2013 Eighth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

Towards practical program execution over fully homomorphic encryption schemes

Simon Fau*, Renaud Sirdey*, Caroline Fontaine†, Carlos Aguilar-Melchor‡ and Guy Gogniat§

*CEA, LIST, France
Email: simon.fau@cea.fr,renaud.sirdey@cea.fr
†Lab-STICC, CNRS and Télécom Bretagne
Université Européenne de Bretagne, France
Email: caroline.fontaine@telecom-bretagne.eu
‡XLIM, Université de Limoges, France
Email: carlos.aguilar@xlim.fr
§Lab-STICC, Université de Bretagne Sud, France
Email: guy.gogniat@univ-ubs.fr

***Abstract*—This paper intends to provide a first assessment of the practicality of using Fully Homomorphic Encryption (FHE) to perform real calculations, in terms of software engineering as well as performances. We present a prototype of a compilation and execution infrastructure targeting any FHE scheme. The paper also provides some preliminary experimental results obtained with our implementation of the Brakerski-Gentry-Vaikuntanathan (BGV) scheme introduced in [6], which is one of the most promising FHE schemes with respect to practicality.**

I. Introduction

Since the introduction of the notion of Privacy Homomorphism by Rivest et al. [20] 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 (see [10] for a survey), the problem of designing such an obviously useful primitive has remained open until the theoretical breakthrough of C. Gentry [12], [11] 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 into the subject, it is important to notice that all FHE schemes are (by necessity) grounded in probabilistic encryption schemes which are traditionally asymmetric and usually operate at the bit level. More formally, such a scheme is specified by two functions $\mathtt{enc}_{\text{pubk}} : \mathbb{Z}_2 \longrightarrow \Omega$ and $\mathtt{dec}_{\text{privk}} : \Omega \longrightarrow \mathbb{Z}_2$, where Ω is a large cardinality set (e.g. $\mathbb{Z}_q^n$ in [6]) thereby ensuring that each of the (two) possible plaintexts are associated to many distinct ciphertexts. The homomorphic part of the cryptosystem then requires the specification of two additional functions working in the encrypted domain, $\oplus_c : \Omega \times \Omega \longrightarrow \Omega$ and $\otimes_c : \Omega \times \Omega \longrightarrow \Omega$, such that given $m_1 \in \mathbb{Z}_2$ and $m_2 \in \mathbb{Z}_2$ we have

$$\mathtt{dec}_{\text{privk}}(\mathtt{enc}_{\text{pubk}}(m_1) \oplus_c \mathtt{enc}_{\text{pubk}}(m_2)) = m_1 \oplus m_2$$

and

$$\mathtt{dec}_{\text{privk}}(\mathtt{enc}_{\text{pubk}}(m_1) \otimes_c \mathtt{enc}_{\text{pubk}}(m_2)) = m_1 \otimes m_2,$$

where $\oplus$ and $\otimes$ respectively denote the `xor` and the `and` operator. Of course, being able to decrypt after one operation does not imply the ability to do so after an arbitrary number of operations has been performed. Hence, it is more generally required that for any polynomial $p_{\oplus;\otimes} : \mathbb{Z}_2^n \longrightarrow \mathbb{Z}_2$ the cryptosystem has the property that the associated polynomial $p_{\oplus_c;\otimes_c} : \Omega^n \longrightarrow \Omega$ is such that

$$p_{\oplus;\otimes}(m_1, \ldots, m_n) = \mathtt{dec}_{\text{privk}}(p_{\oplus_c;\otimes_c}(\mathtt{enc}_{\text{pubk}}(m_1), \ldots)).$$

This property is quite powerful as such a cryptosystem allows to transform any program with bounded input into an equivalent program computing in the encrypted domain (by transforming the bounded input program into a Boolean circuit and replacing the $\oplus$ and $\otimes$ gates by $\oplus_c$ and $\oplus_c$ operations).

