertexts of arbitrary plaintexts. This countermeasure is certainly the most promising countermeasure for LHE schemes, but is not applicable to FHE because of its universality.

Verifiable computation. The last countermeasure is to require verifiable computation. The most straightforward solution is to de-randomize completely all cloud operations, and to ask two independent cloud services to do the exact same computations. If the two encrypted results are equal, then the client can hope that no attack has been performed. This requires of course that the two cloud providers do not collude. In a multi-user setting, this can be achieved by including incentive measures to the system, so that users are rewarded when they verify the computation (like in the bitcoin protocol). Again, this can be expensive in terms of costs and network bandwidth for the client, unless the function is simple enough and its total computation time is not too high (like in the e-voting protocol of [16]).
Another possibility is to force the cloud to produce (a zero-knowledge) proof of correct computation (against the attack on the algorithm), or a proof to demonstrate that all the ciphertexts used in the computation were effectively the one sent by the client (against the attack to the data). Recent works [13],[11],[12],[23], show how to practically construct homomorphic signatures, MACs and authenticators for both leveled and fully homomorphic encryption schemes. Roughly speaking, those tools allow a user (or multiple-users) to verify if the computations done by the cloud are correct and if the set of data taken in input is actually the good one. This indeed prevents safe errors. Intuitively, to verify the validity of a (deterministic) polynomial function against random errors, it suffices to check all the computations modulo a fixed λ-bits number N, where λ is the security parameter. A lot of additional work needs to be added to this simple idea to protect against malicious errors, see the above citations for more details. Overall, the verification process has the same number of steps than the full homomorphic computation, but if λ is smaller than the actual data-types of ciphertexts, the verification of the checksum may be faster that the entire computation[9].

[9] This is just an intuition, but this does not apply to [21] or [15] FHE schemes, which operate on small 32bits primitive types. Packing multiple 32-bits words together in these systems does not correspond to low degree polynomial functions anymore.

Therefore, all this techniques could be used in a larger scenario, where multiple users ask for small computations, fastly and constantly verified, as it happens in “blockchain-type” scenarios.

References

1. J. Alperin-Sheriff and C. Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology–CRYPTO 2014*, pages 297–314. Springer, 2014.
2. J.-F. Biasse and L. Ruiz. Fhew with efficient multibit bootstrapping. In *International Conference on Cryptology and Information Security in Latin America*, pages 119–135. Springer, 2015.
3. D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Advances in Cryptology—CRYPTO'98*, pages 1–12. Springer, 1998.
4. D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography Conference*, pages 325–341. Springer, 2005.
5. Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology–CRYPTO 2012*, pages 868–886. Springer, 2012.
6. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.
7. Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.
8. Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on Computing*, 43(2):831–871, 2014.
9. Z. Brakerski and V. Vaikuntanathan. Lattice-based fhe as secure as pke. In *Proceedings of the 5th conference on Innovations in theoretical computer science*, pages 1–12. ACM, 2014.
10. A. Canteaut, S. Carpov, C. Fontaine, T. Lepoint, M. Naya-Plasencia, P. Paillier, and R. Sirdey. Stream ciphers: A practical solution for efficient homomorphic-ciphertext compression. In *Fast Software Encryption 2016*. Springer Verlag, 2016.
11. D. Catalano and D. Fiore. Practical homomorphic macs for arithmetic circuits. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 336–352. Springer, 2013.
12. D. Catalano, D. Fiore, R. Gennaro, and L. Nizzardo. Generalizing homomorphic macs for arithmetic circuits. In *International Workshop on Public Key Cryptography*, pages 538–555. Springer, 2014.
13. D. Catalano, D. Fiore, and B. Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In *International Cryptology Conference*, pages 371–389. Springer, 2014.
14. M. Chenal and Q. Tang. On key recovery attacks against existing somewhat homomorphic encryption schemes. In *Progress in Cryptology-LATINCRYPT 2014*, pages 239–258. Springer, 2014.
15. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, pages 3–33. Springer, 2016.

16. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. A homomorphic lwe based e-voting scheme. In *International Workshop on Post-Quantum Cryptography*, pages 245–265. Springer, 2016.
17. J.-S. Coron, T. Lepoint, and M. Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *Public-Key Cryptography–PKC 2014*, pages 311–328. Springer, 2014.
18. J.-S. Coron, A. Mandal, D. Naccache, and M. Tibouchi. Fully homomorphic encryption over the integers with shorter public keys. In *Advances in Cryptology–CRYPTO 2011*, pages 487–504. Springer, 2011.
19. J.-S. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2012*, pages 446–464. Springer, 2012.
20. A. Das, S. Dutta, and A. Adhikari. Indistinguishability against chosen ciphertext verification attack revisited: The complete picture. In *Provable Security*, pages 104–120. Springer, 2013.
21. L. Ducas and D. Micciancio. Fhew: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015*, pages 617–640. Springer, 2015.
22. T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Workshop on the Theory and Application of Cryptographic Techniques*, pages 10–18. Springer, 1984.
23. D. Fiore, A. Mitrokotsa, L. Nizzardo, and E. Pagnin. Multi-key homomorphic authenticators. In *Advances in Cryptology–ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part II*, pages 499–530. Springer, 2016.
24. C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.
25. C. Gentry. Computing arbitrary functions of encrypted data. *Communications of the ACM*, 53(3):97–105, 2010.
26. C. Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.
27. C. Gentry and S. Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 107–109. IEEE, 2011.
28. C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011*, pages 129–148. Springer, 2011.
29. C. Gentry, S. Halevi, and N. P. Smart. Better bootstrapping in fully homomorphic encryption. In *Public Key Cryptography–PKC 2012*, pages 1–16. Springer, 2012.
30. C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Advances in Cryptology–EUROCRYPT 2012*, pages 465–482. Springer, 2012.
31. C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the aes circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.
32. C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.
33. C. Hall, I. Goldberg, and B. Schneier. Reaction attacks against several public-key cryptosystem. In *Information and Communication Security*, pages 2–12. Springer, 1999.

34. 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.
35. D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International Journal of Information Security*, 1(1):36–63, 2001.
36. D. W. Kravitz. Digital signature algorithm, July 27 1993. US Patent 5,231,668.
37. 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.
38. M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124. ACM, 2011.
39. P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.
40. R. L. Rivest, L. Adleman, and M. L. Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.
41. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
42. N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography–PKC 2010*, pages 420–443. Springer, 2010.
43. D. Stehlé and R. Steinfeld. Faster fully homomorphic encryption. In *Advances in Cryptology-ASIACRYPT 2010*, pages 377–394. Springer, 2010.
44. M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.
45. S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on computers*, 49(9):967–970, 2000.
46. 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.
47. Z. Zhang, T. Plantard, and W. Susilo. On the cca-1 security of somewhat homomorphic encryption over the integers. In *International Conference on Information Security Practice and Experience*, pages 353–368. Springer, 2012.