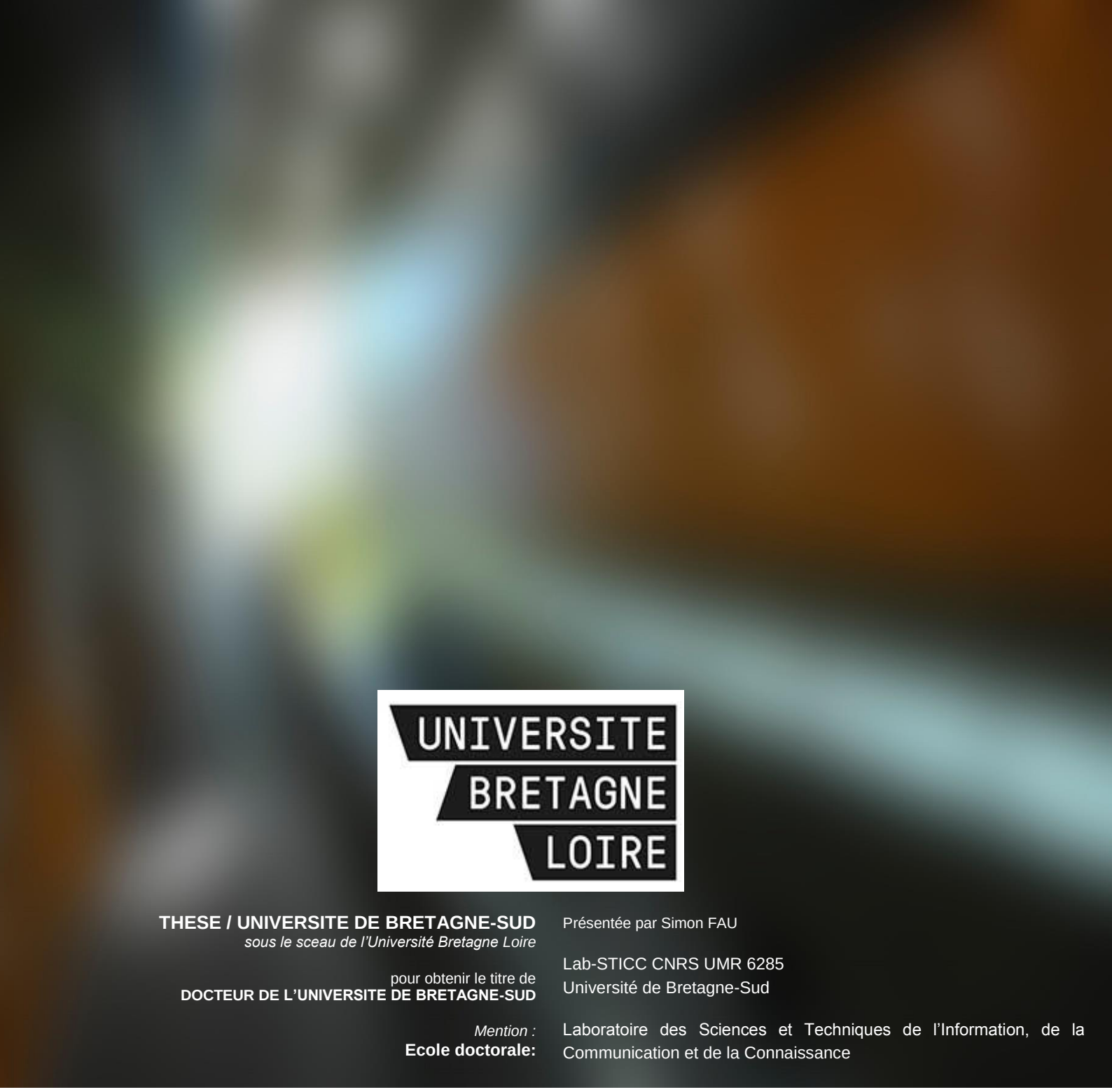
Cryptocomputing systems, compilation and runtime

Simon Fau



THESE / UNIVERSITE DE BRETAGNE-SUD
sous le sceau de l'Université Bretagne Loire

pour obtenir le titre de
DOCTEUR DE L'UNIVERSITE DE BRETAGNE-SUD

Mention :
Ecole doctorale:

Présentée par Simon FAU

Lab-STICC CNRS UMR 6285
Université de Bretagne-Sud

Laboratoire des Sciences et Techniques de l'Information, de la Communication et de la Connaissance

Systèmes de cryptocalculs, compilation et support d'exécution

Thèse soutenue le 22 mars 2016

devant le jury composé de :

M. Carlos AGUILAR-MELCHOR, Maitre de Conférences-HDR, IRIT

M. Albert COHEN, Directeur de Recherche INRIA, ENS

Mme. Caroline FONTAINE, Chargée de Recherche CNRS-HDR, TB

M. Guy GOGNIAT, Professeur des Universités, UBS

M. Philippe GABORIT, Professeur des Universités, Université de Limoges

M. Renaud SIRDEY, Directeur de Recherche CEA, LIST

Contents

Introduction

Background Since the introduction of the notion of Privacy Homomorphism by Rivest et al. [RSA78] in the late seventies, the design of efficient and secure encryption schemes allowing to perform general computations in the encrypted domain has been one of the holy grails of the cryptographic community, with applications in many domains. Despite numerous partial answers and unsuccessful attempts, the problem of designing such an obviously useful primitive has remained open until the theoretical breakthrough of C. Gentry [Gen09b, Gen09a] in the late 2000s, with the construction of the first *Fully Homomorphic Encryption* (FHE) scheme.

Interest in FHE schemes has grown in the past few years along with the widespread adoption of the cloud computing model for more and more critical applications. Indeed, when end users want to preserve the privacy of the data they outsource, they need to encrypt it using a cryptographic scheme, losing in the process the ability to do any other thing with the said data than simply retrieving the whole. In such cases, the possibility to perform computation directly on encrypted data seems like a great solution. As a straightforward example, an end user might want to preserve the confidentiality of his e-mails while still being able to set up filters or to perform searches. This leads to a need for encryption techniques that must be compliant with the storage and processing of outsourced encrypted data in the cloud, private information retrieval, (private) search on or analysis of encrypted data, etc.

Before going deeper in the subject, it is important to notice that for security reasons such encryption schemes are necessarily *probabilistic*. This means that for a given encryption key each *plaintext* can be encrypted in several different ciphertexts. This implies that the set of possible ciphertexts is significantly larger than the set of possible plaintexts. In other words, this implies that the ciphertexts are longer than the plaintexts. For a given probabilistic encryption scheme, the ratio between these two lengths is called the *expansion* of the scheme. Of course, designers try to propose schemes with the smallest possible expansion, for a given security level. We will see that expansion is huge for FHE schemes and this leads to a high overhead when computing in the encrypted domain using FHE. However, we will also see that various strategies can be set up to mitigate these disadvantages. In particular, instead of performing in the encrypted

domain exactly the same operations we would have performed in the clear, we have to consider the encrypted domain with its specitificity, like the fact that a multiplication is greatly more costly than an addition.

Besides, since 2009, a lot of publications provided variants and improvements. In particular, several so-called *somewhat FHE* cryptosystems have been proposed, which allow any number of additions but a bounded number of multiplications [AMGH10, GHV10a]. These schemes are really interesting as they are less complex than the fully homomorphic ones and are able to process a number of multiplications that is sufficient for most applications. Hence, we consider them today as the most promising schemes for practical applications.

Contributions When this thesis started in October 2011, while several theoretical papers dealt with Fully Homomorphic Encryption, only two articles presenting an attempt at implementing FHE had been published: [SV10], [GH11]. In the first one, FHE was not achieved since the paramaters necessary to make the somewhat homomorphic encryption scheme fully homomorphic were computationally out of reach. The second one achieved FHE but with much difficulty. At the same time, new encryption schemes had appeared since 2008 and they seemed far more efficient although no implementation had been published.

Our approach was not to focus on the design on new FHE primitives, as many other cryptography researchers already did, but rather to identify where FHE could be used in computer science and to build an experimental platform that would allow us to test real-life algorithm running on homomorphically-encrypted data. Since there was no opensource implementation at the time, we began by make our own implementation of a FHE scheme and chose what seemed to be to most efficient: BGV [BGV12]. However, we were aware that other implementations of FHE would come and probably be more efficient, so we build our platform in such a manner that the computations specific to the FHE scheme are independant from other computations. That way, when a new (opensource or internal) was available, we were able, with minimum work, to "plug it in" our existing platform and begin cryptocomputing with it.

Another important aspect of our approach was to rewrite the algorithms meant to be executed in the encrypted domain in order to dimension the homomorphic operations necessary. This choice had two main goals. The first one was to be able to parametrize the FHE scheme accordingly to the number of homomorphic additions and multiplications in the algorithm. The second one was to start bringing out some clues about what algorithms were "homomorphic-friendly", i.e. whose computational structure was propitious for homomorphic evaluation. It would also reveal to be a very useful tool to identify the computational hot spots of the homomorphic evaluation of algorithms. We will show in this thesis how to express algorithms seamlessly, regardless on whether they are executed in the plain (during testing) or in the encrypted domain (during opera-

tion). We also show how all classical integer manipulation operators (arithmetic, logical, bitshift, comparison, etc.) can be realized hermetically in the encrypted domain. More importantly, we also demonstrate how data dependent control-flow can be performed (at least to a non trivial extent) over such a system, in particular with respect to the conditional assignment operator as well as array assignment and dereferencing using encrypted indices, thus paving the way for a level of expressiveness that suits a wide spectrum of algorithms.

Once the platform was achieved, we first made experimental tests with the vectorial variant of BGV as well as the opensource implementation HCRYPT [PBS11]. We provided experimental results for a number of elementary but real algorithms (discriminant calculation, array summation, bubble sort, etc.). This work lead to an article in Signal Processing Magazine [AMFF+13]. Then, we made another round of tests with our implementation of the more efficient polynomial variant of BGV and more realistic security paramaters. This work lead to an article published in the proceedings of 3PGCIC [FSF+13]. Finally, we designed a method for performing private queries on an encrypted database using FHE. This work was continued by a fellow researcher in the laboratoire LaSTRE and a pattern has been filed for this concept. An article presenting these last results is also in process.

Overview The first part of this thesis is dedicated to the state of the art. We will first present homomorphic encryption schemes designed before 2008 and then move to the Fully Homomorphic encryption period. We will describe several schemes of interest for this thesis and discuss FHE implementations. Finally, we will present Yao's garbled circuits as they can solve similar problems as FHE and briefly talk about Functional Encryption (FE).

The second part of this thesis is for our contributions to the subject. We will begin by explaining how FHE can be useful in various scenarios and try to provide practical use cases that we identified during the thesis. In the next chapter, we will describe our approach to perform computations on encrypted data using FHE and explain how we were able to build on just the homomorphic addition and multiplication a platform for the execution in the encrypted domain of a wide range of algorithms. We will then detail our solution for performing private queries on an encrypted database using homomorphic encryption. In a final chapter, we will present our experimental results from the beginning to the end of the thesis.

Part I

State of the art and background

Overview

In the first chapter, we will review some of the main encryption schemes providing homomorphic properties from before 2008. Indeed, from 1978 to 2008, several *homomorphic encryption* schemes have been published, *e.g.* the famous Paillier's scheme and its derivatives, which are able to process encrypted data but with only one kind of operator (additions or multiplications) at a time [FG07].

In the second chapter, we will remind the reader of the main definitions useful to classify the homomorphic encryption schemes designed after 2008, give details about the cryptographic operations of these schemes and introduce briefly the mathematical problems that ensure their security. We will also describe briefly some FHE schemes that we find most interesting and finally discuss the main implementations of FHE that have been done since 2008.

The final chapted is dedicated to other tools that allow to compute over encrypted data or simply performing computations while keeping some information from the computer. After presenting the concept of multi-party computing, of which cryptocomputing is close, we will in particular describe Yao's garbled circuits. We will finally mention Functional Encryption since this subject is related to FHE and might bring additional features.

Chapter 1

Homomorphic encryption before FHE

1.1 Definitions

Let **E** be an asymetric encryption scheme equipped with a public key pubk and a private key privk.

E is said *homomorphic* (or sometimes *partially homomorphic*) if some algebraic operation performed on ciphertexts translates in a (possibly different) algebraic operation performed on plaintexts.

Example: **E** = RSA (without padding). For $m_1, m_2 \in (0, n)$,

$$\mathbf{E}(m_1.m_2) = m_1^e.m_2^e \bmod n = (m_1 m_2)^e \bmod n = E(m_1).E(m_2)$$

E is said *fully homomorphic* if it is homomorphic for any number of additions and multiplications, without requiring the use of the private key (no decryption needed).

Equivalently, saying **E** is *fully homomorphic* means that for any polynomial P, any $k \in \mathbb{N}$ and any plaintexts $m_1, \ldots, m_k$ in a ring $\mathbb{R}$:

$$P(m_1, \ldots, m_k) = \mathtt{dec}_{\text{privk}}(P(\mathbf{E}_{\text{pubk}}(m_1), \ldots, \mathbf{E}_{\text{pubk}}(m_k))).$$

Note: in this equality, P denotes both the polynomial over the plaintext space **R** and the corresponding polynomial over the ciphertext space equipped with the homomorphic operations of **E**.

Semantic security As we said earlier, all (good) homomorphic encryption schemes have to be probabilistic. Indeed, the plaintext space is usually very small (it is often $\{0, 1\}$) and it will produce a lot of ciphertexts. Therefore, if we do not

want an adversary to distinguish a ciphertext of 0 from a ciphertext of 1, it cannot be deterministic. *Semantic security* was introduced in [GM82], at the same time as probabilistic encryption, in order to define what could be a strong security level, unavailable without probabilistic encryption. Roughly, a probabilistic encryption is *semantically secure* if the knowledge of a ciphertext does not provide any useful information on the plaintext to some hypothetical adversary having only a reasonably restricted computational power. More formally, for any function f and any plaintext m, and with only polynomial resources (*i.e.* with algorithms which time/space complexities vary as a polynomial function of the size of the inputs), the probability to guess $f(m)$ (knowing f but not m) does not increase if the adversary knows a ciphertext corresponding to m. This might be thought of as a kind of perfect secrecy in the case when we only have polynomial resources.

Together with this strong requirement, the notion of *polynomial security* has been defined: the adversary chooses two plaintexts, and we choose secretly at random one plaintext and provide to the adversary a corresponding ciphertext. The adversary, still with polynomial resources, must guess which plaintext we chose. If the best he can do is to achieve a probability $1/2 + \varepsilon$ of success, the encryption is said to be *polynomially secure*. Polynomial security is now known as the *indistinguishability of encryptions* following the terminology and definitions of Goldreich [Gol93].

Quite amazingly, Goldwasser and Micali proved the equivalence between polynomial security and semantic security [GM82]; Goldreich extended these notions [Gol93] preserving the equivalence. With this equivalence, it is easy to state that a deterministic asymmetric encryption scheme cannot be semantically secure since it cannot be indistinguishable: the adversary knows the encryption function and, thus, can compute the single ciphertext corresponding to each plaintext.

We will now briefly describe some well known homomorphic encryption schemes anterior to 2008. Although not being fully homomorphic, they are interesting for applications that use linear operators.

1.2 Goldwasser-Micali

Alice computes a (public,private) key: she first chooses $n = pq$, p and q being large prime numbers, and g a quadratic non-residue modulo n whose Jacobi symbol is 1; her public key is composed of n and g, and her private key is the factorization of n.

The goal achieved by this scheme is that anyone can send a message to Alice.

Encryption To encrypt a bit b, Bob picks at random an integer $r \in \mathbf{Z}_n^*$, and computes $c = g^b r^2 \mod n$ (remark that c is a quadratic residue if and only if $b = 0$).

Decryption To get back to the plaintext, Alice has to determine if c is a quadratic residue or not. To do so, she uses the property that the Jacobi symbol $\left(\frac{c}{p}\right)$ is equal to $(-1)^b$. Note: the scheme encrypts 1 bit of information, while its output is usually 1024 bits long!

Homomorphic operations For $m_1, m_2 \in \{0, 1\}$, $c_1 = g^{m_1} {r_1}^2 \mod n$ and $c_2 = g^{m_2} {r_2}^2 \mod n$, so $c_1.c_2 = g^{m_1+m_2} r_1.{r_2}^2 \mod n$.
The sum $m_1 + m_2$ can therefore be retrieved by decrypting $c_1.c_2$.

Security and efficiency This scheme is the first one that was proved semantically secure against a passive adversary (under a computational assumption)[1].

