Hardware/Software co-Design of an Accelerator for FV Homomorphic Encryption Scheme using Karatsuba Algorithm

Vincent Migliore, Maria Méndez Real, Vianney Lapotre, Arnaud Tisserand, Caroline Fontaine, Guy Gogniat

Abstract—Somewhat Homomorphic Encryption (SHE) schemes allow to carry out operations on data in the cipher domain. In a cloud computing scenario, personal information can be processed secretly, inferring a high level of confidentiality.
For many years, practical parameters of SHE schemes were overestimated, leading to only consider the FFT algorithm to accelerate SHE in hardware. Nevertheless, recent work demonstrates that parameters can be lowered without compromising the security [1]. Following this trend, this work investigates the benefits of using Karatsuba algorithm instead of FFT for the Fan-Vercauteren (FV) Homomorphic Encryption scheme. The proposed accelerator relies on an hardware/software co-design approach, and is designed to perform fast arithmetic operations on degree 2560 polynomials with 135 bits coefficients, allowing to compute small algorithms homomorphically. Compared to a functionally equivalent design using FFT, our accelerator performs an homomorphic multiplication in 11.9 ms instead of 15.46 ms, and halves the size of logic utilization and registers on the FPGA.

Index Terms—Homomorphic Encryption, SHE, FV, Hardware/software co-design, FPGA, Karatsuba, implementation.

1 Introduction

Homomorphic Encryption schemes are considered as promising in modern cryptography, because they directly allow to carry out operations on data in the cipher domain. Figure 1 illustrates a basic client/server transaction in an homomorphic scenario. The most flexible ones, called *Fully Homomorphic Encryption* (FHE) schemes, are able to process unlimited additions and multiplications secretly, and so make possible to address a wide range of algorithms. To reduce computation times, many applications only consider *Somewhat Homomorphic Encryption* (SHE) schemes, which bound the number of operations to reduce the complexity.
While classical cryptographic schemes have sometimes homomorphic properties, for addition [2] or multiplication [3] operations, it has been necessary to wait until 2009 and C. Gentry [4] breakthrough to discover a way to perform both types of operations with limited restrictions. He provided an SHE scheme based on hard lattice problems, and then turned it into an FHE scheme by using the bootstrapping technique. But, due to the bootstrapping cost, FHE schemes are considered not so practical compared to SHE schemes.
To reduce the complexity of Homomorphic Encryption, FHE/SHE schemes have been successfully adapted to related problems. The most popular ones are based on the *Approximate-Great Common Divisor* (a-GCD) problem [5][6], NTRU problem [7][8] and *Ring-Learning With Error* (R-LWE) problem [9][10][11][12][13][14]. In the following, we only consider SHE schemes with polynomial arithmetic, that is to say NTRU-based and R-LWE-based SHE schemes. Until recently, the most promising SHE scheme with polynomial arithmetic was YASHE′ [7], closely followed by FV [11]. However, the recent so called subfield attack [15] considerably reduced YASHE′ security, in particular the *Decision Small Polynomial Ratio* (DSPR) assumption [16] on which the security of the scheme relies on. Thus, FV regains interest because it does not suffer of the weakness of DSPR. Nevertheless, its ciphertext are double size when compared to those of YASHE′, because the ciphertext is composed by 2 polynomials instead of 1 for YASHE′. Due to its practicability, previous hardware implementations of SHE schemes with polynomial arithmetic targeted YASHE′ [17][18], inferring a lack of hardware implementations of FV. To our knowledge, only software implementations of FV have been proposed. In [19], an implementation of FV using the multipurpose FLINT library [20] performs an homomorphic multiplication of FV in 148 ms for degree 4096 polynomials with 125 bits coefficients. Then, FV has been implemented to

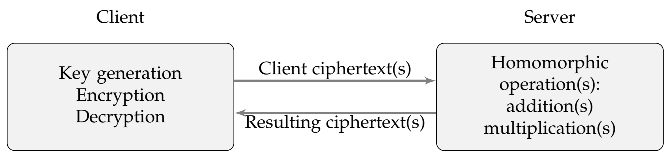


Fig. 1: Presentation of client/server transactions in an homomorphic encryption scenario.

- V. Migliore, M. Méndez Real, V. Lapotre and G. Gogniat are with the Univ. Bretagne-Sud, UMR CNRS 6285, Lab-STICC, F-56100 Lorient, France.
E-mail: firstname.lastname@univ-ubs.fr
- C. Fontaine is with the CNRS and Institut Mines-Telecom / Telecom Bretagne, UMR CNRS 6285, Lab-STICC, Brest, France.
E-mail: caroline.fontaine@telecom-bretagne.eu
- A. Tisserand is with the CNRS - IRISA - Univ. Rennes 1, Lannion, France.

Manuscript received June 20, 2016; revised October 28, 2016.

NFLlib [21], an efficient C++ implementation of ideal lattice cryptography. Authors can perform an homomorphic multiplication in 17.2 ms for the same parameters. Finally, work in [22] proposes to avoid the multi-precision arithmetic required by FV by using a *Residue Number Systems* (RNS) variant of FV. Authors report an homomorphic multiplication of FV for degree 4096 polynomials with 168 bits coefficients in 7.68 ms. However, this implementation has some limitations compared to the proposed accelerator that will be discussed in Section 5.2.

Due to the proximity between YASHE′ and FV, all previous hardware implementations of YASHE′ are still relevant but need to be adapted. Thus, timings for hardware acceleration of the homomorphic multiplication of YASHE′ are not directly comparable to timings provided earlier on software implementations of FV. Hardware accelerators for YASHE′ implement fast arithmetic of degree $n \in [4096, 32768]$ polynomials with coefficients of size $\log_2 q \in [125, 1228]$, depending on the required security and the complexity of the algorithm to be homomorphically performed. To our knowledge, all implementations are based on FFT algorithm [23]. In [17], a classical but optimized FFT implementation is presented for two parameter sets. The proposed accelerator performs an homomorphic multiplication in 6.5 ms for n = 4096 and $\log_2 q$ = 125 bits, and 48 ms for parameters n = 16384 and $\log_2 q$ = 512 bits. Authors of [17] implemented 512×512 bits multipliers with a small modular reduction by selecting a Solinas prime modulus [24]. Due to the size of polynomials and coefficients, a cache is implemented to connect the external memory used to store intermediate coefficients. They also reported a bottleneck due to the divide and rounding required by YASHE′, especially for large integers. That is why in [18] a pre-computation is performed on polynomials to reduce the size of coefficients. They split a ciphertext into a few polynomials by using the *Chinese Reminder Theorem* (CRT) on each coefficient. The overall architecture is based on an array of crypto-units, which gives some flexibility to process several residue polynomials in parallel. For parameters n = 32768 and $\log_2 q$ = 1228 bits, their accelerator performs an homomorphic multiplication in 121 ms including 25 ms spent for CRT.