The theoretical efficiency of an FHE scheme is measured by its per (bit level) operation computational overhead in terms of a security parameter, usually denoted λ (the cryptosystem being dimensioned such that the best known attack requires an order of magnitude of 2^λ operations). Although theoretically efficient, Gentry's initial construction (and most of its siblings) is impractical as its overhead is a high degree polynomial of λ. Having said that, the theoretical efficiency of FHE schemes has tremendously progressed over the last few years, with a lot of

978-0-7695-5094-7/13 $31.00 © 2013 IEEE
DOI 10.1109/3PGCIC.2013.48

contributions introducing new techniques [2], [7]. In 2011, the first (semi) linear [6] and polylog [15] overhead systems were presented, with a "leveled" FHE scheme where the number of homomorphic operations is limited, but the limit can be fixed (as large as required) and the system be parameterized accordingly (in a trade-off with efficiency).

At present, few implementations of fully homomorphic encryption schemes have been realized. In [13] an implementation of Gentry's breakthrough scheme, which was fairly impractical, is described. Two implementations of the simple but quite impractical FHE scheme over the integers of [23] have been realized: [9], [8], the latter focusing on evaluating AES homomorphically. In [14], Gentry et al. describe an optimized version of the BGV scheme for the AES circuit as well. Lately, Boneh et al. [3] present their own implementation of [5] for private database queries using homomorphic encryption. Finally, in [4] Bos et al. propose a new FHE scheme based on [22] with improved security and give some implementation results with practical parameters. We also want to mention the open source implementations HElib (https://github.com/shaih/HElib) of [6] and HCRYPT (www.hcrypt.com) of [21]. In a previous article [1], we used HCRYPT in our framework, in addition to our own implementation of the vectorial variant of BGV, but the parameters were trivial with respect to security. For the results of this paper, we chose to use the polynomial variant of the BGV scheme as it seems to be the most promising performance-wise, and it allows us to use more realistic security parameters. The interest of our framework is its genericity: we can use any FHE scheme to perform computations in the encrypted domain (modulo the code of the basic cryptographic operations of an FHE scheme). Hence, a developper would be able to interface his code with an FHE scheme with minimal integration. In this spirit, we intend to use the recent open source library HElib in our framework in the future, although it was not available at the time of testing.

This paper focuses mainly on the software engineering aspects of using FHE schemes in concrete applications. In particular, we focus on how to express algorithms seamlessly, regardless on whether they are executed in the plain (during testing) or in the encrypted domain (during operation). 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.

The paper then attempts to provide a first assessment of the practical value of the current generation of FHE schemes. In particular, we provide some characteristics (most notably the multiplicative depth which turns out to be an important parameter when performing homomorphic calculations, in particular with leveled schemes) as well as experimental results for a number of elementary but real algorithms (discriminant calculation, array summation, bubble sort, etc.) obtained using both a public implementation (by Perl et al. [17]) of the (fairly impractical) Smart & Vercauteren cryptosystem [21] as well as our implementation of two flavours of the (more practical) leveled cryptosystem BGV. Although our experimental results with the BGV system have been obtained on small computations, this work provides an analysis of the algorithmic structure of that system in particular with respect to its performance hot spots and on ways to mitigate them.

II. Overview of the BGV cryptosystem

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 decisional LWE (Learning With Errors) problem [18]) and the other one with integer polynomials (the security of which is linked with the hardness of the decisional R-LWE (Ring-Learning With Errors) problem [16]). In a few words, the decisional LWE (resp. R-LWE) 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 $\mathbb{A} = \mathbb{Z}_q^n/F(X)$) 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. $\mathbb{A}_q^n$) and e_i is sampled according to a Gaussian distribution. For more precisions on the (R)-LWE problem, we refer the reader to [19]. 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 $\mathbb{A} = \mathbb{Z}[X]/F(X)$ 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 $\mathbb{A}_{q_i} = \mathbb{A}/q_i\mathbb{A}$ of polynomials of $\mathbb{A}$ with integers coefficients into the range $]-q_i/2, q_i/2]$. In practice, elements in $\mathbb{A}_{q_i}$ will be polynomials represented by the d-vector of their coefficients.