The encryption is simple but the decryption step is done in $\mathcal{O}(\ell(p)^2)$. Unfortunately, this scheme presents a strong drawback since its input consists of a single bit. First, this implies that encrypting k bits leads to a cost of $\mathcal{O}(k \cdot \ell(p)^2)$. This is not very efficient even if it is considered practical. The second consequence concerns the expansion: a single bit of plaintext is encrypted by an integer modulo n, that is, $\ell(n)$ bits. Thus, the expansion is really huge.

1.3 Paillier

Alice computes a (public,private) key: she first chooses an integer $n = pq$, p and q being two large prime numbers and n satisfying $\gcd(n, \phi(n)) = 1$, and considers the group $G = \mathbf{Z}_{n^2}^*$ of order k. She also considers $g \in G$ of order n. Her public key is composed of n and g, and here private key consists in the factors of n.

As before, the goal is that anyone can send a message to Alice.

Encryption To encrypt a message $m \in \mathbf{Z}_n$, Bob picks at random an integer $r \in \mathbf{Z}_n^*$, and computes $c = g^m r^n \mod n^2$.

[1] An adversary is said passive if he can only eavesdrop the communications and cannot interact with the parties engaged in the protocol.

Decryption To get back to the plaintext, Alice computes the discrete logarithm of $c^{\lambda(n)} \mod n^2$, obtaining $m\lambda(n) \in \mathbf{Z}_n$, where $\lambda(n)$ denotes the Carmichael function. Now, since $\gcd(\lambda(n), n) = 1$, Alice easily computes $\lambda(n)^{-1} \mod n$ and gets m.

Homomorphic operations For $m_1, m_2 \in \{0, 1\}$, the ciphertext $c_1.c_2$ decrypts as $m_1 + m_2$.

Security and efficiency This scheme is semantically secure and also resistant to chosen plaintext attack (IND-CPA).

In Paillier's scheme, the ciphertext expansion is decreased to only 2, which means that a ciphertext's size is twice the plaintext's size. The encryption cost is not too high and the decryption needs one exponentiation modulo n^2 to the power $\lambda(n)$, and a multiplication modulo n. Paillier showed in his paper how to manage decryption efficiently through the Chinese Remainder Theorem. With smaller expansion and lower cost compared with previous schemes, Pailler's is really attractive.

1.4 El Gamal

Alice generates a (public, private) key: she first chooses a large prime integer p, a generating element g of the cyclic group $\mathbf{Z}_p^*$, and considers $q = p - 1$, the order of the group; building her public key, she picks at random $a \in \mathbf{Z}_q$ and computes $y_A = g^a$ in $\mathbf{Z}_p^*$, her public key being then (g, q, y_A); her private key is a.

Encryption Anyone can send an encrypted message to Alice. To send an encrypted version of the message m to Alice, Bob picks at random $k \in \mathbf{Z}_q$, computes $(c_1, c_2) = (g^k, my_A^k)$ in $\mathbf{Z}_p^*$.

Decryption To get back to the plaintext, Alice computes $c_2(c_1^a)^{-1}$ in $\mathbf{Z}_p^*$, which is precisely equal to m.

Homomorphic Operations For two ciphertexts c_1, c_2 encrypting two messages m_1, m_2 and g_1, g_2 two generating elements, $y_A = g^a$ where a is the secret key and r_1, r_2 two random exponents,
$c_1 \cdot c_2 = E(m_1) \cdot E(m_2) = (g^{r_1}, m_1 \cdot h^{r_1})(g^{r_2}, m2 \cdot h^{r_2}) = (g^{r_1+r_2}, (m1 \cdot m2)h^{r_1+r_2}) = E(m1 \cdot m2)$

Security The security of this scheme is related to the Diffie-Hellman problem: if we can solve it, then we can break ElGamal encryption. It is not known whether the two problems are equivalent or not. This scheme is IND-CPA.

1.5 Additional multiplication

In 2014, Catalano and Fiore found [CF14] a way to transform linearly homomorphic encryption schemes like the ones discussed in this chapter to make them able to evaluate degree-2 polynomials. This means that they can do a homomorphic multiplication with their modified encryption schemes as opposed to only homomorphic additions in the original schemes. However, this transformation leads to some drawbacks. In particular, there is a significant ciphertext expansion after the multiplication and unfortunately the following additions have the same cost than the multiplication.

Chapter 2

Fully Homomorphic Encryption

The development of fully homomorphic encryption (FHE) led to new notions we did not introduce yet. *Homomorphic* is very general while *fully homomorphic* is very difficult to achieve. In between, a lot of encryption schemes introduced starting from 2008 were not *fully homomorphic* but had strong properties that could be enough for some applications and therefore it did not seem appropriate to just call them *homomorphic*. Along the articles on homomorphic encryption, two main notions appeared:

2.1 Definitions

E is said *leveled fully homomorphic* if it is fully homomorphic only for a bounded number of additions and/or multiplications.

More formally, for fixed values $n, k \in \mathbb{N}$ and any k-variable polynomial P of degree n, if we can define private and public keys for **E** so that:

$$P_(m_1, \ldots, m_k) = \mathtt{dec}_{\text{privk}}(P_(\mathbf{E}_{\text{pubk}}(m_1), \ldots, \mathbf{E}_{\text{pubk}}(m_k))).$$

Note 1: As previously, P denotes both the polynomial over the plaintext space with natural additions and multiplications and the polynomial over the ciphertext space equipped with the homomorphic operations of **E**.
Note 2: We want to stress that *leveled fully homomorphic* and *fully homomorphic* are not equivalent. Indeed, the first implies that you fix a degree for the polynomial and generate adequate keys in order to handle a bounded number of homomorphic operations. On the contrary, the latter allows to generate keys for **E** and to perform any number of homomorphic operations.

An encryption scheme **E** is said *somewhat homomorphic* if it is fully homomorphic only for low degree polynomials.

This notion is close to *leveled fully homomorphic*, but it has not the same goals. A somewhat homomorphic scheme is viewed as a step in the design of a

fully homomorphic encryption (FHE) scheme. Indeed, once the somewhat homomorphic encryption (SHE) scheme **E** can handle its own decryption function, i.e. the decryption function (seen as a polynomial) has a degree low enough for **E**, it can be transformed into a FHE scheme through the bootstrapping step. This step will be explained further with the design of Gentry's first scheme [Gen09b].

2.2 The first fully homomorphic encryption (FHE) scheme

This encryption scheme has been introduced in 2009 by Gentry in his PhD thesis [Gen09a], with partial results being published previously in [Gen09b]. It is the first to be fully homomorphic and still proven secure. Indeed, other schemes with both homomorphic addition and multiplication were proposed before, but they all have been broken since.

This scheme was the object in 2010 of an implementation [GH11] that will be discussed in Section 2.6.

Overview

Let R be a ring fixed with respect to a security parameter λ. Let $I \subset R$ be an ideal and B_I a base for I (seen as a vectorial space). Let us first assume that for $t \in R$ and B_K a base for an ideal $K \subset R$, the set $t + K$ has a "unique representative" with respect to the base B_K and that it is easily "distinguishible". We note that unique $t \bmod B_K$. In the same spirit, we note $R \bmod B_K$ the set of all representatives of $r + K$ with respect for B_K.

Key generation: We pick J an ideal so that I and J are relatively prime (i.e. $I + J = R$). Then we generate two bases for J: B_J^{pk}, which will be the public key and B_J^{sk}, which will be the private key.

Encryption: For a plaintext $m \in P \subset R \bmod B_I$, we randomly pick c' in the ideal $m + I$. The ciphertext encrypting m is then $c \leftarrow c' \bmod B_J^{pk}$.

Decryption: For a ciphertext c and the associated private key sk, we have $m \leftarrow (c \bmod B_J^{sk}) \bmod B_I$

Homomorphic operations: Remember that we assumed that for any ideal K and $t \in R$, $t + K$ has a unique representative in B_K which is easily distinguishible. The ideals have the property to be stable for addition and multiplication, which are the homomorphic properties wanted. However, when adding and multiplying ciphertexts, these unique representatives become harder and harder to distinguish. This phenomenon is called *noise growth*.

- addition of c_i and c_j: $c_i + c_j \bmod B_J^{pk}$
- multiplication of c_i and c_j: $c_i \times c_j \bmod B_J^{pk}$

In practice, the ring is $R = \mathbb{Z}/f(x)$, where $f \in \mathbb{Z}[X]$ a polynomial of degree n of the form $X^n + 1$.

Bootstrapping As described, the scheme is not yet *fully homomorphic*, but only *somewhat homomorphic*. Indeed, as additions and multiplications are performed on the ciphertexts, their "noise" grows, which makes them ultimately undecryptable.

The idea behind bootstrapping is to "refresh" ciphertexts, i.e. to get rid of their noise (at least as much as possible), without decrypting them (otherwise it would require the use of the private key). This process is achieved by building, from a "noisy" ciphertext, another ciphertext encrypting the same plaintext but which noise is much smaller.

Let (sk_1, pk_1) and (sk_2, pk_2) be two pairs of private and public keys. Let c be a ciphertext encrypting a plaintext m under the first private key: $c = E(m, pk_1)$. The bootstrapping process is as follows:

$$c = E(m, pk_1) \rightarrow cc = E(c, pk_2) \rightarrow Dec(cc, \bar{sk_1}) \rightarrow c' = E(m, pk_2)$$

where $\bar{sk_1} = E(sk_1, pk_2)$. c, which is an encryption under sk_1 is then encrypted with pk_2 into a double-encrypted ciphertext cc. Then, the decryption function of the scheme is applied to cc using $\bar{sk_1}$, which is sk_1 encrypted with pk_2. Thanks to the homomorphic properties of the scheme, the result is c', which is now only encrypted with pk_2. The ciphertext went to a couple of keys to another without being in the clear in the process. This operation can therefore be performed by a non-trusted party, and does not require the use of sk_1 but rather the use of an encryption of sk_1 under $sk - 2$.

The trick of bootstrapping is to consider the decryption function just as any function we would like to perform in the encrypted domain. That way, we can decrypt (some part of the encryption) while remaining in the encrypted domain (for the other part of the encryption). However, to be able to decrypt homomorphically, we have to make sure that the original encryption scheme can handle enough homomorphic operations to do the decryption function. When this property is verified, the encryption scheme is said to be *bootstrappable*.

An additional security property is *key-dependant messages security*. If the ciphertext $E_{pk}(sk)$ does not allow to retrieve sk, for any pair of keys (pk, sk), the encryption scheme E is then said *key-dependant messages secure*. With that property, we can use bootstrapping without changing keys and the scheme can be made *fully homomorphic*.

Security The semantic security of this encryption scheme relies on the Ideal Coset Problem, which is defined as follows:

Ideal Coset Problem Let R be a ring and B_I a base for an ideal $I \subset R$ of R. Let `IdealGen` be an algorithm that generates the bases B_J^{pk} and B_J^{sk} and `Sample` an algorithm that outputs random elements of R. Let $b \in \{0, 1\}$ and (B_J^{sk}, B_J^{pk}) bases generated with `IdealGen`. If $b = 0$, we pick $r \leftarrow Sample(R)$ and $t \leftarrow r \bmod B_J^{pk}$. If $b = 1$, we pick t uniformly in $R \bmod B_J^{pk}$.

The problem is to find b knowing (t, B_J^{pk}).

The hardness of the Ideal Coset Problem depends on the algorithm `Sample` and on the distribution over R it provides (the more random is `Sample` the better).

The bootstrapping technique relies on the Sparse Subset-Sum Problem (SSSP), which is defined as follows:

Sparse Subset-Sum Problem Let γ_{set} and γ_{sub} be parameters dependant of λ. Let q be a positive integer and $b \in \{0, 1\}$. If $b = 0$, we generate a set τ of $\gamma_{set}(n)$ integers $a_1, ..., a_{\gamma_{set}(n)}$ in $[-q/2, q/2]$, picked randomly and uniformly in the interval, unless there is a subset $S \subseteq \{1, ..., \gamma_{set}(n)\}$ of cardinal $\gamma_{sub}(n)$ so that $\Sigma_{i \in S} a_i = 0 \bmod q$. If $b = 1$, we generate τ without the latter condition.

The problem is to find b knowing τ.

Parameters In [SS10], Stelhé et al. analysed more precisely the requirements for the SSSP and concluded to the following (reduced) parameters sizes:

- size of the private key: $\theta(\lambda^{1,5})$
- size of the public key: $\tilde{\mathcal{O}}(\lambda^{3,5})$
- runtime for a `Recrypt` operation: $\tilde{\mathcal{O}}(\lambda^{3,5})$

2.3 vDGHV: FHE over the integers

This encryption scheme has been presented in June 2010 by van Dijk et al. in [vDGHV10]. This scheme works on integers and is therefore much easier to understand than the previous FHE scheme of Gentry.

Overview We describe here the encryption scheme in its simplest version, which does not include bootstrapping. Indeed, as seen previously in Gentry's FHE scheme, before using bootstrapping, the first scheme built is only *somewhat homomorphic*. To become *fully homomorphic*, we have to be able to perform the decryption function with as little homomorphic operations as needed for the *somewhat homomorphic* scheme to handle it. To do that, the decryption function of the *somewhat homomorphic* scheme needs to be squashed so that its multiplicative depth is low enough to be homomorphically computed on data encrypted with that same *somewhat homomorphic* scheme. When the cost of decryption is reduced enough so that the *somewhat homomorphic* encryption can compute the decryption function homomorphically, bootstrapping can be achieved. The following description concerns the *somewhat homomorphic* encryption scheme, noted E:

- γ: size of the integers in the public key
- η: size of the private key
- ρ: gap between the integers' size and the closest multiples of the private key
- τ: number of integers in the public key

Here is the distribution that provides the integers for the public key:
$D_{\gamma,\rho}(p) = \{x = pq + r \ , \ \text{ou} \ \ q \in \mathbb{Z} \cap [0, 2^\gamma/p) \ \ \text{et} \ \ r \in \mathbb{Z} \cap (-2^\rho, 2^\rho)\}$

Key generation: The private key is an odd integer p of size η picked in $(2\mathbb{Z}+1) \cap [2^{\eta-1}, 2^\eta)$.

For the public key, we first compute τ integers such that $x_i = pq_i + r_i$, where $q_i \in \mathbb{Z} \cap [0, 2^\gamma/p)$ and $r_i \in \mathbb{Z} \cap (-2^\rho, 2^\rho)$ are randomly picked.

Consequently, the private key p is the approximate gcd of the public key integers x_i.

Let us rename the integers x_i so that x_0 is the greater. The generation is rebooted as long as we do not have these two conditions:
- x_0 is odd
- $x_0 \bmod p$ is even,

The public key is the set $\{x_i, 0 \leq i \leq \tau\}$.

Encryption: Let S be a randomly chosen subset of $\{1, 2, ..., \tau\}$ and r be randomly picked in $(-2^{2\rho}, 2^{2\rho})$. For a plaintext $m \in \{0, 1\}$, an encryption of mÂ is:
$c = m + 2r + 2\sum_{i \in S} x_i \bmod x_o$.

Decryption: The plaintext bit can be retrieved by computing $(c \bmod p) \bmod 2$. Indeed:

$$c = (m + 2r + 2\sum_{i \in S} pq_i + r_i) \bmod x_o = m + 2r + 2\sum_{i \in S} pq_i + r_i - (pq_0 + r_0).k \quad ,$$

where $k \in \mathbb{Z}$. In addition, $x_0 = pq_0 + r_0$, with r_0 being even and x_0 odd.

Therefore $c \bmod p = m + 2r + 2\sum_{i \in S} r_i - r_0.k$. As r_0 is even, the modulo 2 reduction gives m.

Homomorphic operations: Let C_E be a Boolean circuit with t inputs. The ciphertexts $c_1, ..., c_t$ are respectively encryptions of $m_1, ..., m_t$. We want to apply the (integer) additions and multiplications on $c_1, ..., c_t$ in the manner of the gates of C_E (integers addition for a XOR gate and multiplication for an AND gate). The result should be a ciphertext encrypting $C_E(m_1, ..., m_t)$.

However, the scheme described here is only somewhat homomorphic, so this homomorphic property will only be true if C_E's multiplicative depth is low enough. The scheme needs to be tweaked to be bootstrappable and thus fully homomorphic.

Security Its security is based on the "approximate-GCD Problem". The idea is to define the secret key as the appromixate gcd of all the public keys.