Due to the security issue on YASHE′, this paper proposes to accelerate the FV scheme in hardware using Karatsuba algorithm [25]. To our knowledge, this is the first use of Karatsuba for R-LWE based SHE schemes, and the first hardware accelerator dedicated to FV scheme. Compared to previous work using Karatsuba for polynomial multiplication, for example elliptic curve and paring-based cryptography, this work investigates polynomials with much larger degrees and with arbitrary size coefficients. Our accelerator implements fast polynomial arithmetic for degree 2560 polynomials with coefficients of size 125 bits, allowing homomorphic circuits of depth up to 4. This choice is motivated by the fact that for lower depths, alternatives exist and in particular the BGN-Based scheme in [26]. We demonstrate that for Homomorphic Encryption with low multiplicative depth circuits, Karatsuba can be a good alternative to FFT. We also evaluate the scalability and the limits of our approach compared to the FFT. In order to fairly compare our approach with previous works, we propose an hardware implementation on DE5-net platform from Terasic embedding an Altera Stratix V FPGA. However, our hardware accelerator does not require such a large FPGA and can possibly be implemented on smaller ones.

The main contributions of this work are as follows:

- A complete study of Karatsuba algorithm adaptation to SHE.
- An end to end solution for accelerating FV with a hardware/software co-design using Karatsuba algorithm.
- A latency-efficient software implementation of Karatsuba algorithm.
- A lightweight Karatsuba hardware accelerator.

This paper is organized as follows. Section II recaps some key information on SHE cryptosystems based on a R-LWE problem. Section III draws some optimizations of FV with Karatsuba algorithm. Section IV details the proposed architecture and provides both software and hardware implementation details. Section V provides several discussions on our Karatsuba approach, in particular the scalability of the design. Section VI draws some conclusions.

2 THEORETICAL BACKGROUND

2.1 Notation

In the following, a polynomial is represented with an uppercase and its coefficients with a lowercase. For polynomial A, a_i represents its i^{th} coefficient. A vector of polynomials is noted in bold. For vector $\mathbf{A}$, $\mathbf{A}[i]$ is the i^{th} polynomial of the vector. For set R and polynomial A, $A \leftarrow U_R$ represents a uniformly sampled polynomial in R and $A \leftarrow \chi_\sigma$ a polynomial sampled in a discrete Gaussian distribution with standard deviation σ. For coefficient a_i of polynomial A, $a_{i,(j..k)}$ corresponds to the binary string extraction of a_i between bits j and k. This notation is extended to polynomial A where $A_{(j..k)}$ is the sub-polynomial where the binary string extraction is applied to each coefficient. Other standard operators are represented as follows.

A modular reduction by an integer q is noted $[\cdot]_q$. For integer a, $\lfloor a \rfloor$, $\lceil a \rceil$ and $\lfloor a \rceil$ operators are respectively the floor, ceil and nearest rounding operations. This is extended to polynomials by applying the operation on each coefficient. For vectors $\mathbf{A}$ and $\mathbf{B}$, $\langle \mathbf{A}, \mathbf{B} \rangle$ represents $\sum \mathbf{A}[i]\mathbf{B}[i]$.

2.2 Karatsuba Algorithm

Karatsuba algorithm is an improvement of the classical polynomial multiplication algorithm which reduces the number of sub-products.

For simplicity, we only address polynomials with an even number of coefficients, but Karatsuba can be easily adapted to odd ones by manipulating unbalanced sub-polynomial multiplications, or by using zero padding. Input polynomials A and B of degree $n-1$ are split into two parts of equivalent size, that is to say $\frac{n}{2}$ coefficients. Let A_H and A_L be two polynomials composed respectively by the coefficients of highest degree of A and lowest degree of A. By the same way, one constructs B_H and B_L. Input

polynomials are now expressed as $A = A_L + A_H x^{n/2}$ and $B = B_L + B_H x^{n/2}$.
When multiplying A and B by the standard approach, the resulting decomposition is given by:

$$\begin{aligned} A \times B &= (A_L + A_H x^{n/2})(B_L + B_H x^{n/2}) \\ &= A_L B_L + (A_L B_H + A_H B_L)x^{n/2} + A_H B_H x^n \end{aligned}$$

Karatsuba optimization is based on noticing that the middle factor $(A_L B_H + A_H B_L)$ can be cleverly computed by $(A_L + A_H)(B_L + B_H) - A_L B_L - A_H B_H$. As one can quote, $A_L B_L$ and $A_H B_H$ are already computed and so it does not require additional multiplications.
At the end, Karatsuba requires 3 sub-polynomial multiplications instead of 4, at a cost of two pre-computations, namely $(A_H + A_L)$ and $(B_H + B_L)$, and two post-computations for the reconstruction of the middle factor. However, these pre- and post-computations are made of additions and subtractions only. To further reduce the complexity of the polynomial multiplication, one can apply Karatsuba algorithm on each sub-polynomials multiplication, that is to say $A_L B_L$, $A_H B_H$ and $(A_H + A_L)(B_H + B_L)$. The number of times Karatsuba algorithm has been applied recursively is called number of Karatsuba recursions.
After several Karatsuba recursions, one has to perform many low degree polynomials multiplications instead of a large polynomial multiplication. This recursiveness allows sharing computations between the software and the hardware. For example, several recursions can be performed in software and the remaining ones in hardware.
Because each Karatsuba recursion halves the size of sub-polynomials, Karatsuba can achieve polynomial multiplication of degree $2^r(p+1) - 1$, where r is the number of Karatsuba recursions and p the degree of the smallest sub-polynomial.

2.3 R-LWE Problem

A R-LWE instance is constructed in the ring $\mathbb{Z}_q[X]/\langle f(X)\rangle = R_q$, where $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ and $f(X)$ is an irreducible degree n polynomial in $\mathbb{Z}_q[X]$. Resolving a R-LWE problem consists on recovering a polynomial S from the pair $(AS + E, A)$, where $S \leftarrow \chi_{key}$, $A \leftarrow U_{R_q}$ and $E \leftarrow \chi_{err}$. If χ_{key} and χ_{err} are cleverly chosen, the R-LWE pair is mostly indistinguishable from the uniform distribution and its resolution is considered as hard as worst-case lattice problems. Usually, S is chosen from a binary set, and χ_{err} with a standard deviation $\sigma_{err} > 2\sqrt{n}$.