Basic encryption functions

The private key $Priv$ is sampled in $\mathbb{A}$. A public key Pub consists in the private key masked by a noise component: $Pub = aPriv + 2e \in \mathbb{A}_{q_L}^N$, where $N = O(\log q_L)$, $a \in \mathbb{A}_{q_L}^N$ and the noise e is sampled from a "discrete" Gaussian distribution over $\mathbb{A}^N$ ("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 [6],[14] for a precise algorithmic description.

`Encrypt`(Plaintext m, PublicKey Pub): 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 $\mathbb{A}_{q_L}$ derived from the plaintext m, the public key Pub 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 $\mathbb{A}_{q_i}$. In our implementation, each ciphertext carries its level, i.e. the information that indicates in which subring it lies.

`Decrypt`(Ciphertext c, PrivateKey $Priv$): Plaintext m

The decryption function is a simple dot product between the ciphertext $c \in \mathbb{A}_{q_i}$ 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 will see in the following, the noise component of the ciphertexts can grow during the homomorphic operations. For the decryption to be correct, the noise must remain under a certain level. That is why we need to introduce the following operations, whose purpose is to reduce the noise of the ciphertexts after homomorphic operations.

Level shifting operations

`Rescale`(Cipertext c): Ciphertext c'

The function transforms the ciphertext $c \in \mathbb{A}^2_{q_i}$ into a ciphertext $c' \in \mathbb{A}^2_{q_{i-1}}$. 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" $c \in \mathbb{A}^3_{q_i}$. To retrieve a regular ciphertext in $\mathbb{A}^2_{q_i}$, 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 $c' \in \mathbb{A}^2_{q_{i-1}}$ (with low noise).

Homomorphic operations

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

For two ciphertexts c_1, c_2 where $c_1 \in \mathbb{A}^2_{q_{i_1}}$ and $c_2 \in \mathbb{A}^2_{q_{i_2}}$, 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}*)*

The resulting ciphertext has a noise component equal to the sum of the noise components of the input ciphertexts.

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

For two ciphertexts $c_1 \in \mathbb{A}^2_{q_{i_1}}$ and $c_2 \in \mathbb{A}^2_{q_{i_2}}$, 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 $c_3 \leftarrow c_1 \otimes c'_2$; *(*$c_3 \in \mathbb{A}^3_{q_{i_1}}$*)*
do $c_{mul} \leftarrow$ `SwitchKey`(c_3); *(*$c_{mul} \in \mathbb{A}^2_{q_{i_1-1}}$*)*

The tensored product applied on c_1 and c_2 consists in adding and multiplying polynomials of $\mathbb{A}_{q_{i_1}}$, which can be very expensive as we will see.
The noise component of the resulting ciphertext is approximately the product of the noise components of the input ciphertexts.

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 $\mathbb{A}$ (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 [6]). 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.

III. Manipulating integers in the encrypted domain

As already stated in the introduction, an FHE scheme allows us to evaluate any 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 $G = (V, A)$ whose vertices are either inputs, outputs or operators (`and` or `xor`) and whose edges represent data dependencies. In higher-level programming terms, this restricts us to programs or algorithms having bounded input as well as a control flow that is independent of encrypted data. In particular, this excludes (encrypted) data-dependant `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 now see.

Let us first state that most of the classical integer manipulation operators can be implemented using the `and` and `xor` operators available with the FHE scheme. Additions and multiplications can be implemented following textbook recipes for n-bit adders and multipliers (although choosing the most appropriate design for execution over an FHE scheme is not so straightforward, since an `and` will have a far greater cost than a `xor` gate). 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 an n-bit subtraction operator using an n-bit adder. It is then also possible to perform comparisons hermetically in the encrypted domain by subtracting the two operands to be compared and keeping the (encrypted) sign bit. The (Boolean) `not` operator can be obtained by `xor`ing the least significant bit with an encryption of 1.