The proof of security of the DGHV scheme consists in a reduction to the approximate-GCD problem, which states that given the integers $x_0, x_1, ..., x_\tau$, all being multiples of p (a very big integer) and picked randomly, retrieving p is hard. More formally, the challenge is defined as follows:

(ρ, η, γ)**-approximate GCD problem:** Given a polynomial number of samples in $D_{\gamma,\rho}(p)$, p being an odd integer of size η, retrieve p.

For the fully homomorphic version of DGHV, which includes bootstrapping, the security also depends on the SSSP-problem defined previously.

Parameters size Let λ be the security parameter (a security of λ means that the best attack would take 2^λ elementary operations to break the encryption). The parameters size for vDGHV as they are fixed in [vDGHV10] are:

- private key size: $\eta = \tilde{\mathcal{O}}(\lambda^2)$.
- size of the integers in the public key: $\gamma = \tilde{\mathcal{O}}(\lambda^5)$.
- number of integers in the public key: $\tau = \tilde{\mathcal{O}}(\lambda^5)$.

The overall size of the public key is $\tilde{\mathcal{O}}(\lambda^{10})$.

The global overhead of the vDGHV scheme is $\tilde{\mathcal{O}}(\lambda^{7,5})$ per bit, which is far from being practical.

2.4 BGV: Leveled-FHE without bootstrapping

BGV is an asymmetric encryption scheme that encrypts bits. Like most (somewhat) FHE schemes, it is based on lattices. There are two versions of the cryptosystem: one dealing with integer vectors (the security of which is linked with the hardness of the Learning With Errors problem) and the other one with integer polynomials (the security of which is linked with the hardness of the Ring-Learning With Errors problem). In a few words, the Learning With Errors (resp. Ring-Learning With Errors) problem consists of distinguishing between a distribution of (a_i, b_i) sampled uniformly in $\mathbb{Z}_q^n \times \mathbb{Z}_q$ (resp. in the ring $\mathbb{R} = \mathbb{Z}_q^n/F(X)$) and a distribution of $(a_i, < a_i, s > +e_i)$, where a_i and s are sampled uniformly from $\mathbb{Z}_q^n$ (resp. $\mathbb{R}_q^n$) and e_i is sampled according to a Gaussian distribution. For more precisions on the (R)-LWE problem, we refer the reader to [Reg10]. In the sequel, we will focus on the polynomial version of the BGV encryption scheme, which seems more promising in terms of performances.

We consider the polynomial ring $\mathbb{R} = \mathbb{Z}[X]/F(X)$ where $F(X)$ is a cyclotomic polynomial of degree $d = 2^k$ and a chain of odd moduli $q_1 < ... < q_L$ and their corresponding subrings $\mathbb{R}_{q_i} = \mathbb{R}/q_i\mathbb{R}$ of polynomials of $\mathbb{R}$ with integers coefficients into the range $]-q_i/2, q_i/2]$. In practice, elements in $\mathbb{R}_{q_i}$ will be polynomials represented by the d-vector of their coefficients.

Basic encryption functions The private key sk is sampled in $\mathbb{R}$. A public key pk consists in the private key masked by a noise component: $pk = ask + 2e \in \mathbb{R}_{q_L}^N$, where $N = O(\log q_L)$, $a \in \mathbb{R}_{q_L}^N$ and the noise e is sampled from a "discrete" Gaussian distribution over $\mathbb{R}^N$ ("discrete" meaning here that we sample from a Gaussian distribution and round to the nearest integer). Here follows a set of black box descriptions of the main functions associated with the encryption scheme. We have decided not to include the exact algorithms to avoid drowning the important issues in technical descriptions. If interested, the reader can refer to [BGV12],[GHPS12] for a precise algorithmic description.

`Encrypt`**(Plaintext** m**, PublicKey** pk**): Ciphertext** c

The integers we manipulate need to be encrypted one bit at a time. For $m \in \{0, 1\}$, the resulting ciphertext c is a pair of two elements in $\mathbb{R}_{q_L}$ derived from the plaintext m, the public key pk and a random seed (since it is a probabilistic scheme). In the following, a ciphertext can be transformed into a pair of two elements in any subring $\mathbb{R}_{q_i}$. In our implementation, each ciphertext carries its level, i.e. the information that indicates in which subring it lies.

`Decrypt`**(Ciphertext** c**, PrivateKey** sk**): Plaintext** m

The decryption function is a simple dot product between the ciphertext $c \in \mathbb{R}_{q_i}$ and the private key followed by a modular reduction into the range $]-q_i/2, q_i/2]$ and finally a parity test to retrieve the plaintext m. As we mentioned in previous examples, the noise must be under a certain level for the decryption

to be correct.

Level shifting operations `Rescale`**(Cipertext** c**): Ciphertext** c'

The function transforms the ciphertext $c \in \mathbb{R}^2_{q_i}$ into a ciphertext $c' \in \mathbb{R}^2_{q_{i-1}}$. The resulting ciphertext has a reduced noise.

`SwitchKey`**(Augmented Cipertext** c**): Ciphertext** c'

The tensored product of two ciphertexts $c_1 \otimes c_2$ results in an "augmented ciphertext" $c \in \mathbb{R}^3_{q_i}$. To retrieve a regular ciphertext in $\mathbb{R}^2_{q_i}$, we essentially multiply c by a public matrix (a different one for each level $1 < i < L$). Then we call the `Rescale` function to get $c' \in \mathbb{R}^2_{q_{i-1}}$ (with low noise).

Homomorphic operations `Add`**(Ciphertext** c_1**, Ciphertext** c_2**): Ciphertext** c_{sum}

For two ciphertexts c_1, c_2 where $c_1 \in \mathbb{R}^2_{q_{i_1}}$ and $c_2 \in \mathbb{R}^2_{q_{i_2}}$, we follow these steps:

if $i_1 \neq i_2$ *(for example* $i_1 < i_2$*)* **then**
  do $c'_2 \leftarrow$ `Rescale`(c_2) $i_2 - i_1$ times; *(at this point we have* c_1, c_2 *at the same level* i_1*)*
end
do $c_{sum} \leftarrow c_1 + c'_2$; *(simply by adding the coefficients of the polynomials modulo* q_{i_1}*)*

`Mul`**(Ciphertext** c_1**, Ciphertext** c_2**): Ciphertext** c_{mul}

For two ciphertexts $c_1 \in \mathbb{R}^2_{q_{i_1}}$ and $c_2 \in \mathbb{R}^2_{q_{i_2}}$, we follow the steps:

if $i_1 \neq i_2$ *(for example* $i_1 < i_2$*)* **then**
  call $c'_2 \leftarrow$ `Rescale`(c_2) $i_2 - i_1$ times; *(at this point we have* c_1, c_2 *at the same level* i_1*)*
end
do $c_3 \leftarrow c_1 \otimes c'_2$; *(*$c_3 \in \mathbb{R}^3_{q_{i_1}}$*)*
do $c_{mul} \leftarrow$ `SwitchKey`(c_3); *(*$c_{mul} \in \mathbb{R}^2_{q_{i_1-1}}$*)*

The tensored product applied on c_1 and c_2 consists in adding and multiplying polynomials of $\mathbb{R}_{q_{i_1}}$, which can be very expensive as we will see.

Parameters The size of the ciphertexts and therefore the cost of additions and multiplications on those ciphertexts, depends on the size of the $\{q_i\}_i$ and on the size of the ring $\mathbb{R}$ (i.e. the size of d or n). To give an idea of the cost of these operations, we want to stress that each bit is encrypted by a pair of polynomials that can be of degree $d > 10000$ and have coefficients of size > 200 bits. For security and noise management reasons, these parameters grow as the number of `Mul` increases (as shown in [BGV12]). More precisely, the key value to dimension

the cryptosystem is the multiplicative depth.[1]

We can also already point out that the order in which we perform the homomorphic operations may have an impact on the number of times we have to call the `Rescale` and `SwitchKey` functions, therefore on the number of levels (multiplicative depth) we need.

2.5 Scale-invariant FHE schemes

Scale-invariance As seen before in the leveled-FHE schemes, the ciphertexts are polynomials with coefficients modulo some q. q is a large integer that changes anytime a "modulus-switching" operation is performed (in order to decrease the level of noise of a ciphertext).

This technique works well for reducing noise but in practice, the process is costly since we need a switching key for each level of the scheme. Since the public keys are quite large, the need to store and then to compute with L large public keys (to achieve a multiplicative depth of $L-1$) was very detrimental when implementing the scheme. To resolve this issue, Brakerski introduced at Crypto 2012 a new tensor product technique that didn't need to use different moduli and therefore different switching-keys.

These scale-invariant FHE schemes are the latest to be introduced and they offer the best performances so far. Their computational overhead and practical parameters will be discussed in Section 2.6, specifically dedicated to implementations.

2.5.1 Brakerski's scale-invariant FHE scheme

This encryption scheme is very close to BGV and the (original) variant based on the LWE problem has been introduced at CRYPTO 2012 [Bra12]. The main difference lies in the structure of the dot product of a ciphertext and its corresponding private key. In BGV, for a message m, a private key s, a modulus q and a ciphertext c, the decryption works as follows:

$$< c, s >= m + 2e \bmod q,$$

which gives m by reducing modulo 2.

In Brakerski's scale-invariant scheme, the same dot product gives:

$$< c, s >= \lfloor q/2 \rfloor \cdot m + e \bmod q,$$

which provides the same result when applying the reduction modulo 2.

[1] In a Boolean circuit, the multiplicative depth is defined as the maximal number of multiplication gates on any path.

The result of this tweak is that the noise growth is much smaller when multiplying two ciphertexts. Consequently, the same modulus can be used throughout the homomorphic operations and the noise does not grow more than in BGV. This also means that the modulus switching operation needed in BGV is now skipped, making the homomorphic computations much faster.

2.5.2 FV

In [FV12], Fan and Vercauteren ported to the R-LWE problem the Brakerski's scale-invariant scheme mentioned previously. This means that the ciphertexts are elements of $\mathbb{R} = \mathbb{Z}_q^n/F(X)$ for some polynomial $F(x)$ (rather than elements of $\mathbb{Z}_q^n \times \mathbb{Z}_q$ in the previous scheme). The encryption, decryption and homomorphic operations are very similar to Brakerski's scale-invariant scheme so we will not detail them here.

In their paper, Fan and Vercauteren also gave concrete parameters for the somewhat homomorphic scheme and for the FHE scheme. In addition to its speed, that is why this scheme was chosen as the underlying homomorphic scheme for the most recent experimentations in the laboratoire LaSTRE. We will talk about the implementation and the performances of this scheme further in this thesis.

2.5.3 YASHE

This encryption scheme has been introduced in 2013 by Bos et al. in [BLLN13b] and is also scale-invariant. Like BGV polynomial variant, its security is based on the R-LWE problem.

Encryption For a modulus q, a plaintext modulus $1 < t < q$ and a public key pk in a polynomial ring $\mathbb{R}$, the ciphertext c is:

$$c = \lfloor q/t \rfloor \cdot [m]_t + e + pk \cdot s,$$

where s and e are error components.

Decryption The decryption works as in BGV: the dot product of a ciphertext c and the private key is reduced modulo q and then modulo t, to retrieve the message m.

FV and YASHE seem to be the most promising FHE schemes at the time of writing and they are compared by Lepoint and Naehrig in [LN14].

2.5.4 Scale-invariant FHE scheme over the integers

This scheme is an adaption of the vDGHV FHE scheme over the integers, in a way that makes it be scale-invariant. It was introduced by Coron et al in [CLT14] in 2014. We remind that in vDGHV a ciphertext c is built as follows:

$$c = m + 2r + q \cdot p,$$

where q works as the public key and p as the private key and r is a noise component.

In the adapted scheme, the same ciphertext c is now:

$$c = r + (m_1 + 2r') \cdot (p-1)/2 + q \cdot p^2,$$

where r and r' are noise components.

2.6 Implementations of Homomorphic Encryption

The implementation of the first FHE scheme has been done by Gentry and Halevi in [GH11]. Prior to that, an implementation had been done by Smart and Vercauteren and detailed in [SV10]. Their FHE scheme is very similar to Gentry's first FHE scheme, with reduced key size and ciphertext expansion. However, their implementation did not achieve fully homomorphic encryption. In 2011, a little after Gentry and Halevi's implementation, Perl et al. [PBS11] presented another implementation of the Smart and Vercauteren's scheme of [SV10] that achieved fully homomorphic encryption.

2.6.1 Implementation by Gentry and Halevi

This implementation is the first to achieve fully homomorphic encryption in practice.

In the large setting, the public key size was 2.3 GB (2.2 hours to generate the keys) and the bootstrapping operation took around 30 minutes on a large memory machine. The security parameter was fixed at $\lambda = 72$.

Although the overhead of this implementation makes it highly prohibitive in the real world, it was the first time fully homomorphic encryption was achieved for real.

2.6.2 Implementation by Smart and Vercauteren

Although the encryption scheme described is asymptotically fully homomorphic, the implementation made by Smart and Vercauteren could not achieve fully homomorphic encryption.

Indeed, as previously seen, the construction of a FHE scheme requires to use a somewhat FHE scheme first, and use bootstrapping to make it fully homomorphic. The somewhat FHE scheme has to be "homomorphic enough" to handle its own decryption function in the encrypted domain in order to achieve bootstrapping.

In the case of Smart and Vercauteren's scheme, the parameters of the somewhat FHE scheme have to be too big to make it fully homomorphic in practice. They showed that the parameter N, which is basically the degree of their polynomial ciphertexts, has to be greater than 2^{27} to achieve bootstrapping, and they were only able to generate keys up to $N = 2^{11}$.

2.6.3 Implementation by Perl et al. of Smart and Vercauteren's scheme

This implementation is called HCRYPT and is particularly interesting since it can be downloaded and used freely at http://hcrypt.com. As a matter of fact, we were able to use HCRYPT as the underlying encryption scheme in our experimental platform. The results obtained with HCRYPT will be discussed in Chapter 7.

Perl et al. implemented the FHE scheme of Smart and Vercauteren but they managed to perform bootstrapping. Indeed, they decreased the multiplicative depth of the decryption circuit so that the somewhat homomorphic scheme could apply the decryption in the ciphertext space. However, the recrypt operation took several seconds to complete for decent security parameters.

2.6.4 Implementations of vDGHV

Two implementations of this FHE scheme have been published: [CNT12], [CLT13], with the latter focusing on evaluating AES homomorphically.

An implementation of the symmetric somewhat homomorphic version of vDGHV was also produced by the author. It will be briefly discussed in Chapter 7.

2.6.5 Implementations of BGV

An implementation of the vectorial variant and an implementation of the polynomial variant of BGV have been realized during this thesis and are used in our experimental platform. These will be thoroughly discussed in Chapter 7.

2.6.6 AES-oriented implementations

Several implementations were specifically aimed at running the AES algorithm on already (homomorphically) encrypted data. The interest of this approach is to use AES-encrypted data for the communications, while still being able to perform

homomorphic operations (modulo a partial decryption). Given a FHE scheme noted **E** and an AES secret key **sk**, the process works as follow:

- Alice sends to Bob a homomorphic encryption of her AES secret key:

$$\mathbf{E}(\mathbf{sk})$$

- Alice sends to Bob the AES-encrypted ciphertexts of her confidential data $m_1, ..., m_n$:

$$AES_{\mathbf{sk}}(m_i)$$

- Bob computes the homomorphic encryptions of the AES-encrypted ciphtertexts sent by Alice:

$$\mathbf{E}(AES_{\mathbf{sk}}(m_i))$$

  Since **E** *is an asymmetric encryption scheme, Bob knows its public key and can therefore produce ciphertexts*

- Thanks to the homomorphic properties of **E**, Bob can decrypt the AES part of the ciphertexts using $\mathbf{E}(\mathbf{sk})$:

$$Dec_{\mathbf{E(sk)}}(\mathbf{E}(AES_{\mathbf{sk}}(m_i))) = \mathbf{E}(m_i)$$

  Note that the last equality is only true if **E** *is "homomorphic enough" to handle the AES algorithm.*

- Bob has now homomorphic encryptions of Alice's data, so he can perform the homomorphic operations required by Alice.

In this approach, all the encrypted communications are AES ciphertexts, except for the homomorphically encrypted AES key. This is the main advantage of the process, since the AES has a much smaller ciphertext expansion than any FHE scheme (by construction).

In [GHPS12], Gentry et al. implemented the polynomial variant of BGV with single key and made several optimisations aimed at evaluating the AES function in the encrypted domain. The AES-128 algorithm, viewed as a Boolean circuit, has a multiplicative depth of 40. Thus, they needed to take parameters that would guarantee that 40 homomorphic multiplications (and some additions as well) could be performed successively and that the result was still decryptable.