2.4 Cryptosystem FV

FV is a transposition of the scale-invariant Brakerski scheme [10] into the R-LWE problem. Let λ be the security parameter that determines $(q, n) \in \mathbb{Z}^2$, the parameters of a R-LWE instance. Let $t \in \mathbb{Z}$ with $1 < t < q$ be an integer which provides the upper bound of a message size, and $\omega \in \mathbb{Z}_q$ that splits an element of $\mathbb{Z}_q$ into $l_{w,q} = \lceil \log_2 q / \log_2 w \rceil$ elements. Figure 2 presents elementary primitives of FV in a flowchart style, and introduces the notations of all operations used below.
In FV, the public key is a pair $(AS + E, A)$ of a R-LWE instance, and the secret key the polynomial S. This setup inevitably introduces an error term E called noise. During computations, the noise will grow until possibly making the decryption procedure faulty. An homomorphic addition is considered not critical because the noise is just added. For the homomorphic multiplication, the noise is multiplied, and infers a limitation on the number of operations achievable. Because the noise is mostly lead by the number of multiplications performed on a ciphertext, namely the multiplicative depth L, the impact of the homomorphic addition is usually neglected. In practice, this impact can possibly reduce the multiplicative depth if significant homomorphic additions are performed. Table 1 provides some parameters for FV extracted from [19] satisfying a security level λ of 80 bits. In particular, we used Equation (2) to calculate the upper bound of the modulus for a given degree and security level, and equations in Section 3.5 to extract the multiplicative depth for a given set of parameters. We also set ω to 27 bits to efficiently use hardware resources of the Stratix V. Additional information are given in Table 1, that will be discussed in Section 2.5.
While an homomorphic addition is just a polynomial addition of ciphertexts, an homomorphic multiplication requires an extra step after the polynomial multiplication called *relinearization*.
To understand why a relinearization step is required, it

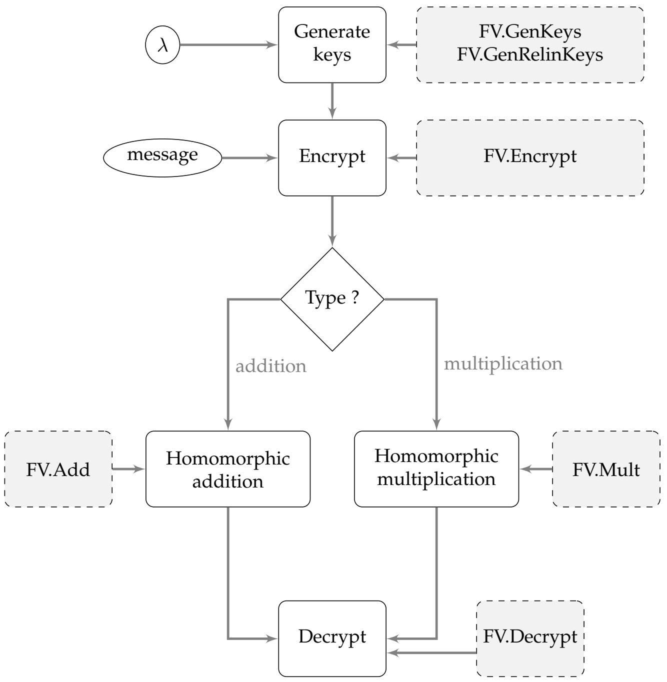


Fig. 2: Flowchart of a basic operation in FV.

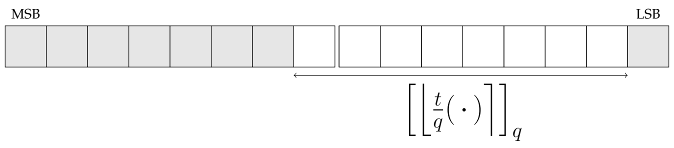

MSB
LSB
$\left[\left\lfloor \frac{t}{q}(\cdot) \right\rceil\right]_q$

Fig. 3: Bits to be extracted for the $\left[\left\lfloor \frac{t}{q}(\cdot) \right\rceil\right]_q$ operation in FV when t = 2 and q a power of two.

TABLE 1: Parameters for FV extracted from [19] satisfying a security level λ of 80 bits, with ω set to 27 bits. Required n for both NWC and classical FFT is also provided.

L	$\log_2 q$	n	n for NWC	n for classical FFT
1	48	904	1024	2048
2	73	1428	2048	4096
3	98	1951	2048	4096
4	125	2515	4096	8192
5	152	3077	4096	8192
6	179	3636	4096	8192
7	207	4215	8192	16384
8	235	4792	8192	16384
9	264	5388	8192	16384
10	293	5982	8192	16384

is important to notice that a ciphertext is proportional to the secret key S, plus an error. When multiplying two ciphertexts, the resulting polynomial is of the form of $A + BS + CS^2$, proportional to S^2. To continue homomorphic operations, the ciphertext needs to recover its initial form, and thus the knowledge of S^2 is required on the server-side, which is not acceptable for security purposes. Instead of manipulating S^2 directly, a sub R-LWE instance is created in order to hide S^2.

However, creating a sub-instance of R-LWE introduces an error term, which will penalize the multiplicative depth. To address this issue, several optimizations are performed during the relinearization step based on two functions, FV.PowersOf$_{w,q}$ and FV.WordDecomp$_{w,q}$:

```
• FV.PowersOf_{w,q}(A) :
    $\mathbf{A} \in R_q^{l_{w,q}}$
    for $i$ in 0 to $l_{w,q} - 1$
        $\mathbf{A}[i] = \left[Aw^i\right]_q$
    end for
    return $\mathbf{A}$
```

```
• FV.WordDecomp_{w,q}(A) :
    $\mathbf{A} \in R_q^{l_{w,q}}$
    for $i$ in 0 to $l_{w,q} - 1$
        $l_0 = i \times \log_2 \omega$
        $l_1 = (i+1) \times \log_2 \omega - 1$
        $\mathbf{A}[i] = A_{(l_0..l_1)}$
    end for
    return $\mathbf{A}$
```

$$\langle \text{FV.PowersOf}_{w,q}(A), \text{FV.WordDecomp}_{w,q}(B) \rangle = \left[A \times B\right]_q .$$

By cleverly using FV.WordDecomp$_{w,q}$, one can perform a scalar product with sub-polynomials with coefficients of size $\log_2 w$ instead of $\log_2 q$, and in the context of FV, multiply the error polynomial E by a polynomial with coefficients of size $\log_2 w$ instead of $\log_2 q$.