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

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

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

Such an operator can then straightforwardly be rewritten as follows

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

Provided the implementations of addition, multiplication and negation mentioned earlier in this section, Eq. (1) translates as

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

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. (1), this algorithm can be rewritten in a suitable fashion for execution over an 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.

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

$$t[i] = \sum_{j=1}^{n} \delta(i, j) \times t[j], \tag{2}$$

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

$$t[j] = \delta(i, j) \times v \oplus (1 - \delta(i, j)) \times t[j], \forall j. \tag{3}$$

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.

We shall now see how all these can be put together in order to obtain a full solution from a software engineering point of view.

IV. 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 $\star$ 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.

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 $\star$ are trivial), instrumented clear bits (see `ClearBit` below) or crypto bits (in which case the + and $\star$ 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).

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.

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 in order to maximize the efficiency of certain mechanisms such as the depth caching technique discussed in the next section.

[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.

V. Some preliminary experimental results

We have developed a prototype of the compilation and execution infrastructure sketched in the previous section and (seamlessly) interfaced it with several fully homomorphic cryptosystem implementations: two implementations of the polynomial and vectorial flavors of the Brakerski-Gentry-Vaikuntanathan cryptosystem written by the authors and a public domain implementation of the Smart-Vercauteren one [17].

Our prototype supports all the functions that have been presented in Sect. IV, inclunding Boolean circuit generation and (possibly) parallel execution.

Table I (from [1]) provides characterization data for a number of elementary algorithms obtained using instrumented clear domain bit-level executions. For each algorithm, we provide the number of bit-level additions (`xor`s), the number of bit-level multiplications (`and`s), the depth, the multiplicative 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).

	$b^2 - 4ac$ (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 I. Characterization of a few elementary algorithms.

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

Internal parallelism in 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 II 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 II. 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 article, 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 [1].

VI. CONCLUSION

In this work, we have made a number of steps towards bridging the gap between non trivial algorithms and their practical, relatively seamless, execution on fully homomorphic encryption schemes. We have also provided some preliminary experimental results indicating that there is hope, in the near term, to be able to homomorphically execute simple algorithms on BGV-style cryptosystems in reasonable time.

Still, performance-wise, our results, in line with the results of other research teams (most notably [14]), show that the level of performance achieved is still far from enabling the execution of more computationally involved algorithms in non prohibitive time.

Despite this, there is hope in the sense that theoretical progress has been fast-paced since 2009 (whith the theoretical overhead decreasing by an order of magnitude equal to the square root every year or so). From a practical viewpoint, the cost of a homomorphic multiplication went from a couple of hours in 2010, to a few seconds in 2011 and to a few milliseconds in late 2012 (for similar architectures, usually a single core of a relatively powerful commercial processor). In comparison to a multiplication in the clear, we went from a factor 10^{12} in 2010 to a factor 10^6 in 2006. Such an evolution builds the hope of achieving overheads that are quite satisfactory for numerous applications in 2013-2014 (since countless applications only use a thousandth of a processor capacity). In addition, research work on algorithm "FHE-friendlyness", on compilation (in the wide sense) as well as on ad hoc optimized (parallel) execution support for these cryptosystems is only just beginning. These latter fields of research, as we have hinted in this paper, can be expected to contribute significantly to the performance improvements required to make homomorphic encryption-based computations a practical reality.

REFERENCES

[1] 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.

[2] 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.

[3] D. Boneh, C. Gentry, S. Halevi, and F. Wang. Private database queries using somewhat homomorphic encryption. In *11th International Conference on Applied Cryptography and Network Security - ACNS 2013*, 2013.

[4] 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.

[5] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *CRYPTO*, pages 868–886, 2012.

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

[7] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference*, volume 6841, page 501, 2011.

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

[9] 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.

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

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

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

[13] C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, pages 129–148, 2011.

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

[15] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, pages 465–482, 2012.

[16] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, pages 1–23, 2010.

[17] 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.

[18] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), 2009.

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

[20] 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.

[21] 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.

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

[23] 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.