They ran their evaluation of the AES function on a machine with 256 GB of RAM and the ten rounds of AES took over 36 hours. However, in their configuration, a ciphertext could carry many plaintexts at once, so they could

process 54 AES blocks at the same time, which makes the amortized rate 40 minutes per AES block. This performance has to be taken cautiously since they used a machine with such a large memory. We have seen that public keys, switch keys and ciphertexts are very large, so the 256 GB of RAM really makes a big difference in the computations.

An implementation of AES-128 was made in the laboratoire LaSTRE and executed in the encrypted domain using a platform improved from the one built during this thesis. The underlying FHE scheme used is FV and the running time is consequently smaller: 18 minutes for the AES-128 evaluation, with RAM memory usage under 40 GB. More details about this implementation are provided in [CDS15].

2.6.7 Other main implementations

An implementation of [Bra12] in its polynomial variant (based on the R-LWE problem), similarly to [FV12], was created by a fellow researcher in the lab during this thesis.

In [BLLN13a] Bos et al. propose a new FHE scheme based on [SS11] with improved security and give some implementation results with practical parameters.

Finally, in [LN14], Lepoint and Naehrig implement both the aforementioned schemes using the arithmetic library FLINT and were able to compare to two FHE schemes in the same conditions.

2.6.8 Discussing FHE overhead

First part, theoretical point of view => per-bit overhead in terms of the security parameyter λ, decrease of that overhead not only by the design of new schemes based on different mathematical problems but also by more thorough security analysis that leads to lower parameters.

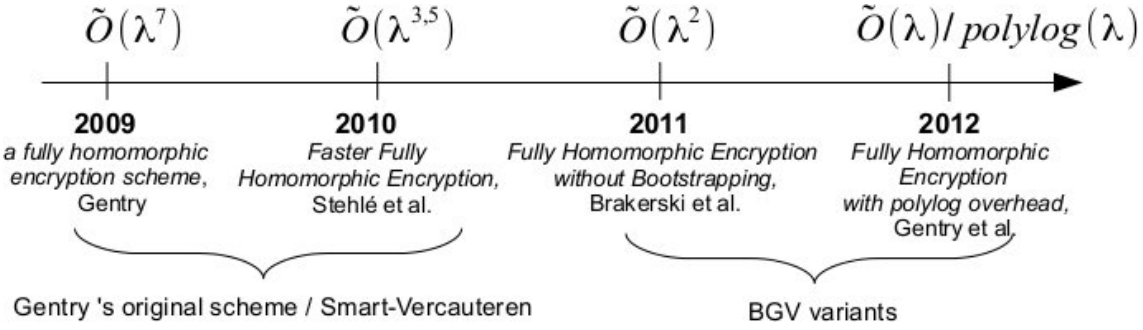


Figure 2.1: The fast paced decrease of FHE schemes' computational overheads

Second part, practical results => per-bit overhead calculated for each operation (key generation, encryption, homomorphic addition and multiplication, decryption), overhead varies according to each FHE scheme, tricks when implementing, starting to identify what scheme might be better with some use case but not with another.

Chapter 3

Other tools of cryptocomputing

Although homomorphic encryption is now a field of research on its own, it can also be seen as a special case of multi-party computation. This latter field has for goal to create protocols that enable two or more parties to jointly compute a function of their inputs, while ensuring that each party's inputs remain private.

A good and simple example of multi-party computation is the "millionaire problem":

Alice and Bob want to know who is the richer, but none of them wants the other to know exactly how rich the other is.

A generalisation of this problem is that Alice knows x and Bob knows y, and they want to compute $f(x, y)$ without Bob knowing x and Alice knowing y.

Homomorphic encryption is a powerful solution for multi-party computation since Alice and Bob would simply have to encrypt their inputs. Then, one of them would perform the computations over encrypted ciphertexts and finally both would engage in a distributed decryption as described in [MSS11]. However, this might not be the fastest and simplest solution since homomorphic encryption is a more powerful primitive than needed for multi-party computation. Indeed, homomorphic encryption does not require any communications between the parties to enable computations in the encrypted domain, while most applications of multi-party computation allow several rounds of communications between the parties.

We will describe here a few tools enabling multi-party computation in use cases where homomorphic encryption is probably not the most efficient solution.

3.1 Yao's garbled circuits

Garbled circuits were introduced by Yao in 1989 and have been the subject of many articles since then. The general issue addressed is these papers is multi-party computation. Typically, they can resolve the "millionaire problem" or enable

Yao's garbled circuits can achieve this confidentiality and give a correct result for this problem in the semi-honest model[1] .

3.1.1 Protocol overview

Let $x_1, ..., x_n \in \{0,1\}$ denote Alice's data and $y_1, ..., y_2 \in \{0,1\}$ denote the ones of Bob. Given $f : \{0,1\}^{2n} \to \{0,1\}^*$, the goal of this protocol is to compute $f(x,y)$ while preserving the confidentiality of x and y. One of the two parties, for example Alice, will do the computations and perform the operations of the garbled circuit, which will be built by the other party, here Bob.

Here are the steps defined in the protocol:

1. Bob builds a "garbled circuit", which is a "garbled" version of the Boolean circuit that computes the function f. It is composed of tables and keys, which are necessary to the computation of the garbled circuit.
2. Alice and Bob first engage in an *oblivious transfer*. In this step, Bob sends to Alice the keys associated with $x = (x_1, ..., x_n)$ and $y = (y_1, ..., y_n)$. During the transfer, neither Alice nor Bob learn anything about the other one's data. At the end of the oblivious transfer, Alice has the entry keys $k_1, ..., k_n, k_{n+1}, ..., k_{2n}$ associated with x and y. These keys allow Alice to decrypt the tables of the first gates of the garbled circuit.
3. Alice executes the garbled circuit and computes $f(x,y)$.
4. Alice sends (optionally) $f(x,y)$ to Bob.

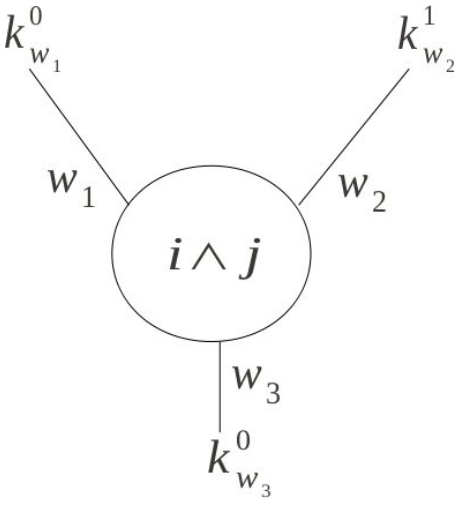
Step 1: building the circuit

Building a garbled circuit consists in creating for each wire w in the circuit, a set of two keys k_w^0 and k_w^1 representing respectively the binary values 0 and 1. For each two-fan gate, a table of 4 inputs is created, one for each possible input of the gate: $T_{(0,0)}, T_{(0,1)}, T_{(1,0)}, T_{(1,1)}$. To compute these tables, we perform a double encryption using the two keys associated to the two inputs of the gate and we encrypt the key associated to the gate's output.

Let E be the encryption scheme used. For w_1 and w_2 two input wires and w_3 the output wire, we have:
$T_{(i,j)} = E(k_{w_1}^i, E(k_{w_2}^j, k_{w_3}^{g(i,j)})$,
where $g(i,j)$ is the output bit of the gate when the input bits are i and j.

For example, the garbled circuit for an AND-gate will be:

[1] In the semi-honest model, we assume that both parties respect the protocol, but they try to find confidential information by keeping all the computations performed during the protocol and exploiting these data.



$$T_{(0,1)} = E(k^0_{w_1}, E(k^1_{w_2}, k^{0\wedge 1}_{w_3})$$

Using the two keys obtained previously, one can decrypt the table of the current gate: in the example, the input bits are 0 and 1, represented by the keys $k^0_{w_1}$ et $k^1_{w_2}$. Using these keys, one can decrypt $k^{0\wedge 1}_{w_3} = k^0_{w_3}$. As soon as the inputs $T_{(0,0)}, T_{(0,1)}, T_{(1,0)}$ and $T_{(1,1)}$ are computed, they are randomly permuted so that Alice cannot know which couple (i, j) is associated to the input she can decrypt.

Step 2: Oblivisous Transfer

For each wire w, Bob creates the keys k^0_w and k^1_w, which represent the input bits 0 and 1 for this wire. He wants to send to Alice the key associated with Alice's datum $\sigma \in \{0, 1\}$ without revealing to Alice the other key and without Bob learning the value of σ.

1. Bob picks randomly a one-way permutation f with a trapdoor t and sends f to Alice.

2. Alice picks a random a_σ and computes $b_\sigma = f(a_\sigma)$ as well as another random $b_{1-\sigma}$ in the domain of f. She then sends (b_0, b_1) to Bob.

3. Bob uses t to compute $a_i = f^{-1}(b_i)$, then $c_i = P(a_i) + k^i_b$, where P is a "hard predicate"[1] for f. Bob sends (c_1, c_2) to Alice.

4. Alice computes $k^\sigma_w = c_\sigma + B(a_\sigma)$.

[1] A hard predicate $P(x)$ is easily computable using x but hardly computable using only $f(x)$.

At the end of these steps, Alice owns the key associated with her input bit σ. These steps are repeated for all of Alice's input bits. Bob then completes by sending all the keys associated with his input bits.

Finally, Alice has the keys $k_1, ..., k_n, k_{n+1}, ..., k_{2n}$ and will be able to start the computation of the garbled circuit.

Step 3: Circuit computation

Let us consider a fan-in-2 gate G at the top of the circuit. Since Alice owns a key for each input wire of the circuit, so she has two keys for the two inputs of G and can decrypt one of the four table entries for the gate G. With this double decryption, Alice gets the output key corresponding to the gate computation.

To ensure that Alice can decrypt correctly one (and only one) of the table entries for each gate, the encryption scheme used must have the following properties:

- For two plaintexts x and y, a ciphertext encrypting x and a ciphertext encrypting y are indistinguishible in a polynomial time.
- *elusive range*: the probability that a ciphertext under the key k_1 is in the domain (i.e. ciphertext space) of a different key k_2 is negligible.
- *efficiently verifiable range*: given a ciphertext c, we can easily verify whether c is in the domain of a key k or not.

These properties allow Alice to make sure that she gets an output key correctly and efficiently.

3.1.2 Security

The garbled circuits are proven to be secure in the semi-honest model. However, they can only be used once. Indeed, if a garbled circuit was used a second time with different data, Alice (the one who did not build the garbled circuit) would not have only one key per wire, and could possibly decrypt several entries in the same table. This would give her information about the type of gate it is, and thus on the function that the garbled circuit computes.

For example, let us consider a gate G with two input wires w_1 and w_2 and an output wire w_3. During the first execution of the garbled circuit, Alice learns that the two bits s_1 and s_2 give in output a bit s_3 (although Alice does not know the values of the bits). During a second execution, she could learn that the bits $\tilde{s}_1$ and s_2 give the same output bit s_3. As the executions go by, Alice could find out the tables of the gates in the garbled circuit.

Alice could also learn information about the input data of Bob.

In order to resolve this issue, Gentry et al. proposed a rerandomization of the garbled circuits in [GHV10b], which would make further executions of the garbled circuits possible, while being significantly faster than building another garbled circuit.

3.1.3 Performances

In [MNPS04], Malkhi et al. implemented Yao's garbled circuits in their system called Fairplay, which has its own procedural programmation language and its own compiler.

One of the functions tested using garbled circuits was about searching in a database:

Bob owns a database of 16 entries, each one being composed of 24 bits of data and represented by a 6-bit key. Alice wants to retrieve confidentially one of the entries by giving its specific key.

The tests were run using both local and extend networks (LAN and WAN). The overall process of retrieving an entry took 0,49s with LAN and 3,38s with WAN. Here is the detail of how much time took each step of the protocol:

Network	Building the garbled circuit	Communication	Oblivious Transfer	Evaluation	Total
LAN	$40,4\%$	$2,8\%$	$54,1\%$	$2,7\%$	$0,49s$
WAN	$5,9\%$	$64,3\%$	$29,4\%$	$0,4\%$	$3,38s$

An implementation using garbled circuits has also been made (in Java) in our laboratory, with AES as the underlying encryption scheme. The goal is to enable keywords search in the chapter summaries of the children's book *Alice in Wonderland*.

Building the garbled circuits took about 7 seconds per chapter (an average chapter summary was about 120 words) and the computations of the garbled circuits about 5 seconds per chapter.

3.1.4 Rerandomization

The idea of rerandomization is to create a new garbled circuit from an old one, at a lesser cost. The new garbled circuit has to be unrecognizable for the party who executed the old one, and even for the party who created it (in the case of a rerandomization by a third party).

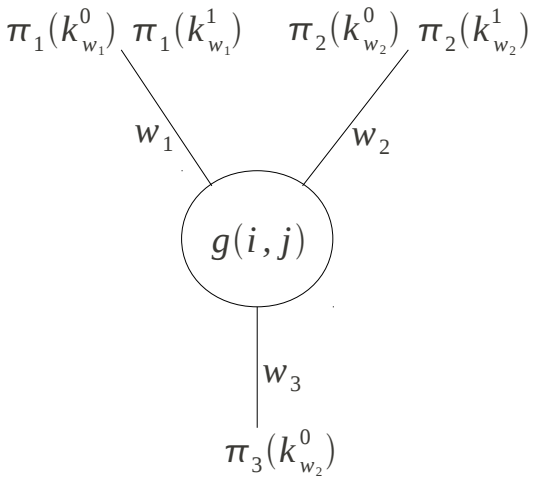
One solution for rerandomization is to perform permutations on the table entries and the garbled circuit's keys. A permutation has to be chosen for each wire and the same permutation applied to the keys associated with each wire. In

addition, the tables have to be modified so that the new keys allow to execute the new tables. For each gate g, the table entries would then become the following couples:

$$\left(E_{\pi_{w_1}(k^i_{w_1})}(\delta_{i,j}), E_{\pi_{w_2}(k^j_{w_2})}(\pi_{w_3}(k^{g(i,j)}_{w_3} \oplus \delta_{i,j})) \right)$$

For the table entries, which are encrypted, to pass on the permutations performed on the keys, the encryption method used must have the adequate homomorphic properties.

This is what a gate of the new garbled circuit would look like:



In [BHHO08], one of the encryption method used has the following property: one can, without decrypting nor knowing the secret key, transform $E_k(x)$ in a new ciphertext $E_{\pi_1(k)}(\pi_2(x))$, where π_1 and π_2 are permutations of $\{0,1\}^*$.

This encryption thus allows to transform the tables of the garbled circuit so that the keys k_i needed to compute it become $\pi_1(k_i)$. Therefore they can get a new garbled circuit without having to perform additional computations on the tables, which is even better.

3.2 Functional Encryption

One of the main disadvantages of homomorphic encryption is that the cryptocomputer, i.e. the party doing the homomorphic computations, is completely blind with respect to the encrypted data it processes. In the first place, this unawareness is a security requirement, but it would be very useful for some applications to be able to reveal to the cryptocomputer *some* information or even allow him to decrypt *some* ciphertexts.

Imagine for example the case of videosurveillance, where we would like to allow the detection of prohibited behavior or dangerous individuals, without allowing to observe everything that is filmed by the videosurveillance camera. The idea would be to encrypt the videos and design a private key that would only decrypt data of the type "dangerous individual detected" or "everything is ok". This is not possible with homomorphic encryption and even less with traditional cryptography.

The concept of *functional encryption* was introduced to try to resolve this issue. Basically, using a functional encryption scheme for some function f, an authority owns a *master secret key* and can generate a private key sk_f which allows its owner to compute $f(x)$ from an encryption of x and nothing else.

For the last 10 years or so, various encryption schemes have been proposed, among others are: *identity-based encryption* (IBE), *attribute-based encryption* (ABE) or *searchable encryption*. These schemes have different properties like allowing only to decrypting the messages intended to you, only decrypting if you meet some criteria or searching through encrypted data.

However, functional encryption remains theoretical at the time of writing, since all the existing functional encryption schemes carry prohibitevely large overheads.

Part II

Contributions: homomorphic encryption, from theory to practice

Overview

In Chapter 4, we list some of the possible applications of FHE and explain briefly how it could be done. We will see that this work is not trivial since it requires to dimension the computations to find where a homomorphic evaluation is realistic.