All primitives of FV are as follows:

```
• FV.GenKeys($\lambda$) :
    $S \leftarrow \chi_{key}$ , $A \leftarrow U_{R_q}$ , $E \leftarrow \chi_{err}$
    $P_{key} = (-AS + E, A)$
    $S_{key} = S$
    return $(P_{key}, S_{key})$

• FV.GenRelinKeys($P_{key}, S_{key}$) :
    $\mathbf{A} \leftarrow U_{R_q}^{l_{w,q}}$, $\mathbf{E} \leftarrow \chi_{err}^{l_{w,q}}$
    $\gamma = \left(\left[\text{FV.PowersOf}_{w,q}(S_{key}^2) - (\mathbf{A}S_{key} + \mathbf{E})\right]_q, \mathbf{A}\right)$
    return $\gamma$

• FV.Encrypt($m, P_{key}$) :
    $U \leftarrow \chi_{key}$ , $(E_1, E_2) \leftarrow \chi_{err}$
    $C = \left(\left[\frac{q}{t}m + P_{key}[0]U + E_1\right]_q, \left[P_{key}[1]U + E_2\right]_q\right)$
    return $C$

• FV.Decrypt($C, S_{key}$) :
    $\widetilde{M} = \left[C[0] + C[1]S_{key}\right]_q$
    $m = \left\lfloor \frac{t}{q}\widetilde{M}[0] \right\rceil$
    return $m$

• FV.Add($C_A, C_B$) :
    $C_+ = \left(\left[C_A[0] + C_B[0]\right]_q, \left[C_A[1] + C_B[1]\right]_q\right)$
    return $C_+$

• FV.Mult($C_A, C_B, \gamma$) :
    $\widetilde{C}_0 = \left[\left\lfloor \frac{t}{q} C_A[0] \times C_B[0] \right\rceil\right]_q$
    $\widetilde{C}_1 = \left[\left\lfloor \frac{t}{q} \left(C_A[0] \times C_B[1] + C_A[1] \times C_B[0]\right) \right\rceil\right]_q$
    $\widetilde{C}_2 = \left[\left\lfloor \frac{t}{q} C_A[1] \times C_B[1] \right\rceil\right]_q$
    $C_\times = \text{FV.Relin}(\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \gamma)$
    return $C_\times$

• FV.Relin($\widetilde{C}_0, \widetilde{C}_1, \widetilde{C}_2, \gamma$) :
    $C_R = (C_{R,0}, C_{R,1})$
    $C_{R,0} = \left[\widetilde{C}_0 + \left\langle \text{FV.WordDecomp}_{w,q}(\widetilde{C}_2), \gamma[0] \right\rangle\right]_q$
    $C_{R,1} = \left[\widetilde{C}_1 + \left\langle \text{FV.WordDecomp}_{w,q}(\widetilde{C}_2), \gamma[1] \right\rangle\right]_q$
    return $C_R$
```

As one can see, all FV primitives are based on a few polynomial additions and multiplications. This is why speeding up polynomial multiplication is a good choice.

2.5 Choosing Parameters

In SHE, polynomial multiplication is typically implemented with FFT algorithm. To be efficient, FFT must be generated by a polynomial with irreducible factors of very small degree. That is why $x^n - 1$ and $x^n + 1$ are often chosen because they can be completely factorized with degree 1 factors. Moreover, because $x^n + 1$ is also a cyclotomic polynomial, this method provides a solution where the polynomial reduction is directly integrated into the computation. This special FFT is called *Negative Wrapped Convolution* (NWC) and requires a FFT of size n instead of $2n$ in the standard case. However, this cyclotomic has an important issue: When factoring $x^n + 1$ modulo 2, the resulting polynomial is $(x + 1)^n$, which has a unique factor, namely $(x + 1)$. This is incompatible with the CRT on polynomials because this latter requires factors with different polynomials. Thus, the NWC, which is optimized for performance, cannot pack several messages inside one ciphertext using CRT. This technique allows to perform the same homomorphic operation on each message in parallel, and is called batching. For further explanation on how to use CRT in the context of *Homomorphic Encryption*, reader can refer to [27].

Because we address the problem with Karatsuba algorithm, we have no particular restriction on input polynomials, and we can choose a cyclotomic polynomial with batching capabilities. However, for Karatsuba efficiency, polynomials with a degree of $2^i p$ are preferable, $(i, p) \in \mathbb{Z}^2$. As it can be noticed in Table 1, many multiplicative depths require n to be relatively distant from a power of two. Critical cases are when n is just above a power of two, like for a multiplicative depth of 4 and 7, where FFT is inefficient. To the best of our knowledge, no particular lack of security has been demonstrated on the modulus of R-LWE instances, thus we set it up to a power of two. Usually, q is prime due to FFT.

In order to demonstrate the interest of the proposed approach based on Karatsuba, we choose a multiplicative depth of 4, which corresponds to a parameter set of $n = 2515$ and $\log_2 q$ = 125 bits.

To be as close as possible to the required n, we set the smallest sub-polynomial to degree 4, with 9 recursions of Karatsuba. That allows a polynomial multiplication of degree at most $2^i p - 1 = (4+1) \cdot 2^9 - 1 = 2559$. Thus, the associated irreducible polynomial can be selected in the range $[2515, 2560]$. For $n = 2560$, one can find a cyclotomic polynomial with 5 coefficients, and thus the polynomial reduction can be fastly implemented. If one wants batching capabilities, setting n to 2560 allows to pack at least 64 bits in a ciphertext in a batching fashion.

3 Proposed Optimizations

Proposed optimizations focus on two FV primitives: FV.Mult and FV.Relin. Even if accelerating the polynomial multiplication impacts all FV primitives, an homomorphic server will mainly perform homomorphic additions and multiplications. Accelerating polynomial addition is also possible, but is not relevant compared to the complexity of a polynomial multiplication.

3.1 FV.Mult

Referring to FV.Mult, a rounding is required after polynomial multiplication. This operation can be time consuming for FFT implementations because the modulus has to be prime. For Karatsuba, it can be set to a power of two and thus the $\left\lfloor \frac{t}{q}(\,\cdot\,) \right\rceil$ operation becomes very simple, corresponding to a shift of $\log_2 q - 2$ bits. In parallel, one can also execute the modular reduction to further optimize the operation. Finally, computing $\left[\left\lfloor \frac{t}{q}(\,\cdot\,) \right\rceil \right]_q$ is equivalent to extracting several bits, as shown in Figure 3.