In Chapter 5, we present our experimental platform. From the implementation of bitwise logical operators in the encrypted domain to data-dependant control (to some extent), we explain in detail how we manage to perform computations on encrypted data in the most efficient way.

Finally, we present in Chapter 6 our solution for performing private queries on an encrypted database.

Chapter 4

Possible applications of FHE

In this chapter, we try to identify realistic scenarios where homomorphic can provide security and/or confidentiality. In some cases, it is about bringing additional confidentiality to new applications where confidentiality is not possible as of today. In other cases, it is about offering to outsource computations or data storage while guaranteeing the same security needs.

4.1 Computing on outsourced confidential data

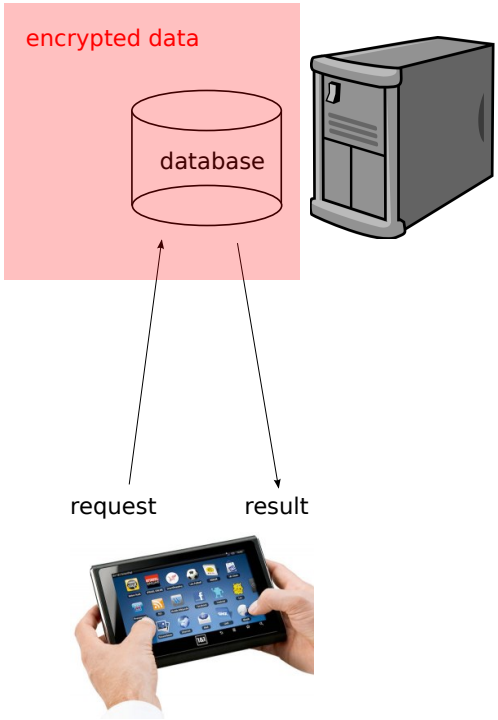


Figure 4.1: A need for preserving the privacy of outsourced data, while being able to proceed to some requests on them. A typical example is keyword searching or filtering on an outsourced encrypted email box.

4.1.1 Keyword searching

It is the simplest example we can think of. A text is encrypted and the result is uploaded to an untrusted server. We want to be able to test the presence of a word in that text, while keeping the text as well as the searched word secret. On the other hand, we are willing to reveal the number of matches in the text.

Several solutions exist to address this issue, like encrypted keywords or garbled circuits, but if we want to extend our use to the search of strings or even regular expression, most of the solutions will not work anymore. FHE can provide a generic solution for searching on encrypted data. However, FHE usually remains too heavy and faster answers have been proposed for many use cases such as conjunctive search and Boolean queries on symmetrically-encrypted data, as shown in [CJJ+13].

4.1.2 E-mail filters

Another simple application is the filtering of an outsourced encrypted email box. Being able to apply the spam filters or category labels depending on the sender while keeping the confidentiality of your emails is very interesting. Computationnaly, it is basically equivalent to a key-word search on encrypted data.

However, using only FHE, the client must decrypt all the results of the filtering on his side, and therefore the e-mail sorting remains on the client side. Indeed, when the server computes the filtering operations on encrypted e-mails, the results are encrypted as well. The only way to make that the server could sort the e-mails itself is by using Functional Encryption. As we mentioned in Section 3.2, FE allows the client to reveal, for some function f and encrypted data x, the (clear) result of $f(x)$. In that case, the server could compute the spam function and know the resulting Boolean. That way, the server could sort the e-mails as spam or non-spam for example and there would be no additional computation on the client side. We mention this as a possible application for FHE because, as shown in [GKP+13], a Functional Encryption scheme can be built on an existing FHE scheme.

4.1.3 Private queries on an encrypted database

An increasing number of services provided to Internet users involve cloud computing. In this model, the personal data of a client is often outsource in distant servers, which raises the issue of privacy. Still, using the cloud remains preferable when the device has limited computing capacities or when the service is only available by the cloud and cannot be offered inside the device for commercial reasons (such as industrial confidentiality) or security reasons. Considering the success of all sorts of connected devices with various computing capacities, securing the cloud becomes a more and more significant issue everyday.

Fortunately, a lot of tools are available in cryptography to achieve this goal. Lightweight cryptography, garbled circuits and homomorphic encryption are good candidates for various scenarios involving cloud computing. In Chapter 6, we propose our own solution for performing private queries on an encrypted database, based on FHE.

4.2 Performing outsourced computation on confidential data

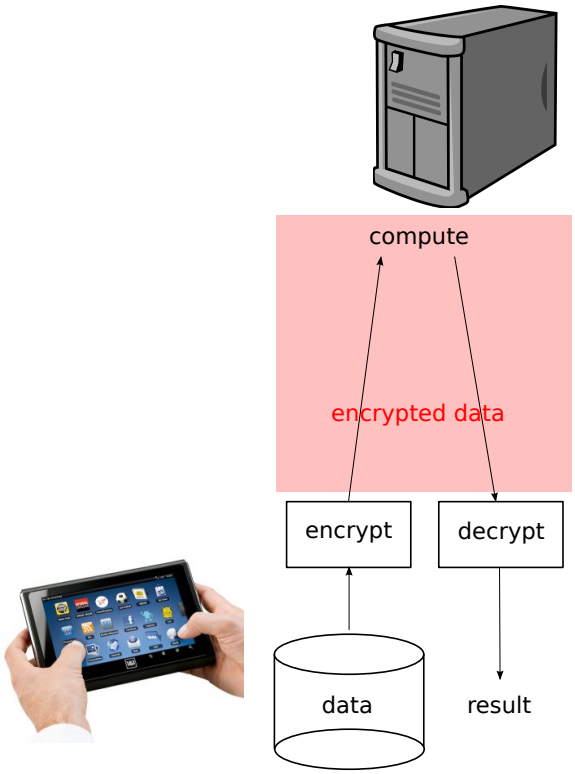


Figure 4.2: A need for processing encrypted data in a scenario of outsourced computation. For example, applying a one-shot enhancement algorithm on a private image or, more futuristically, performing automatic on-the-fly translation of private voice calls.

4.2.1 Medical data processing

The medical data of a patient are confidential. If the patient, the doctors laboratory analysts are all in the same place, there is no problem to ensure this confidentiality when performing diagnostic tests on the patient or monitoring his vitals.

However, there are more and more cases when these conditions are not verified: for example for an outpatient, or when a patient's samples have to be sent to a more technologically advanced facility to be analyzed. In these cases, homomorphic encryption can be a solution.

Figure 4.3 shows various diagnostic tests used to detect cardiac issues. A fictional yet realistic algorithm would be to compare the patient's blood results to the "high risk" values and determine if the patient is at risk.

Cardiology diagnostic tests

Test Name	Lower/normal risk	High risk	Cost $US (approx)
Total Cholesterol	<200 mg/dL	>240 mg/dL	
LDL-C	<100 mg/dL	>160 mg/dL	$150*
HDL-C	>60 mg/dL	<40 mg/dL	
Triglyceride	<150 mg/dL	>200 mg/dL	
Blood Pressure	<120/80 mmHg	>140/90 mmHg	
C-reactive protein	<1 mg/L	>3 mg/L	$20
Fibrinogen	<300 mg/dL	>460 mg/dL	$100
Homocysteine	<10 µmol/L	>14 µmol/L	$200
Fasting Insulin	<15 µIU/mL	>25 µIU/mL	$75
Ferritin	male 12–300 ng/mL female 12–150 ng/mL		$85
Lipoprotein(a) - Lp(a)	<14 mg/dL	>19 mg/dL	$75
Calcium Heart Scan	<100	>300	$250–600

(*) due to the high cost, LDL is usually calculated instead of being measured directly
source: Beyond Cholesterol, Julius Torelli MD, 2005 ISBN 0-312-34863-0

Figure 4.3: Simple threshold tests in cardiology diagnostic from Wikipedia.

A patient could sample his blood at home using a connected sensor and send his results to the hospital through a device capable of encrypting data. This device would have to possess a pre-established private key and the corresponding public key to encrypt and decrypt data using a homomorphic encryption scheme. The patient's blood results are thus homomorphically encrypted at home and sent to the hospital that computes a medical algorithm in the encrypted domain. It then sends the encrypted results for the patient to decrypt.

Of course, we are in that case talking about a more sophisticated algorithm than simple thresholds tests. It would be particularly interesting if the medical algorithm is private and thus the processing of the medical data cannot be done directly by the patient.

This example and its practical realization will be discussed further in Section 7.1.4.

4.3 Computing simultaneously on confidential local data and outsourced confidential data

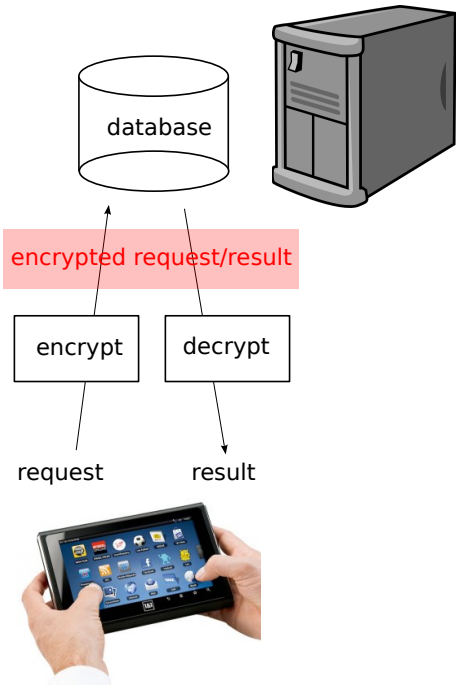


Figure 4.4: A need for performing private requests on public data. A typical example would be deep packet inspection without revealing e.g. the traced IP address.

4.3.1 Targeted advertising

Various services such as Gmail or Facebook are using the personal information of their users in order to target their advertising more accurately. This is the main (or only) way of earning money, so anyone who would like to use such tools has to be willing to give up completely on the confidentiality of everything he puts on the service.

However, thanks to homomorphic encryption, their personal data could still be used to profile the advertising accurately, thus ensuring the viability of the commercial service, while still guaranteeing data confidentiality. The overhead resulted from the homomorphic encryption has two main inconvenients: it will decrease the speed of the service and increase the amount of computations on

the server end. The first inconvenient is a price to pay for a user if he wants to protect the confidentiality of his data. The second inconvenient, on the other end, is the service provider's burden, and we cannot really imagine that Google or Facebook would be willing to waste a lot of computational power just for the sake of the user's privacy.

Anyway, the arrival of a competing tool offering basically the same services but adding the user's privacy may change this current state and force the providers to include data confidentiality. We can also imagine two options for the same service: a premium one, where the client pays for its privacy while enjoying the exact same features, and a regular one where all the client's data are transparent.

This solution could also allow said providers to go further in the ways of targeting advertising. If installed on a smartphone, the service could use information like the user's geographic position to target advertising, like indicating nearby restaurants. A lot of users are understandably reluctant to that kind of use of their geolocalisation information today, but a guarantee on the protection of their privacy might help them to accept these features.

4.3.2 Biometric authentication

The typical setting of this application is when a client owning a biometric passport wishes to authentify without revealing his biometric data. The devices performing the verification may be numerous (everywhere an authentication is needed) so if it just reads transparently the biometric data of any passport owner, it is a major confidentiality issue. A malicious user of a device could indeed gather a lot of valid and useful biometric data.

Here again, homomorphic encryption provides a solution, at least theoretically. The biometric data in the client passport can be encrypted using a FHE scheme as well as the data needed for authentication. Therefore, even if the device is attacked or use maliciously, all useful data is protected by the homomorphic encryption.

It is a realistic solution since the biometric data is usually lightweight and the authentication computations are not very costly. In [BCP13], Bringer et al. used homomorphic encryption (and garbled circuits as well in some settings) to perform face recognition or fingerprint authentication in a matter of seconds.

4.3.3 Cloud-based biochemical reactor control

Homomorphic encryption's security is mathematically proved so it can be used in applications where security is critical (assuming the parameters are correctly set).

In this scenario, a manufacturer is selling biochemical reactors to an operator who would like to use it. The reactor has various functioning modes and the operator may want to switch from one to another according to external factors

such as weather, as well as measurements collected in the reactor. The functioning mode of the reactor is to be updated every hour. Unfortunately, the algorithm computing what the next mode should be is the property of the manufacturer, who doesn't want to reveal it to its client because of its commercial value. On the other side, the operator wants to keep the data relative to its reactor confidential, as some of it might be sensitive: production capacity, efficiency, etc.

If neither the manufacturer nor the operator accept to reveal some of their secrets, there is apparently no way of making it work. This is where homomorphic encryption can solve the problem.

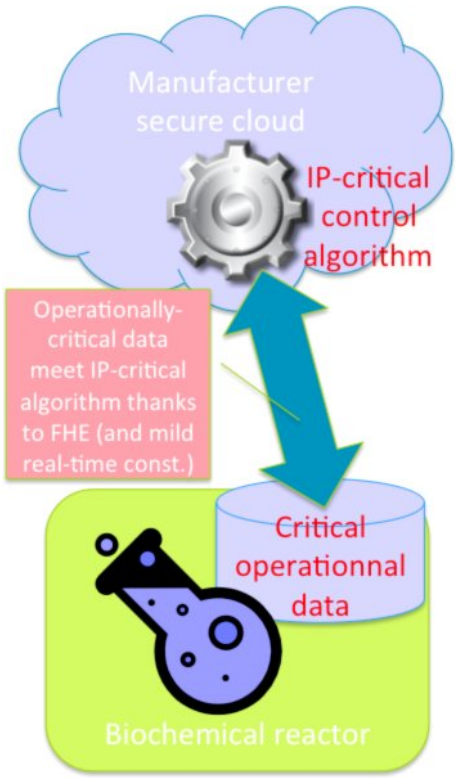


Figure 4.5: An outsourced control of a biochemical reactor.

Indeed, the operator can encrypt the reactor's sensitive data through homomorphic encryption and send it to the manufacturer who has the algorithm on its private servers. This way, the manufacturer can compute the next functioning mode without ever knowing the reactor's data.

The functioning mode consists in 8 bits (which represents a couple of MB when homomorphically encrypted) so it is not problematic to send such data every hour to the reactor. The data needed as input for the algorithm computing the next mode is about 10 Bytes and the time constraint is about 10 minutes. Since the manufacturer's algorithm is not heavy in terms of computation (quite similar to the medical algorithm aforementioned), homomorphic encryption can

provide a viable and secure solution in this scenario.

4.4 Signal processing

Processing signals in the encrypted domain is an important challenge. These last years, more and more researchers designed specially tailored solutions dedicated to many applications. Without being exhaustive, we can mention: privacy-enhanced face recognition [EFG+09], privacy-preserving electrocardiogram signal classification [BFL+11], privacy protection of biometric data [BBC+10, FSB10], buyer-seller protocols [KT05, PEL07, KLC+08], and zero-knowledge watermark detection [ARS05, PCC+06, TPPG06]. In parallel, other works developed some general tools for processing some particular operations on encrypted signals, that can be useful in many applications: *e.g.* Gram-Schmidt Orthogonalization [FB10], DCT computation [BPB09a], DFT computation [BPB09b]. Finally, we can mention general discussions on the processing of encrypted signals [EPK+07] and attempts to find adequate representations for such processing, as in [BPB10].

These publications rely on regular homomorphic encryption. Hence, when needed, computing over encrypted data functions involving both additions and multiplications is really tricky. It requires linearizing the computation in an *ad hoc* manner and using *multi-party computation* techniques. This demands the use of heavy protocols, designed precisely for each application. Moreover, these protocols need many interactions between the parties to do the job correctly. For more details on the issues of privacy in Signal Processing applications and on how homomorphic encryption can help to solve them, we refer the reader to [LEB].

With an accessible (somewhat) FHE scheme, it would be possible to compute polynomial functions directly, without linearization or multi-party computation techniques. Of course, the cost to pay would be directly related to the complexity of the (somewhat) FHE scheme used. But according to recent works, the complexity of such schemes is currently dropping down faster than expected, even one year ago. There is still lots of work to do to get a very efficient scheme, but each step forward makes real applications closer than before.

Chapter 5

Computing in the encrypted domain

(Somewhat) FHE schemes allow to evaluate any (bounded degree) polynomial from $\mathbb{Z}_2^n$ to $\mathbb{Z}_2$ or, equivalently, any Boolean circuit. Recall that a Boolean circuit consists in a directed acyclic graph where vertices are either inputs, outputs or operators (`and` or `xor`) and where edges represent data dependencies. In higher-level programming terms, working with (somewhat) FHE schemes restricts us to programs or algorithms having bounded input and a control flow that is independent of encrypted data. In particular, this *a priori* excludes (encrypted) data-dependent `if-then-else` statements as well as loop termination criteria. At first, this may seem highly restrictive. However, control depending on encrypted data can still be performed to some extent, as we shall see in this section.