The 4 polynomial multiplications in FV.Mult can also be reduced to 3 ones with the help of Karatsuba algorithm. In fact, computations of $\widetilde{C}_0$, $\widetilde{C}_1$ and $\widetilde{C}_2$ can be seen as a computation of sub-factors of Karatsuba. By that way, the polynomial $\widetilde{C}_1$ can be expressed as:

$$\widetilde{C}_1 = \frac{t}{q}\Bigg(\Big(C_A[0] + C_A[1]\Big) \times \Big(C_B[0] + C_B[1]\Big)\Bigg) - \widetilde{C}_0 - \widetilde{C}_2;$$

3.2 FV.Relin

FV.Relin requires $l_{w,q}$ degree $n-1$ polynomial multiplications, with sub-products of size $\log_2 w \times \log_2 q$ bits, and $l_{w,q} - 1$ degree $n-1$ polynomial additions. Because $\log_2 q \times \log_2 q = (l_{w,q} \times \log_2 w) \times \log_2 q$, the number of elementary operations between the polynomial multiplication and the relinearization step are theoretically equivalent. However, FFT algorithm cannot be optimized in that way because coefficients are in a different space. Thus, sub-polynomial multiplications must be performed separately.

Because Karatsuba algorithm does not have such a limitation, it is possible to modify the architecture to perform the relinearization step with limited modifications. The optimization relies on two properties:

1 Product/accumulation can be performed on coefficients instead of sub-polynomials.

2 Polynomial product/accumulation required by the relinearization step can be performed on sub-polynomials.

Assertion 1 can be easily demonstrated by writing the definition of the standard polynomial multiplication algorithm:

$$\mathbf{A} = \text{FV.WordDecomp}_{w,q}(A)$$

for a given set of sub-polynomials $\mathbf{C}[i] = \mathbf{A}[i] \times B$.

$$\begin{aligned}\sum_{i=1}^{l_{w,q}} \mathbf{C}[i]_k &= \sum_{i=1}^{l_{w,q}} \underbrace{\sum_{j=0}^{k} \mathbf{A}[i]_j B_{k-j}}_{\text{Standard multiplication}} \\ &= \sum_{j=0}^{k} \underbrace{\sum_{i=1}^{l_{w,q}} \mathbf{A}[i]_j B_{k-j}}_{\substack{\text{Integer multiplication} \\ \text{accumulation}}}\end{aligned}$$

As one can see, because each sum is independent, one can swap product/accumulation at coefficient level.

For Assertion 2, it is required to expand the product/accumulation with Karatsuba:

$$\begin{aligned} C &= \Big\langle \mathbf{A}, B \Big\rangle \\ &= \sum_{i=1}^{l_{w,q}} \mathbf{A}[i] B \\ &= \sum_{i=1}^{l_{w,q}} \Big[\mathbf{A}_L[i] B_L + \mathbf{A}_H[i] B_H x^n \\ &+ \Big((\mathbf{A}_L[i] + \mathbf{A}_H[i])(B_L + B_H) \\ &- \mathbf{A}_L[i] B_L - \mathbf{A}_H[i] B_H \Big) x^{n/2} \Big] \\ &= \sum_{i=1}^{l_{w,q}} \mathbf{A}_L[i] B_L + x^n \sum_{i=1}^{l_{w,q}} \mathbf{A}_H[i] B_H \\ &+ x^{n/2} \sum_{i=1}^{l_{w,q}} (\mathbf{A}_L[i] + \mathbf{A}_H[i])(B_L + B_H) \\ &- x^{n/2} \sum_{i=1}^{l_{w,q}} \mathbf{A}_L[i] B_L - x^{n/2} \sum_{i=1}^{l_{w,q}} \mathbf{A}_H[i] B_H \end{aligned}$$

4 Implementation

The complete implementation relies on a hardware/software co-design approach. The software runs a complete SHE library and deports some specific polynomial multiplications to the FPGA when needed. The flow chart in Figure 5 shows how the different operations are dispatched between hardware and software. The proposed architecture is composed of a CPU Intel core i7-4910MQ with 4 cores running at 2.9 GHz, connected to the hardware platform through a PCIe 3.0 with 8 lanes. The hardware platform embeds a powerful FPGA, a stratix V GX from Altera. Our accelerator is designed to fully speed-up both FV.Mult and FV.Relin, but can accelerate any operation which requires polynomial multiplication, that is to say almost all steps of FV. However, this study only focuses on FV.Mult and FV.Relin.

4.1 Software Implementation Details

The software part of FV implementation is performed using NTL library [28] compiled with GMP [29] support, 64 bits version with -O3 option. We also ported FLINT [20] cyclotomic calculation to NTL, in order to process at runtime any cyclotomic polynomial. Because the bottleneck of SHE schemes is the homomorphic multiplication, much effort has been done to optimize pre- and post-computations of Karatsuba. The pseudo-code for pre- and post-computations are provided in Algorithms 1 and 2. The starting point of both algorithms is respectively PRE_COMPUTATION$(r, 0)$ and POST_COMPUTATION$(r, 0)$, where r is the number of Karatsuba recursions.

Algorithm 1 Karatsuba Pre-Computation

```
 1: procedure PRE_COMPUTATION($r, f$)
 2:     $i \leftarrow r \,,\, j \leftarrow f$
 3:     $i_0 \leftarrow i - 1$
 4:     $j_0 \leftarrow 3j + 0 \,,\, j_1 \leftarrow 3j + 1 \,,\, j_2 \leftarrow 3j + 2$
 5:     if $i > 0$ then
 6:         $P_{i_0,j_0} \leftarrow P_{i,j}\left[{\scriptstyle n/2^{i+1}\cdots 0}\right]$
 7:         $P_{i_0,j_1} \leftarrow P_{i,j}\left[{\scriptstyle n/2^{i}\cdots n/2^{i+1}+1}\right]$
 8:         $P_{i_0,j_2} \leftarrow P_{i_0,j_0} + P_{i_0,j_1}$
 9:         PRE_COMPUTATION($i_0, j_0$)
10:         PRE_COMPUTATION($i_0, j_1$)
11:         PRE_COMPUTATION($i_0, j_2$)
12:     end if
13: end procedure
```

Table 2 provides latencies of software pre- and post-computations with various optimizations.

Each line of the pre-computation section in Table 2 represents latency for one polynomial, and must be multiplied by two for a complete evaluation of a pre-computation cost during a polynomial multiplication. At the opposite, the “fully-threaded” approach in line 7 processes two pre-computations at a time with 6 threads in parallel. For a deeper analysis of software performances, results are provided for two different Karatsuba recursions.

As one can see in lines 1 and 2 in Table 2, pre-allocation of sub-polynomials is crucial due to the size of ciphertexts.

Algorithm 2 Karatsuba Post-Computation

```
 1: procedure POST_COMPUTATION($r, f$)
 2:     $i \leftarrow r \,,\, j \leftarrow f$
 3:     $i_0 \leftarrow i - 1$
 4:     $j_0 \leftarrow 3j + 0 \,,\, j_1 \leftarrow 3j + 1 \,,\, j_2 \leftarrow 3j + 2$
 5:     if $i > 1$ then
 6:         POST_COMPUTATION($i_0, j_0$)
 7:         POST_COMPUTATION($i_0, j_1$)
 8:         POST_COMPUTATION($i_0, j_2$)
 9:     end if
10:     $P_{i,j}\left[{\scriptstyle n/2^{i}-1\cdots 0}\right] \leftarrow P_{i_0,j_0}$
11:     $P_{i,j}\left[{\scriptstyle n/2^{i-1}-1\cdots n/2^{i}}\right] \leftarrow P_{i_0,j_1}$
12:     $P_{i,j}\left[{\scriptstyle 3n/2^{i+1}-1\cdots n/2^{i+1}}\right] \leftarrow P_{i_0,j_2} - P_{i_0,j_0} - P_{i_0,j_1}$
13: end procedure
```

Because Karatsuba is constructed with recursive calls (lines 9, 10, 11 of Algorithm 1), there is no data dependencies between calls. Thus, multi-threading can possibly be used to reduce latency. Based on our experiments, applying multi-threading beyond the first recursion of Karatsuba is counterproductive. Moreover, threading each sub-calls of PRE_COMPUTATION (line 6 in Table 2) is not efficient.

To further reduce pre-computations latency, an optimized version of the presented algorithm has been designed and implemented, which reduces at the same time pre-allocation and latency. If one carefully examines lines 6 and 7 of Algorithm 1, these steps duplicate a given polynomial into two sub-polynomials. This is counterproductive because no operation is performed. To avoid this issue, we added a few extra parameters to the PRE_COMPUTATION function in order to give the index and the number of coefficients instead of duplicating them. Figure 4 presents the proposed optimization, where dashed polynomials represent polynomials which are duplicated during the initial algorithm. This strategy saves 66% of memory and reduces latency by 13% compared to the basic one (line 2 and 4 in Table 2). Finally, the whole optimizations reduce computation time by 62% for the pre-computation.

We also try various optimizations to reduce computation time of post-computations, however no particular method has given sufficient results to be implemented in the final design, except multi-threading which is very efficient for this step. This implies that extra post-computations in hardware may be a good alternative to improve performances of the overall acceleration.

4.2 Hardware Implementation Details

Our hardware implementation of Karatsuba is based on a DE5-450 Terasic platform with an Altera Stratix V GX (5SGXEA7N2F45C2) FPGA. The DE5 platform is plugged as a peripheral of a computer and communication between the software and the hardware is done through PCIe.

The advantage of Karatsuba algorithm is that it can be scaled upon the available hardware resources. If one has limited hardware resources and can only compute polynomial of relatively small degree, the software part can compute extra pre- and post-computations at a cost of a higher computation time. However, if too many pre- and post-computations are performed in software, total computation

TABLE 2: Software performances for pre- and post-computations. Computations are executed on a Intel core i7-4910MQ with 4 cores running at 2.9GHz.

	Setup (n,$\log_2 q$)	(2560, 125)	
	Karatsuba recursions	5	6
pre	Dynamic allocation	1.97 ms	7.25 ms
	Pre-allocation	338 µs	546 µs
	Threaded (×2)	263 µs	409 µs
	Optimized (Opt.)	285 µs	480 µs
	Opt. and threaded (×2)	213 µs	336 µs
	Opt. and threaded (×3)	285 µs	366 µs
	fully-threaded (×6)	406 µs	488 µs
post	Basic	1.89 ms	2.94 ms
	Threaded	931 µs	1.37 ms

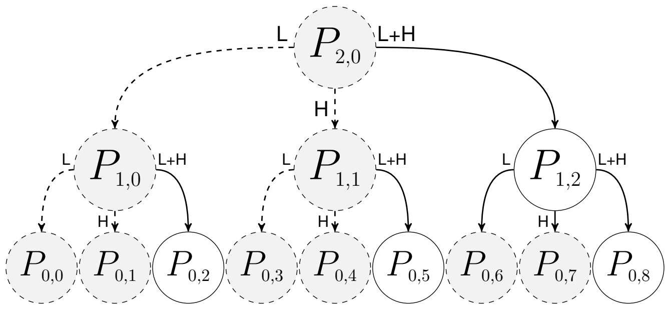


Fig. 4: Data dependencies on Karatsuba pre-computations. Dashed circles represent polynomials that can be extracted from the input polynomial directly instead of solid lined ones which require a polynomial addition.

time can become higher than a pure software polynomial multiplication. As stated before, for $n = 2560$, our Karatsuba setup requires 9 Karatsuba recursions, with smaller sub-polynomials of degree 4. In order to be competitive, 6 recursions are made in software, and the 3 remaining in hardware. With this setup, $2 \cdot 3^6 = 729$ sub-polynomials of degree $2560/2^6 - 1 = 39$ are sent to the accelerator, corresponding to 29160 coefficients.

Figure 6 provides a high-level overview of the hardware accelerator, where input sub-polynomials are named P and Q. The accelerator has been designed to perform the operation on sub-polynomials as soon as they arrive. Thus, transfer latency is completely hidden during Karatsuba computations.

After the pre-computation and the pre-crossbar, the accelerator generates several lines of sub-polynomials, which are multiplied in parallel. The post-crossbar and the post-computation perform the reconstruction of the polynomial before sending it through the PCIe.

In the following, an architecture of Karatsuba with degree 3 sub-polynomials instead of 4 is presented in order to simplify the comprehension, and all intermediate pipeline stages are not represented for the same reason.

4.2.1 Bus constraint

Because the bus is based on a PCIe 3.0 with 8 lines, it can handle transfers up to 250 MB/s per line in full-duplex. By taking into account the relatively low frequency of FPGAs and their parallel capabilities, the FPGA interface can send/receive 256 bits in parallel at 250 MHz. According to