5.1 Basic bitwise logical operators

Let us first see how a FHE scheme permits the implementation of pretty much any of the classical integer manipulation operators. Additions and multiplications can be implemented following textbook recipes for n-bit adders and multipliers (although choosing the most appropriate design for execution over a FHE scheme is not so straightforward). Because multiplications (`and`s) are particularly costly, the multiplier itself should be optimized when either both or one of the (encrypted) operands are Boolean, in which case there is only one layer of bit-level multiplication (`and`s), or when one of the operand is available in the clear, in which case the multiplication becomes a sequence of additions of shifted versions of the encrypted domain input.

Bitwise logical operators (`and`, `xor`, `or`, etc.) turn out to be easy to implement using the two basic cryptosystem operations.

Negation (minus) can be implemented using the textbook trick of 2-complementing: `xor`ing all "crypto-bits"|*cbits* in the sequel|with an encryption of 1, in order to

complement them, and adding an encryption of 1 (with carry propagation) to the result. This allows to implement a n-bit subtraction operator using an n-bit adder. Also, when subtraction is implemented that way, the most significant *cbit* provides the sign of the integer, a fact that can be known and used by the "cryptocomputer" despite the fact that it has no access to the effective value of that bit as it is itself locked in the encrypted domain.

5.1.1 Comparing encrypted integers

It is then also possible to perform comparisons hermetically in the encrypted domain. Although there are a number of ways to implement comparison operators we have designed our operators so as to avoid multiplications (`and`s) as much as possible. Our solution thus consists in starting from the `less than` operator which can be implemented by subtracting the two operands and then by producing a result which consists of $n-1$ leading encryptions of 0 followed by the most significant bit of the subtraction result i.e., the aforementioned sign *cbit* (which is in this case stored in the least significant bit). The `greater than` operator is performed similarly. Note that following the execution of such an operator, the "cryptocomputer" knows (legally) that there is only one bit of payload in the result and can exploit that fact in further calculations (most importantly in multiplier optimizations as already stated). The (Boolean) `not` operator can be obtained by `xor`ing the least significant bit with an encryption of 1. Having both the `less than`, `greater than` and `not` operator, the `equal to` operator can be performed as well (which allows to implement the δ function used in Equations (5.2) and (5.3) below) in a fashion which is suboptimal with respect to the number of gates but much less involved in terms of multiplications than more classical designs.

5.1.2 Bitshift operators

Lastly, left and right bitshift operators can also be obtained hermetically in the encrypted domain. The left bitshift operator requires copying the relevant rightmost *cbits* of its operand and then (right) padding with as many encryptions of 0 as required. The right bitshift operator, on the other hand, requires copying the relevant leftmost *cbits* of its operand and then (left) padding with as many copies of the most significant *cbit* (i.e., the sign *cbit*) of that operand which lives in the encrypted domain. Left and right rotations can also be implemented by moving *cbit* around.

5.2 Data-dependant control

Now that these classical operators are available, we can go back to the data-dependant control issue. Let us consider a selection operator $\texttt{select} : \mathbb{Z}_2 \times \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ such that

$$\texttt{select}(c, a, b) = \begin{cases} a & \text{if } c = 1 \\ b & \text{otherwise.} \end{cases}$$

Such an operator can then straightforwardly be rewritten as follows

$$\texttt{select}(c, a, b) = ca + (1 - c)b, \tag{5.1}$$

Provided the implementations of addition, multiplication and negation mentioned earlier in this section, Eq. (5.1) translates as

$$\texttt{select}(c, a, b) = ca \texttt{ xor } (\texttt{not } c)b.$$

5.2.1 A meaningful example: the bubble sort

As this construction allows to perform a conditional assignment operator, it enables the implementation of a wide range of algorithms. As an example, consider the following simple (although quite demonstrative) example of a bubble sort algorithm which may be expressed as follows in C-style programming languages:

```
void bsort(int *arr,int n)
{
 for(int i=0;i<n-1;i++)
 {
  for(int j=1;j<n-i;j++)
   if(arr[j-1]>arr[j])
   {
    int t=arr[j-1];
    arr[j-1]=arr[j];
    arr[j]=t;
   }
  }
}
```

Using the selection operator of Eq. (5.1), this algorithm can be rewritten in a suitable fashion for execution over a FHE scheme, that is, without requiring any access to the value of the test `arr[j-1]>arr[j]`:

```
void bsort(int *arr,int n)
{
```

```
 for(int i=0;i<n-1;i++)
 {
  for(int j=1;j<n-i;j++)
  {
   int gt=arr[j-1]>arr[j];
   int t=select(gt,arr[j-1],arr[j]);
   arr[j-1]=select(gt,arr[j],arr[j-1]);
   arr[j]=t;
  }
 }
}
```

Still, it should be emphasized that, expressed as above, the bubble sort algorithm always achieves its worst-case $O(n^2)$ complexity: this is a price to be paid unless one accepts leaking information about the sorted data.

5.2.2 Non linear operators

This bubble sort example is demonstrative and reveals that pretty complex algorithms can be realized over a FHE scheme. Recall furthermore that sorting is a naive algorithm for computing the median of a sample [Knu73], thus allowing to construct non linear DSP primitives such as a median filter. At that point, it should be clear that almost any non linear signal or image processing primitive (thresholding, mathematical morphology operator, etc.) can be performed. As a more advanced example, as long as one is able to homomorphically evaluate the objective function of an optimization problem, at least in theory, then a full blown simulated annealing algorithm, which is often used to solve inverse problems in both signal and image processing, can be performed homomorphically. The selection operator allows to keep track of the best solution encountered while executing the algorithm and also allows to perform the randomized temperature-driven acceptance rule for a new solution

$$\text{if } u \leq e^{-\frac{c(\omega')-c(\omega)}{T}} \text{ then } \omega = \omega'.$$

where u is chosen uniformly in $[0, 1]$, ω and ω' respectively denote the current and the candidate solution, $c(\omega)$ denotes the cost of solution ω and T denotes the temperature.

Now, if ω, ω', $c(\omega)$ and $c(\omega')$ need to remain private, we use encryptions of these values, noted $\overline{\omega}$, $\overline{\omega'}$, $\overline{c(\omega)}$ and $\overline{c(\omega')}$. In the encrypted domain, the condition above then translates to:

$$\texttt{enc}(u) \leq e^{-\frac{\overline{c(\omega')}-\overline{c(\omega)}}{\texttt{enc}(T)}},$$

where $\leq$ is performed homomorphically as described earlier. Let us call $\overline{\alpha}$ an encryption of the Boolean associated to this condition. Thanks to the selection operator, we can then perform (homomorphically):

$$\overline{\omega} = \texttt{select}(\overline{\alpha}, \overline{\omega'}, \overline{\omega}).$$

5.2.3 Array with encrypted indices

It turns out that array dereferencing and assignment with encrypted indices is also possible. Indeed,

$$t[i] = \sum_{j=1}^{n} \delta(i,j)t[j], \tag{5.2}$$

with $\delta(i,j) = 1$ if $i = j$ and 0 otherwise. Similarly, array assignment ($t[i] = v$) can be done by performing

$$t[j] = \delta(i,j)v \oplus (1 - \delta(i,j))t[j], \forall j. \tag{5.3}$$

Of course, both operations are done in $O(n)$ rather than $O(1)$ in the clear index case. It should also be emphasized that, as a result of an assigment, *all* the array entries change although all but one of them decrypts to the same value as before the assigment. Again, this is a price to pay for index privacy.

Some of the above operators involve inserting encryptions of 0 or creating multiple copies of certain *cbit* such as the sign *cbit* of a difference. Note that, due to the probabilistic nature of the FHE scheme underlying the calculation, the cryptocomputer loses track of these values as soon as they are involved in a further operation. For example, adding (`xor`ing) a *cbit*, say c_0, known to be an encryption of 1 (because the encryption has been performed by the cryptocomputer as part of the data it injects in the calculation) to another *cbit* of unknown value necessarily leads, by construction of the cryptosystem, to a result which has nothing to do with c_0 and, thus, which does not allow to (practically) infer any information about the value of the *cbit* of unknown value.

5.3 Expressing high level algorithms

Having defined integer manipulation operators, we are now in theory ready to express many high level algorithms in a natural fashion. This can easily be done using the operator overloading features of object-oriented programming languages such as C++, for example via a `CryptoBit` class provided with `+` and `*` operators and by using it to build a `CryptoInt` class provided with the operators specified in the previous section.

However, from a software engineering point of view, it is desirable to be able to do more and in particular to be able *from a single code* to perform the following tasks:

1. Test and debug of an algorithm in the clear domain (either at the integer level or at the bit level).

2. Characterize an algorithm to obtain dimensioning parameters for the underlying FHE scheme (e.g., the multiplicative depth of the algorithm) and predict performances.

3. Execute literally an algorithm in the encrypted domain.

4. Generate compilation data (e.g., the Boolean circuit topology) for further optimizations of the calculation and later executions on an ad hoc, non literal, execution support.

5.3.1 From the clear domain to the encrypted domain

Again, this can be achieved by using the type parameterization feature of object-programming languages (such as the so-called templates provided in the C++ language) by creating an `integer` class parameterized by both a `bit` type and a size. The `bit` type representing either clear bits (in which case the operators `+` and `*` are trivial), instrumented clear bits (see `ClearBit` below) or crypto bits (in which case the `+` and `*` operators are implemented with respect to the underlying FHE scheme). As an example, in this framework, the bubble sort code sample of the previous section simply becomes

```
template<typename integer>
void bsort(integer *arr,int n)
{
 for(int i=0;i<n-1;i++)
 {
  for(int j=1;j<n-i;j++)
  {
   integer gt=arr[j-1]>arr[j];
   integer t=select(gt,arr[j-1],arr[j]);
   arr[j-1]=select(gt,arr[j],arr[j-1]);
   arr[j]=t;
  }
 }
}
```

and this *unique* code is either invoked as

```
bsort<Integer<ClearBit,8> >(arr,n);
```

for execution in the clear in order to (e.g.) sort an array (of public size) of 8-bits integers or as

```
bsort<Integer<CryptoBit,8> >(arr,n);
```

in order to do the same thing in the encrypted domain (of course in that case `arr` contains 8-bits integers encrypted at the bit level with the underlying FHE scheme).

5.3.2 Revealing useful characteristics of an algorithm

Since, as already emphasized, we are dealing only with programs with a static control structure, any execution in the clear domain allows to infer the relevant characteristics of an algorithm. For example, `ClearBit` objects can be instrumented to track the depth[1] and multiplicative depth of each bit involved in the calculation. Straightforwardly, the depth of the result of either the `xor`ing or the `and`ing of two bits of depth d_1 and d_2 is $1 + \max(d_1, d_2)$ and the *multiplicative* depth of the result of the `xor`ing (respectively the `and`ing) of two bits of multiplicative depth d'_1 and d'_2 is $\max(d'_1, d'_2)$ (respectively $1 + \max(d'_1, d'_2)$). The maximum depth and multiplicative depth can be tracked along an initial clear domain execution so as to dimension the number of levels of a BGV-style cryptosystem for later execution*s* in the encrypted domain.

5.3.3 Generating compilation data

In addition, the `ClearBit` objects can be instrumented in order to explicitly build the acyclic directed graph representing the Boolean circuit underlying the algorithm. This is a very convenient representation at least for two reasons. First it reveals a high degree of parallelism, as the so-called equivalence classes with respect to a topological ordering of the graph vertices reveal (potentially) large sets of operators which can be performed in parallel. This is crucial in order to mitigate the performance hit of using homomorphic encryption. Second, this representation allows to perform fine grain optimized scheduling of the calculations.

[1] By depth of a bit, we mean, similarly to the circuit depth, the length of the longest path from the circuit inputs to the operator that computes the said bit.

5.3.4 Overview of the compilation process

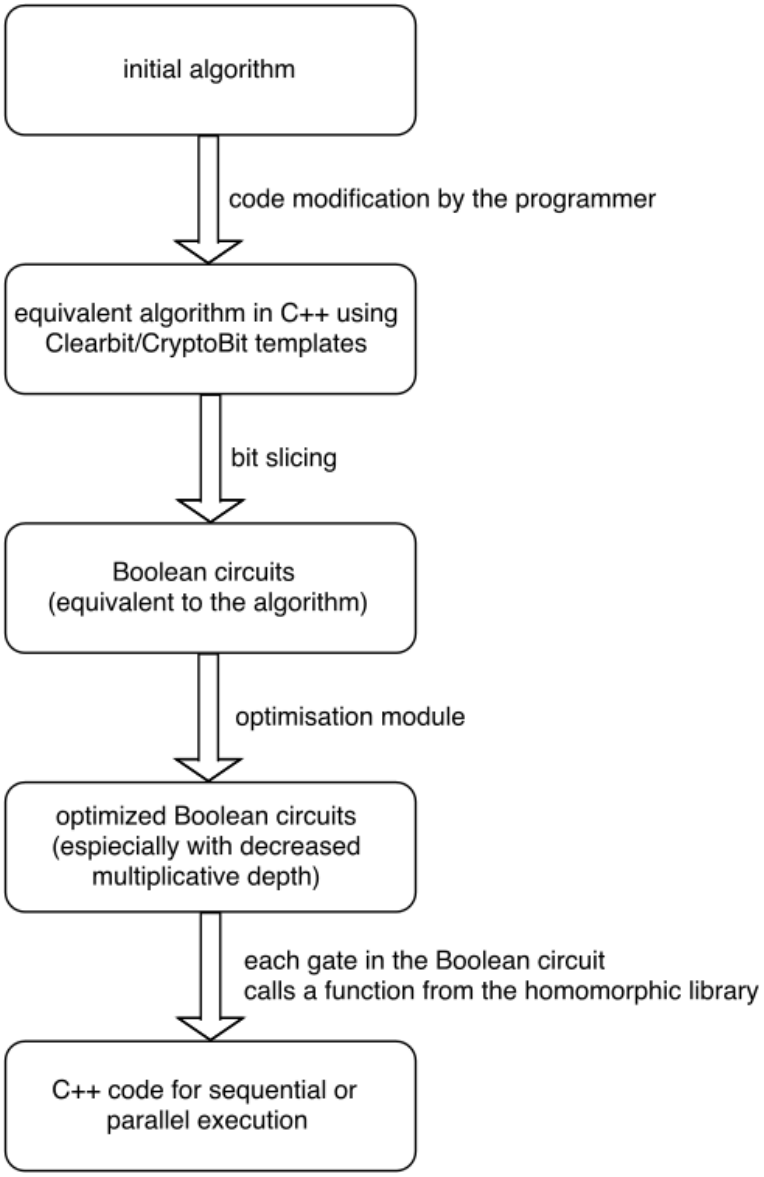


Figure 5.1: Compilation process.

Once we have all the tools described above, we can consider any program with a static control structure. In order to run such a program in the encrypted domain, we follow the process schematized in Figure 5.1:

- The first step is for the programmer to translate the initial code into an equivalent C++ code using templates that can be instantiated with either the `ClearBit` or the `CryptoBit` object. This rewriting of the initial

code does not require a full understanding of the underlying structure and therefore should be quite easily manageable by a programmer without any cryptographic knowledge.

- The second step is the construction of the Boolean circuits. We use the C++ template classes to slice integers into bits and then define all the arithmetic operations supported by the encryption scheme bitwise. We already saw that from XOR and AND we are able to provide a lot of other operations, so that is exactly what we put in our integer template. When an operation is performed on a sliced integer, it creates a gate in the Boolean circuit. This way, the execution of the program with our C++ templates tracks all the bitwise operations and it builds our Boolean circuit, which is equivalent to the initial program.

- The third step is to optimise our Boolean circuit. Indeed, we already mentioned that the execution in the encrypted domain when using FHE makes the multiplication much more costly than the addition. The multiplicative depth of the Boolean circuit is also a variable we want to minimize as much as possible. It means that from our Boolean circuit, we seek to build another (equivalent) Boolean circuit with decreased multiplicative depth and fewer AND gates (we can consider XOR gates to be practically free compared to AND gates). Minimizing the multiplicative depth of a Boolean circuit is not a classic issue and there is no proven method to do so as far as we know. In this thesis, we just began to look at this issue and tried to identify what directions to take. However, fellow researchers of the laboratoire LaSTRE used the ABC system for synthesis and verification of binary sequential logic circuits to modify the Boolean circuits our way. As it is not part of this thesis, we won't elaborate on this work, but more details can be found in [CDS15].