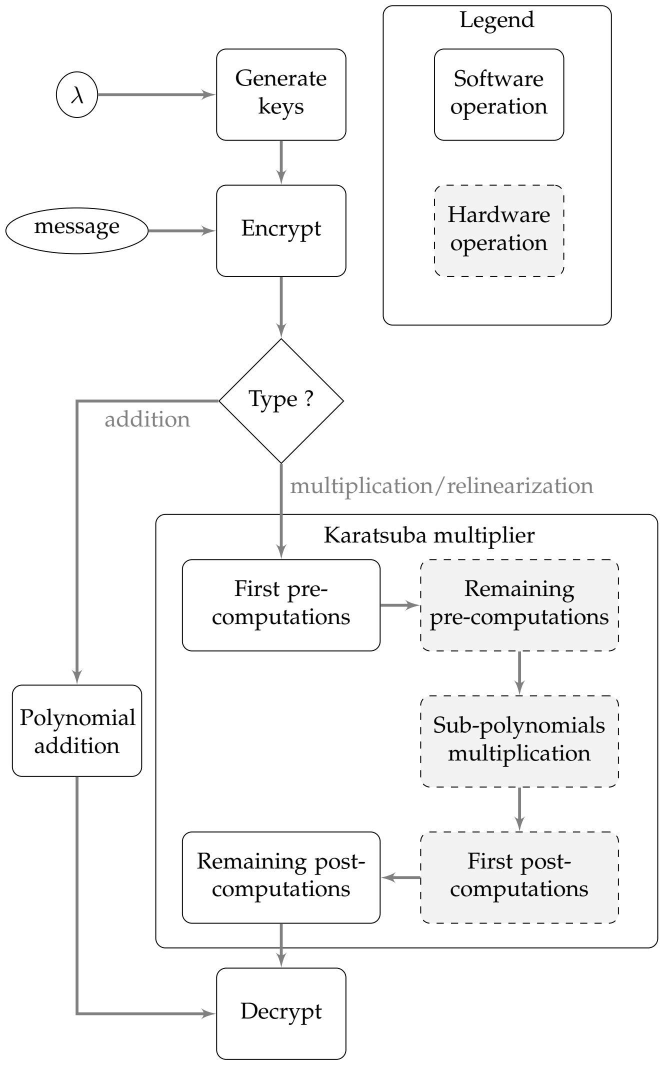


Fig. 5: Flowchart of a basic operation in our architecture.

our setup, coefficients have a size of 125 bits and so the bus size is sufficient to send two coefficients at a time. However, implementing a 125×125 bits multiplier is not efficient in practice, that is why all elementary operators have been serialized in the proposed architecture. Because embedded DSPs are optimized to perform a 27×27 bits integer product, we decided to split coefficients into 27 bits parts. This choice allows 9 inputs/outputs simultaneously and gives some flexibility to implement multiple Karatsuba operations in parallel. The remaining bandwidth is also beneficial for FV.Relin, giving the possibility to send relinearization keys γ during the transfer of the polynomial to be relinearized, avoiding to store them temporarily in hardware.

In the following, an add/subtraction/multiplication operator is considered to be serialized, with a carry propagation.

4.2.2 Pre-computation

The pre-computation step is the first block of Karatsuba accelerator and must be applied to the two input polynomials P and Q, preferably simultaneously in order to limit temporary storage of polynomials. This is also an important step because this stage determines the parallelism of the design. Our implementation is based on a recursive structure where

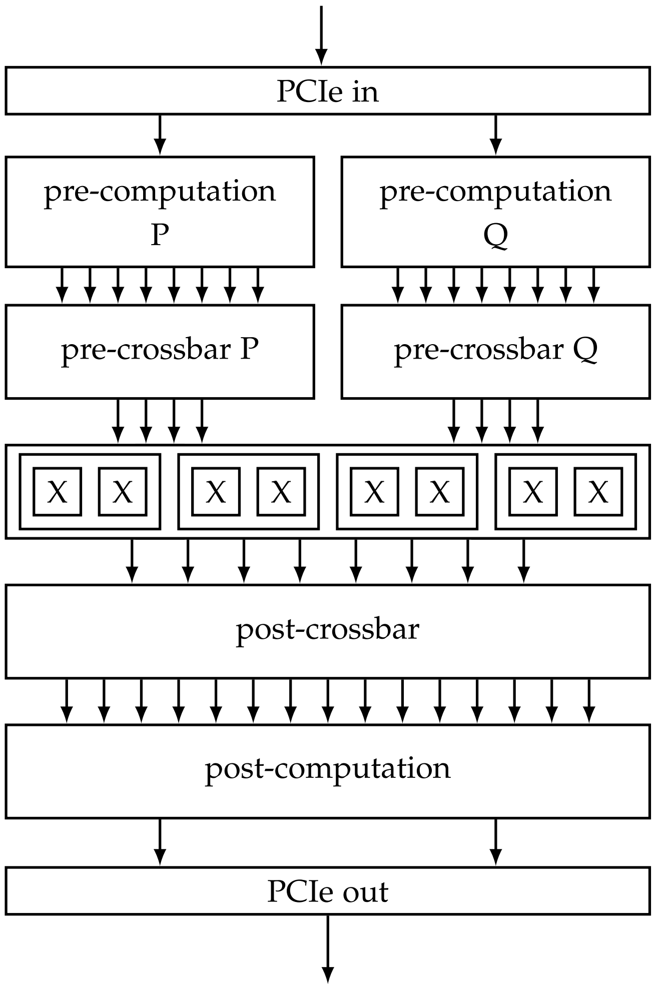
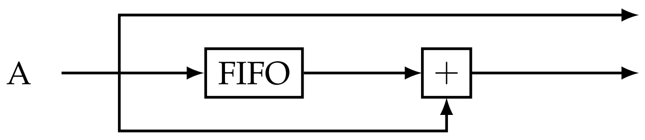


Fig. 6: Karatsuba hardware accelerator architecture overview.

Input	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Elementary operations					$a_4 + a_0$	$a_5 + a_1$	$a_6 + a_2$	$a_7 + a_3$
Output					r_0	r_1	r_2	r_3
Input	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
Elementary operations					$a_{12} + a_8$	$a_{13} + a_9$	$a_{14} + a_{10}$	$a_{15} + a_{11}$
Output					r_4	r_5	r_6	r_7

(a) Elementary operations schedule



(b) Architecture

Fig. 7: Overview of elementary pre-computation operations schedule 7(a) and the associated architecture 7(b).