- The final step is the execution of the Boolean circuit either in the clear or the encrypted domain. Every XOR or AND gate calls respectively the addition or the multiplication available. In the clear it is simply the usual binary addition and multiplication but in the encrypted domain it calls the homomorphic addition and multiplication relative to the encryption scheme chosen. These homomorphic operations that we described in the first part are coded in C++ in the `CryptoBit` class. More than the addition and multiplication of ciphertexts, all the operations needed to manage the noise created when performing homomorphic operations have to be defined. This part needs, especially the parameters setting (ciphertext space, modulus size, etc.), requires a full understanding of the encryption scheme. When the ciphertext space is a polynomial ring, such as it is for the schemes

based on the R-LWE problem, a library like *flint* can be used efficiently to implement the homomorphic operations.

The C++ code generated from the Boolean circuit can be set for a sequential or a parallel execution.

5.3.5 Static control structure

As we mentioned earlier, the use of FHE sets a limit for the possible algorithm to run in the encrypted domain: these algorithms need to have a static control structure.

However, a program with a static control structure can be seen as an oblivious Turing machine. Indeed, an oblivious Turing machine is a Turing machine where the movements of the heads are independant of the input and the tapes are scanned, advanced and written to follow a predetermined sequence. In 1979, Pippenger and Fischer [PF79] demonstrated that any computation that can be performed by Turing machine with one-dimensional tapes in n can also be performed by an oblivious Turing machine with two-dimensional tapes in $\mathcal{O}(n \log n)$.

This means that any dynamic control program can be regularized as a static control program. Although the overhead might be significant, we are theoretically no longer limited in terms of programs that can be run in the encrypted domain using FHE.

Chapter 6

Private queries on encrypted database

In the cloud computing world, data confidentiality is an essential issue, yet very complicated to address. Once a client's data is outsourced, he usually has to choose between accessing and managing its data easily and garantee the confidentiality of the data. These criteria are most of the time mutually exclusive.

However, in the past few years, some solutions have been discussed and a few even designed, to offer confidentiality to the client, yet letting him have some wiggle room to access and manage its data. Various cryptography tools have been used in these solutions, and among them is additively homomorphic encryption. We can cite for example the tool offered by Google (in beta version) called "encrypted bigquery client". This solution uses Paillier's additively homomorphic encryption scheme combined with encrypted keywords searching. Thanks to homomorphic addition, the client is able to sum encrypted numbers (like prices) or multiply an encrypted number by a clear constant (which is ultimately just an addition and/or substraction of encrypted numbers).

On the other hand, fully homomorphic encryption has not been used that way in the real world yet (to our knowledge).

6.1 Defining the use case: client/server model

A client owns a database that he would like to outsource to a distant server. In our model, the server is not trusted in terms of confidentiality, but it will perform the operations that it is asked to (honest but curious).

We suppose that client and server are communicating through the internet (or a comparable channel speed-wise). The client does not need much computational power, like a standard personal computer. We can also imagine the client using a small device with little computational power but enhanced for some cryptographic primitives. For example, a device that could perform accelerated modal

multiplications would be enough on the client side even if its general computational power was lower than that of a personal computer.

6.2 Encryption of the database

In our solution, we only use bit per bit encryption, mainly for simplicity and manageability purposes. Hence, before any encryption, each entry of the database has to be decomposed into bits like we described in Chapter 5.

On the client, key generation operations are performed (once for all), which gives a private key and a public key. Typically, a private key is a short vector of integers, that the client will not reveal to anyone. The public key includes an encryption key and a computation key. The encryption key is used to create ciphertexts when given a plaintext (i.e. a bit here) and will be revealed to the server in the first steps. Indeed, the server will need to create ciphertexts to include its private data into the computation. For example, this is necessary if the server offers to perform in the encrypted domain a private algorithm. The computation key is used by the server to perform operations on ciphertexts. It is intrinsic to the use of (leveled)-fully homomorphic encryption.

The client will then proceed the encryption of his database and upload it to the server, as well as the public key.

6.3 Example of a private query

In order to describe a private query, we choose to take the example of a database of IP addresses (32 bits of data).

A possible query of the client would be the following: the client wants to know what are the IP addresses in his database which last 8-bit number is "255". We detail now the steps corresponding to an private query in Figure 6.1 (inside the dashed lines).

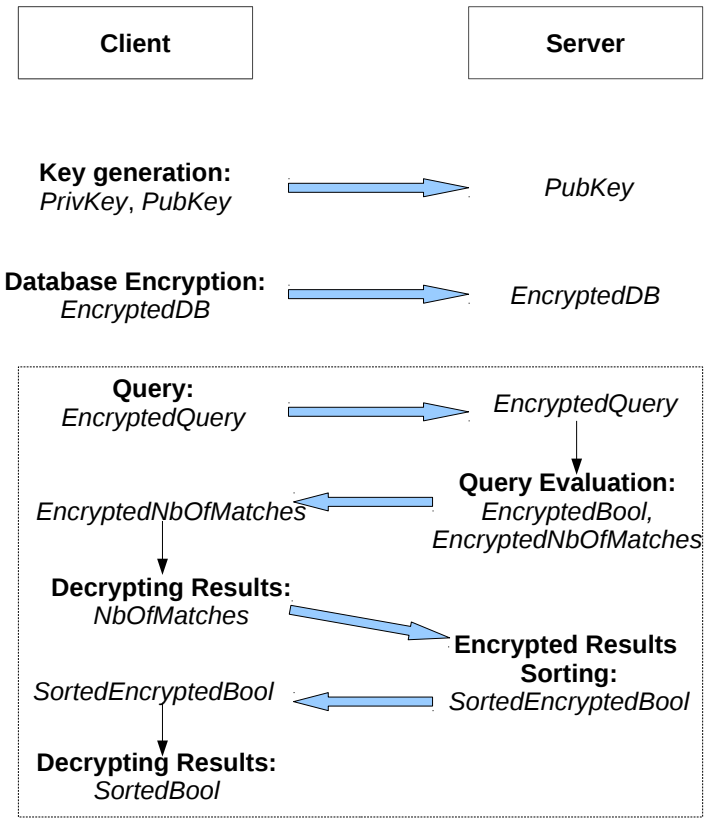


Figure 6.1: Flowchart of private querying on an encrypted database.

Query construction The query we take in this example is: "what are the IP addresses in my database which last byte is *255*?"

The integer "255" or in binary "11111111" is encrypted bit per bit using the encryption key. The resulting ciphertexts are carried in a *EncryptedQuery* type variable that is sent to the server. The client also sends (explicitly) a request for comparing the last byte in each entry of his database with his encrypted *query* variable.

Query evaluation The server navigates through the encrypted database and performs the following operations:

for *int* $j = 0; j < N; j++$ **do**
  CryptoBool $is\_a\_match[j] = (!(req < data[j])) * (!(req > data[j]))$;
end

where $data[j]$ is an encrypted entry of the database, req is the previoulsy encrypted number "*255*" and **CryptoBool** is the type defined for encrypted booleans (according to the encryption scheme).

The result $is\_a\_match[j]$ is the encryption of either *0* (if $data[j]$ is not a match) or *1* (if $data[j]$ is a match).

Optionnaly, the server can compute the (encrypted) number of matches for all or part of the database and return it to the client. In that case, the server simply performs:

$$\Sigma_{j=0}^{j=n}\ is\_a\_match[j]$$

The result is an encrypted integer that the client can decrypt. The client can then decide whether he wants to know all the entries that matched his request, or just some part of it, or none. We assume for this example that the client asks to the server for all the matches.

At that point, the client can ask the server for a number of results to get the matches of his query, depending on how much information he is willing to reveal to the server.

Encrypted Results Sorting On the client's demand or automatically, the server will now sort the results so that the entries that matched get on the top of the list. In order to do that, it works on the encrypted indexes of the database entries. It performs a bubble sort like algorithm where the condition for two encrypted indexes $Encryption(j_1)$ and $Encryption(j_2)$ to be swapped are the encrypted booleans $is\_a\_match[j_1]$ and $is\_a\_match[j_2]$.

Since this sorting has to be performed in the encrypted domain, it is done according to process described in Chapter 5, using the ***select*** operator:

for *int* $j = 0; j < N; j++$ **do**
  for *int* $k = 0; k < N; k++$ **do**
    CryptoBool $swap =$
    $is\_a\_match[j] + is\_a\_match[j-1] * is\_a\_match[j]$;
    CryptoInt $buf =$ **select**$(swap, Encryption(j-1), Encryption(j))$;
    $Encryption(j-1) =$
    select$(swap, Encryption(j), Encryption(j-1))$;
    $Encryption(j) = buf$;
  end
end

where *swap* is the encrypted boolean which value decides whether the two encrypted indexes will be swapped or not, **CryptoInt** is the type for encrypted integers and *select* the operator discussed in Chapter 5.

We want to stress that the encrypted data in this step are all very small: they are either Boolean or integers of size $\log_2(N)$, where N is the total number of entries in the database.

When these computations are done, the server will return the first n encrypted indexes on the sorted list, where n is the number set by the client. For example, if the client wants all the indexes that matched, he can set n to be the number of matches he was able to decrypt earlier. However, he can also set n to a bigger number, in order to hide from the server what was the number of matches for his query. This strategy has to be established considering what degree of confidentiality the client requires in a tradeoff with the speed at which he wishes to get his results.

Note: By a previous agreement between the client and the server, the last one can work directly with the encrypted entries of the database instead of the encrypted indexes, thus returning to the client directly the encrypted entries that matched the client's query.

Decrypting Results The client decrypts the encrypted results (indexes or entries) using the private key. In the homomorphic encryption scheme we use, the decryption algorithm is very simple and can be significantly optimised since the operations of decryption are very structurally predictible.

For example, using the homomorphic encryption scheme of [BLLN13a], the decryption of a ciphertext takes under 5 ms on an 2.9 GHz Intel Core i7.

We want to stress that, in the query routine, all the operations performed on the client side are quite simple, allowing the client to use a device with small computational power. This issue will be discussed more extensively in the next section.

Field of possible queries The field of possible queries is limited by the homomorphic operations available: addition and multiplication of bits. Still, we are able to perform:

- selecting encrypted entries using their encrypted indexes
- selecting encrypted entries with conditions defined using comparators such as: equality, greater than, less than, bounded between two values.
- counting the number of entries that satisfy such conditions
- modifying entries that satisfy such conditions

6.4 Dimensioning the computations and optimisations

For our solution to work, we need to use a fully homomorphic encryption scheme, or at least a leveled fully homomorphic encryption scheme. We chose to focus on the leveled ones because they seem more promising, as bootstrapping is complex and costly and the computations we need can be multiplicatively bounded.

In a leveled FHE scheme, the sizes of the public and private keys depend on the multiplicative depth of the computations, as discussed in chapter 1. Therefore, we have to dimension precisely the computations that will be performed by the server in order to generate adequate keys.

Since the multiplicative depth depends on values such as the size of the database entries, the number of entries in the database and the type of query we want to perform, the keys generated by the client will only be adequate for a certain database and certain types of query.

However, in order to avoid having to generate another set of key each time an entry is added to the database, we suggest to split the client's database into equal parts of N entries. Our goal is to find the most efficient N, split the database into several N-entry databases and let the server work on each database independantly.

The integer N is set so that it allows the server to perform the queries required by the client, i.e. the resulting multiplicative depth of the whole process is low enough for the encryption scheme to be able to handle such queries. Therefore the choices for N are from 1 to the maximum number that keeps the multiplicative depth low enough.

Using our platform and the ClearBit C++ class, we can find out what the overall multiplicative depth is for different values of N and different queries. Below is a table of the multiplicative depths of sorting the results of various queries, with respect to the size of the integers in the database n and the number of entries in the database N.

N	Query depth	size of an (integer) entry (bits)	Sorting depth
5	4	4	18
5	8	8	26
10	4	4	28
10	8	8	36

Table 6.1: Multiplicative depth with respect to the query type, the size of the DB and the size of an entry.

Once we are settled with a leveled-FHE scheme and given the size of the entries of the database, we can use this table to chose the best N, i.e., the N for which the overhead computation due to the encryption will be the smallest. Then, we have to split the database in parts of N entries and process these parts independantly.

However, before we do this, we should optimise the computations performed in the encrypted domain in a specific way.

Boolean optimisations Having in mind that the multiplicative depth is the key value to dimension our computations, we want to decrease this value as much as we can. Therefore, the optimisations that we seek are to eliminate the homomorphic multiplications, i.e. the AND gates in the boolean version of the algorithms.

This issue will be discussed more thoroughly in the next chapter but we can already state that these optimisations will be crucial for this application. Indeed, the multiplicative depth determines the most efficient number of entries N in each subset of the database, so the smallest the multiplicative depth is, the smallest number of subsets is needed. In addition, the Boolean optimisations will be all the more effective that they apply to each subset.

Chapter 7

Experimental platform and results

7.1 Evolution of the implementations and results

7.1.1 Implementation of vDGHV

This implementation was achieved during an internship previously to the thesis and the point was to demonstrate experimentally that we were able to perform (small) computations on encrypted numbers. The vDGHV scheme was chosen for its simplicity even though it was not at the time the most promising encryption scheme performance wise.

The goal was to achieve a 16-bit computer (i.e. that can add and multiply integers as long as the result is smaller than $2^{16} - 1$) that works in the encrypted domain. The integers are encrypted using the homomorphic vDGHV scheme and all the additions and multiplications are done on encrypted data.

For this implementation, the security parameter λ was set to 8 (when $\lambda \geq 10$, the key generation failed to complete). The size of the secret key p is $\eta = \lambda^2$ and of the public key $\gamma = \lambda^5$.

This was a first implementation and we managed to multiply very small integers in the encrypted domain (smaller than 10). Although it had no practical use, it was a good initiation to the implementation of a homomorphic encryption scheme.

7.1.2 Experimental results with vectorial BGV (LWE)

We have developed a prototype of the compilation and execution infrastructure sketched in the previous chapter and (seamlessly) interfaced it with two somewhat fully homomorphic cryptosystem implementations: our own implementation of the vectorial flavor of the BGV cryptosystem and a public domain implementation of the Smart-Vercauteren one [PBS11] available on `http://www.hcrypt.com`.

Our prototype supports all the functions that have been presented in Chapter 5, including Boolean circuit generation and parallel execution.

As far as the implementation of the BGV cryptosystem is concerned, in order to avoid redundant level shifts (i.e., calls to `Rescale` on an i-th level ciphertext when there already is a $i-1$-th version of said ciphertext), we have implemented a depth caching technique whereby each `CryptoBit` object remembers all its different-level copies in a small associative data structure keyed by level. This technique results approximately in speedups of around 45%.

Table 7.1 provides characterization data for a number of elementary algorithms obtained using instrumented clear domain bit-level executions. For each algorithm, the number of bit-level additions (# add), the number of bit-level multiplications (# mul), the depth, the multiplicative depth ($\times$ depth) as well as the average number of operations per topological equivalence classes of the underlying Boolean circuit (a number which gives an idea of the amount of circuit-level parallelism and is labeled "av. //") are given. The multiplicative depth is necessary to parametrize the BGV scheme (it tells how many levels we need to be able to handle). The other figures can be used to try to predict the performances of a homomorphic evaluation of these algorithms (or at least what we should expect about the level of performances).

Parallelism is handled in two (so far exclusive) different ways, either internally to the cryptosystem or externally at the Boolean circuit level.

Internal parallelism is handled via an OpenMP parallel for pragma in the outer loop of the matrix product in `SwitchKey` (which as already emphasized is the main hot point, performance-wise). This parallelization strategy results in further speedups of around 41% on an average dual core laptop and seems to be the optimal strategy for this kind of machines.

Table 7.2 provides experimental results obtained on a laptop with a 2 GHz Intel dual core processor, using both the aforementioned depth cache and `SwitchKey` parallel `for`. The metrics given are the execution time ("CPU"), the percentage of depth cache hits ("cache eff.") as well as the size of the overall public key ("pubk size") which accounts for the size of the public keys of the cryptosystem at each level and the key switching matrices. Lastly, for completeness sake, Table 7.2 also presents the execution times we have obtained on the same set of elementary algorithms using the HCRYPT library of Brenner et al. (`www.hcrypt.com`).

External parallelism, i.e. parallelism at the Boolean circuit topological equivalence classes level, is intended to target the execution of heavier algorithms on higher-end multicore machines. Although we cannot report on a speedup measurement, this external parallelism strategy has allowed us to perform a full 32-bit 256-point FFT in less than four hours on a 48 cores AMD-based NUMA machine, a calculation which otherwise appeared to be undoable in "non prohibitive" time.

However, the reader should be warned that these results have been obtained using cryptosystem parameter values which are presumably too small to provide a non trivial level of security. They should thus be considered giving more of an optimistic lower bound on the level of performance which can be achieved using the BGV system rather than a conservative upper bound. In our opinion, despite

	$b^2 - 4ac$ (8 bits)	$b^2 - 4ac$ (16 bits)
# add	332	1188
# mul	302	1126
depth	43	83
× depth	16	32
av. //	14.74	27.88
	$\sum_{i=1}^{10} t[i]$ (8 bits)	$\sum_{i=1}^{10} t[i]$ (16 bits)
# add	207	423
# mul	135	279
depth	24	48
× depth	8	16
av. //	6.75	14.62
	b. sort (10×4 bits)	b. sort (10×8 bits)
# add	1620	3240
# mul	1350	2790
depth	214	350
× depth	68	136
av. //	13.88	17.23
	FFT (256×32 bits)	
# add	7291592	
# mul	5296128	
depth	674	
× depth	166	
av. //	18676.10	

Table 7.1: Characterization of a few elementary algorithms.

	$b^2 - 4ac$ (8 bits)	$b^2 - 4ac$ (16 bits)
CPU	0.406 s	4.124 s
cache eff.	46%	40%
pubk size	1.1 MB	7.8 MB
HCRYPT	58.9 s	3 m 39 s
	$\sum_{i=1}^{10} t[i]$ (8 bits)	$\sum_{i=1}^{10} t[i]$ (16 bits)
CPU	0.125 s	0.562 s
cache eff.	47%	47%
pubk size	196 kB	1.1 MB
HCRYPT	27.2 s	55.4 s
	b. sort (10×4 bits)	b. sort (10×8 bits)
CPU	5.219 s	18.110 s
cache eff.	64%	64%
pubk size	68.5 MB	525 MB
HCRYPT	5 m 5 s	9 m 41 s

Table 7.2: Execution times for a number of elementary algorithms on an average dual core laptop.

the fact that BGV-style cryposystems enjoy very strong theoretical security properties, practical parameter setting for the BGV system as well as for its siblings is a question that still needs additional theoretical investigations. These figures are representative as they have been obtained with one of the first implementations of a full blown fully homomorphic cryptosystem.

In addition to these results, we managed to execute the sum of 10 4-bit elements over the variant of the BGV scheme of [GHPS12] with larger parameters. With an approximative 40-bit security level, the sum of encrypted elements took about 1 minute (without parallelization). For information, a 64-bit security level is considered suitable for small attackers, 80-bit is the smallest general-purpose protection and 128-bit is considered a long-term protection. Testing our implementation with a higher security level on various algorithms and developping compilation tools will be the subject of future work.

Finally, the execution times of HCRYPT have been obtained with default parameters (which are also too small to provide a non trivial level of security), as the underlying FHE scheme in our system. Something we were able to do seamlessly (as soon as an HCRYPT-based `CryptoBit` class was implemented). Although the performances obtained with our implementation of BGV appear to be much better, we should still emphasize that these results are hard to compare to those of Table 7.2 for two reasons. First, the HCRYPT library implements the bootstrapping-based Smart-Vercauteren FHE scheme which is by no means a potentially non prohibitive scheme. Second, we only have a limited understanding of the extent to which parallelism is used in that library (as well as its numerous

dependencies).

7.1.3 Experimental results with single-key polynomial BGV (RLWE)

Table 7.3 provides experimental results obtained on a laptop with a 2 GHz Intel dual core processor, with or without the aforementioned parallel for. The execution time ("CPU"), the parallel for speedup and the security level (λ) are given.

	$\sum_{i=1}^{10} t[i]$ (4 bits)	threshold (4 bits)
CPU seq.	54.9 s	193.4 s
CPU //	36.3 s	140.2 s
speedup	33.9%	27.5%
λ	40	40
	$\sum_{i=1}^{10} t[i]$ (4 bits)	$b^2 - 4ac$ (4 bits)
CPU seq.	77.6 s	158.9 s
CPU //	51.2 s	107.5 s
speedup	34%	32.3%
λ	80	40

Table 7.3: Execution times for a number of elementary algorithms with the polynomial BGV.

These results show that we can achieve homomorphic computation with a non-trivial level of security for circuits of small multiplicative depth, although the overhead makes it still impractical. As we have said earlier, the ciphertexts are vectors of thousands of integers and the experimentation revealed that the memory issue is in fact more limitating than the cost of homomorphic operations themselves, at least when working on an ordinary computer. While the (per bit) computational overhead has decreased fastly over the past few years and is getting closer or even better than other existing solutions, the ciphertext growth still requires (too) much RAM memory. Indeed, we implemented a "depth cache" process to avoid redundant `Rescales`, but it turns out the memory used by the cache is slowing the computation more than the additional `Rescales` (at least for the polynomial flavor of BGV and running on a basic computer).

In addition to these results, we were able to execute the same algorithms using the other two cryptosystems: the vectorial BGV and the Smart-Vercauteren cryptosystem. For the latter, the authors used the public implementation of the Smart-Vercauteren cryptosystem HCRYPT (www.hcrypt.com) to build a `CryptoBit` class. This way, the high-level algorithms can be executed using any cryptosystem, providing the writing of its own `CryptoBit` class. However, the only results we were able to get with the vectorial BGV are for toy parameters

(with respect to security). Similarly, the set parameters for HCRYPT are of trivial security. For these reasons, the previous results cannot be compared with the ones we give in this chapter, since we achieve here a level of security of 40 and 80 (against 10 or 15 at most for the previous results). That is why we decided not to include them along with the results of the polynomial flavor of BGV, but they can be found in [AMFF+13].

7.1.4 Experimental results with FV aka Brakerski12

In the context of this thesis, an implementation of the R-LWE setting of Brakerski's scale-invariant scheme has been done. It is used as the underlying scheme in our platform and thanks to the good performances of the scheme, it is possible to run the medical algorithm discussed in Section 4.2 in the encrypted domain.

The security parameter is set at 128 according to the security analysis of [FV12]. To make the example more realistic, the client side has been set up on a tablet where you simply put the results of various tests. The tablet encrypts homomorphically the medical data and sends them to the server that computes the algorithm in less than 1 second.

Figure 7.1: A photo of the client application for the input and encryption of the tests results

Conclusion

During this thesis, we developped a platform for the execution of algorithms in the encrypted domain using Fully Homomorphic Encryption. We showed that FHE was already efficient for a range of algorithms with small multiplicative depth. More recent experimentations on the same platform were done in the laboratoire LaSTRE, including the execution of a medical algorithm on encrypted data in a realistic runtime for practical use.

For the future of FHE, there is room for more efficiency but also some concerns. First, the security, while based on strong mathematical problems, is not exaclty stable in reality. Attacks on lattice-based cryptography are still evolving, all the more since the interest for FHE has grown. Most of the practical parameters found in the articles are calculated according to the then best known attacks and need to be updated. The computational overhead caused by the use of FHE has decreased a lot for the past few years, but the state of the art has yet to stabilize for FHE to be a widespread real-life solution.

Secondly, while the cryptographic primitives have improved, there is very little work about how reworking the algorithm (or the corresponding Boolean circuits) can dramatically change the computation on homomorphically encrypted data. In our platform, fellow researchers added the possibility to use the ABC tool to modify the Boolean circuits so that the multiplicative depth would be decreased. However, this work is just beginning and the automatic transformation of Boolean circuits according to the specificities of FHE would bring a lot to the subject.

Finally, the possibility to use FHE conjointly with a lighter symmetrical encryption scheme has not been very much explored outside the use with AES. Since AES has a high multiplicative depth, it would be interesting to find another encryption scheme with strong security but more "homomorphic-friendly", or even design a new one. An article to appear is dedicated to the implementation of FHE with a streamcipher and was done with the collaboration of Dr Carpov, Fontaine and Sirdey [CCF+15].

Bibliography

[AMFF+13] C. Aguilar-Melchor, S. Fau, C. Fontaine, G. Gogniat, and R. Sirdey. Recent advances in homomorphic encryption: A possible future for signal processing in the encrypted domain. *IEEE Signal Process. Mag.*, 30(2):108–117, 2013.

[AMGH10] C. Aguilar-Melchor, P. Gaborit, and J. Herranz. Additively homomorphic encryption with d-operand multiplications. In *CRYPTO'10*, volume 6223 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2010.

[ARS05] A. Adelsbach, M. Rohe, and A.-R. Sadeghi. Non-interactive watermark detection for a correlation based watermarking scheme. In *Communications and Multimedia Security: 9th IFIP TC-6 TC-11International Conference, CMS 2005*, number 3677 in Lecture Notes in Computer Science, pages 129–139. Springer-Verlag, 2005.

[BBC+10] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, A. Piva, and F. Scotti. A privacy-compliant fingerprint recognition system based on homomorphic encryption and fingercode templates. In *Proc. of BTAS 2010, IEEE Fourth International Conference On Biometrics: Theory, Applications And Systems*, 2010.

[BCP13] Julien Bringer, Hervé Chabanne, and Alain Patey. Privacy-preserving biometric identification using secure multiparty computation: An overview and recent trends. *IEEE Signal Process. Mag.*, 30(2):42–52, 2013.

[BFL+11] M. Barni, P. Failla, R. Lazzeretti, A.-R. Sadeghi, and T. Scheider. Privacy-preserving ECG classification with branching programs and neural networks. *IEEE Transactions on Information Forensics and Security*, 6(2):452–468, 2011.

[BGV12] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, pages 309–325, 2012.

[BHHO08] Dan Boneh, Shai Halevi, Michael Hamburg, and Rafail Ostrovsky. Circular-secure encryption from decision diffie-hellman. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008. Proceedings*, pages 108–125, 2008.

[BLLN13a] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2013:75, 2013.

[BLLN13b] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings*, pages 45–64, 2013.

[BPB09a] T. Bianchi, A. Piva, and M. Barni. Encrypted domain DCT based on homomorphic cryptosystems. *EURASIP Journal on Information Security*, 2009(Article ID 716357), 2009.

[BPB09b] T. Bianchi, A. Piva, and M. Barni. Implementing the discrete Fourier transform in the encrypted domain. *IEEE Transactions on Information Forensics and Security*, 4(1):86–97, 2009.

[BPB10] T. Bianchi, A. Piva, and M. Barni. Composite signal representation for fast and storage-efficient processing of encrypted signals. *IEEE Transactions on Information Forensics and Security*, 5(1):180–187, 2010.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 868–886, 2012.

[CCF+15] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, TancrÃ¨de Lepoint, MarÃa Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. How to compress homomorphic ciphertexts. Cryptology ePrint Archive, Report 2015/113, 2015. http://eprint.iacr.org/2015/113.pdf.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. Armadillo: A compilation chain for privacy preserving applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing, SCC@ASIACCS '15, Singapore, Republic of Singapore, April 14, 2015*, pages 13–19, 2015.

[CF14] Dario Catalano and Dario Fiore. Boosting linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. *IACR Cryptology ePrint Archive*, 2014:813, 2014.

[CJJ+13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 353–373, 2013.

[CLT13] J.-S. Coron, T. Lepoint, and M. Tibouchi. Batch fully homomorphic encryption over the integers. *IACR Cryptology ePrint Archive*, 2013, 2013.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *Public-Key Cryptography - PKC 2014 - 17th International Conference on Practice and Theory in Public-Key Cryptography, Buenos Aires, Argentina, March 26-28, 2014. Proceedings*, pages 311–328, 2014.

[CNT12] J.-S. Coron, D. Naccache, and M. Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 446–464. Springer, 2012.

[EFG+09] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, R. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies*, pages 235–253, 2009.

[EPK+07] Z. Erkin, A. Piva, S. Katzenbeisser, R. Lagendijk, J. Shokrollahi, G. Neven, and M. Barni. Protection and retrieval of encrypted multimedia content: when cryptography meets signal processing. *EURASIP Journal on Information Security*, 2007:Article ID 78943, 2007.

[FB10] P. Failla and M. Barni. Gram-Schmidt orthogonalization on encrypted vectors. In *Proc. of the 21st International Tyrrhenian Workshop on Digital Communications, ITWDC 2010*, 2010.

[FG07] C. Fontaine and F. Galand. A survey of homomorphic encryption for nonspecialists. *EURASIP J. Inf. Secur.*, 2007(1):1–15, 2007.

[FSB10] P. Failla, Y. Sutcu, and M. Barni. eSketch: a privacy-preserving fuzzy commitment scheme for authentication using encrypted biometrics. In *Proc. of the ACM Multimedia and Security Workshop 2010*, 2010.

[FSF+13] Simon Fau, Renaud Sirdey, Caroline Fontaine, Carlos Aguilar Melchor, and Guy Gogniat. Towards practical program execution over fully homomorphic encryption schemes. In *Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC 2013, Compiegne, France, October 28-30, 2013*, pages 284–290, 2013.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[Gen09a] C. Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[Gen09b] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of STOC'09*, pages 169–178. ACM Press, 2009.

[GH11] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT'2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011.

[GHPS12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. *IACR Cryptology ePrint Archive*, 2012:99, 2012.

[GHV10a] C. Gentry, S. Halevi, and V. Vaikuntanathan. A simple BGN-type cryptosystem from LWE. In *EUROCRYPT'2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 506–522. Springer, 2010.

[GHV10b] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. *i*-hop homomorphic encryption and rerandomizable yao circuits. In *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, pages 155–172, 2010.

[GKP+13] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable garbled circuits and succinct functional encryption. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 555–564, 2013.

[GM82] S. Goldwasser and S. Micali. Probabilistic encryption and how to play mental poker keeping secret all partial information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 365–377, 1982.

[Gol93] Oded Goldreich. A uniform-complexity treatment of encryption and zero-knowledge. *J. Cryptology*, 6(1):21–53, 1993.

[KLC+08] S. Katzenbeisser, A. Lemma, M. Celik, M. van der Veen, and M. Maas. A buyer-seller watermarking protocol based on secure embedding. *IEEE Transactions on Information Forensics and Security*, 3(4):783–786, 2008.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[KT05] M. Kuribayashi and H. Tanaka. Fingerprinting protocol for images based on additive homomorphic property. *IEEE Transactions on Image Processing*, 14(12):2129–2139, 2005.

[LEB] R. Lagendijk, Z. Erkin, and M. Barni. Encrypted Signal Processing for Privacy Protection. *IEEE Signal Processing Magazine, January 2013, to appear*.

[LN14] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes FV and YASHE. In *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*, pages 318–335, 2014.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 287–302, 2004.

[MSS11] Steven Myers, Mona Sergi, and Abhi Shelat. Threshold fully homomorphic encryption and secure computation. *IACR Cryptology ePrint Archive*, 2011:454, 2011.

[PBS11] H. Perl, M. Brenner, and M. Smith. Poster: an implementation of the fully homomorphic Smart-Vercauteren crypto-system. In *ACM Conference on Computer and Communications Security*, pages 837–840, 2011.

[PCC+06] A. Piva, V. Cappellini, D. Corazzi, A.D. Rosa, C. Orlandi, and M. Barni. Zero-knowledge ST-DM watermarking. In *IS&T/SPIE International Symposium on Electronic Imaging 2006 - Security, Steganography, and Watermarking of Multimedia Contents VIII*, 2006.

[PEL07] J.P. Prins, Z. Erkin, and R. Lagendijk. Anonymous fingerprinting with robust QIM watermarking techniques. *EURASIP Journal on Information Security*, (Article ID 31340), 2007.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[Reg10] O. Regev. The Learning with Errors Problem (invited survey). In *IEEE Conference on Computational Complexity*, pages 191–204, 2010.

[RSA78] R. 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.

[SS10] Damien Stehlé and Ron Steinfeld. Faster fully homomorphic encryption. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, volume 6477 of *Lecture Notes in Computer Science*, pages 377–394. Springer, 2010.

[SS11] D. Stehlé and R. Steinfeld. Making NTRU as secure as worst-case problems over ideal lattices. In *EUROCRYPT*, pages 27–47, 2011.

[SV10] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography, PKC'2010*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010.

[TPPG06] J.R. Troncoso-Pastoriza and F. Pérez-González. Zero-knowledge watermark detector robust to sensitivity attacks. In *ACM Multimedia & Security - MM&SEC 2006*, pages 97–107, 2006.

[vDGHV10] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT'2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.