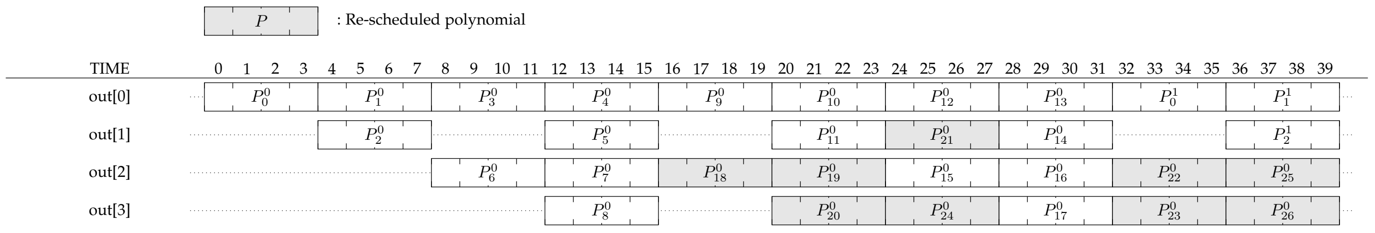
the elementary unit is visible in Figure 7(b). Because coefficients are sent by ascending order and because we need to add low order coefficients to high order coefficients, a FIFO is implemented to temporary store first arrived coefficients. This leads to two outputs: the first one is just a copy of the input and refers to polynomials P_L and P_H of Karatsuba algorithm, when the second one refers to the polynomial P_{L+H}. For the next recursion, a pre-computation needs to be applied to P_L, P_H and P_{L+H}. This can be easily performed by implementing a pre-computation unit on each output of the first unit. A minor modification of the FIFOs is required, because input polynomials are halve sized compared to the first unit. This approach has been applied once more on the proposed accelerator to implement 3 Karatsuba recursions.
As it can be easily noticed, the more recursions we perform, the more outputs the unit has and, so, the more parallelism is reached. For 1 recursion, the unit has 2 outputs, 4 outputs for 2 recursions and so on. It is also important to notice that each output creates a valid sub-polynomial which can be multiplied with the related output of the other pre-computation unit. However, because the branch P_{L+H} creates a valid polynomial only half of the time, many outputs are used inefficiently. Figure 7(a) shows this phenomenon. That is why a scheduling is implemented to reorder sub-polynomials and reduces the number of outputs, as it will be explained in Figure 8.

4.2.3 Pre-crossbar

After the pre-computations, 8 outputs are generated because 3 recursions of Karatsuba are applied, and so 8 sub-polynomial multipliers are required if no optimization is done. This infers an idleness of multipliers of $1 - \frac{3^3}{2^3 \cdot 2^3} = 58\%$ which is not efficient. The sub-polynomials reordering is performed by a simple crossbar because the pre-computation is deterministic. Figure 8 presents the strategy adopted for the scheduling of sub-polynomials, requiring 4 polynomial multipliers instead of 8. The new idleness of multipliers is $1 - \frac{3^3}{2^3 \cdot 2^2} = 16\%$ which is much more acceptable. In order to reduce even more this idleness, implementing extra pre-computations in hardware provides more flexibility to efficiently schedule sub-polynomials. Table 3 recaps the minimum number of outputs for a given number of Karatsuba recursions in hardware, according to our architecture. As it can be noticed, when sufficient pre-computations are deported to the hardware, the multipliers usage can tend toward 100%, at a cost of a complex crossbar.

4.2.4 Serial polynomial Multiplier

Implemented degree 4 polynomial multipliers are based on the standard polynomial multiplication algorithm. In order to be able to send polynomials without interruption, a full-parallel design is implemented and requires 5 serial integer multipliers in parallel. By doing that, the accelerator can benefit to the full potential of PCIe and its high throughput. Figure 9(a) presents the elementary operations required for a polynomial multiplication of degree 3. Each column of the elementary operations section represents the output of a serial integer multiplier. The polynomial multiplier itself is split into three distinct parts. First, a scheduling



P
: Re-scheduled polynomial
TIME
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
out[0]
P_0^0 P_1^0 P_3^0 P_4^0 P_9^0 P_{10}^0 P_{12}^0 P_{13}^0 P_0^1 P_1^1
out[1]
P_2^0 P_5^0 P_{11}^0 P_{21}^0 P_{14}^0 P_2^1
out[2]
P_6^0 P_7^0 P_{18}^0 P_{19}^0 P_{15}^0 P_{16}^0 P_{22}^0 P_{25}^0
out[3]
P_8^0 P_{20}^0 P_{24}^0 P_{17}^0 P_{23}^0 P_{26}^0

Fig. 8: Schedule example after a pre-computation with 3 recursions of Karatsuba.

TABLE 3: Impact of the pre-crossbar over the efficient use of embedded multipliers, and the number of sub-polynomials multipliers required in parallel.

	without scheduling		with scheduling	
Recursions	multipliers usage	required sub-polynomials multipliers in parallel	multipliers usage	required sub-polynomials multipliers in parallel
1	75 %	2	75 %	2
2	56.25%	4	75 %	3
3	42.19%	8	84.38%	4
4	31.64%	16	84.38%	6
5	23.73%	32	94.92%	8
6	17.8 %	64	94.92%	12
7	13.25%	128	94.92%	18
8	10.01%	256	98.57%	26
9	7.51%	512	98.57%	39
10	5.63%	1024	99.42%	58

of input coefficients is performed. Then, a coefficient-wise multiplication is performed using serial integer multipliers. Finally, a reconstruction step, which consists on additions between coefficients, is applied.

When sub-polynomials are sent successively, the polynomial multiplication produces output coefficients when the next input polynomials are received. This implies to carefully manage the output of serial integer multipliers in order to avoid any overlapping. To address this issue, a demultiplexer is implemented just after serial integer multipliers, and dispatches coefficients into two branches. To further reduce the complexity of the architecture, demultiplexers also set to zero their outputs when no coefficients are sent, allowing to implement very simple elementary units during the reconstruction step. Figure 9(b) shows the proposed polynomial multiplier. This architecture is very flexible and can be scaled upon the size of sub-polynomials.

4.2.5 *Serial integer multiplier*

As stated before, embedded DSPs are optimized for 27×27 bits integer multiplications, so coefficients are split in 27 bits segments. This infers that coefficients are divided into $\lceil 125/27 \rceil = 5$ parts. Similarly to the serial polynomial multiplier, serial integer multipliers are based on a standard multiplication approach. This conducts to very close architectures, as it can be shown in Figure 9(c). The main difference relies on the management of a carry propagation between intermediate coefficients.

Now, one needs to decide if the $\left\lfloor \frac{t}{q}(\cdot) \right\rceil$ operation is done at this point or later. By implementing it now, remaining computations are performed on smaller coefficients and so it reduces hardware consumption. By implementing it later, this operation can be scheduled more efficiently, or can even be deported to the software. Because this operation is very simple in our setup, the reduction is executed following the integer multiplication as it can be seen in Figure 9(b). Section IV discusses some cases where implementing the reduction just after the integer multiplication is not necessarily the best choice, especially for higher multiplicative depths.

4.2.6 *Post-crossbar*

Because a scheduling has been applied on sub-polynomials after the pre-computation, a reverse scheduling is required to realign sub-polynomials before post-computations. However, this step requires much more storage than the pre-crossbar because all sub-polynomials must be aligned with the most delayed one during the pre-crossbar. Two strategies can be used here. One consists on implementing a complex crossbar, producing directly well aligned outputs, like during the pre-crossbar. Another strategy consists on implementing successive stages of post-crossbars and post-computations in order to realign as less as possible sub-polynomials. This can lead to reduce storage requirements because many polynomials are already well aligned for a given recursion, considering that only 33% of sub-polynomials have been moved during the pre-crossbar.

4.2.7 *Post-computation*

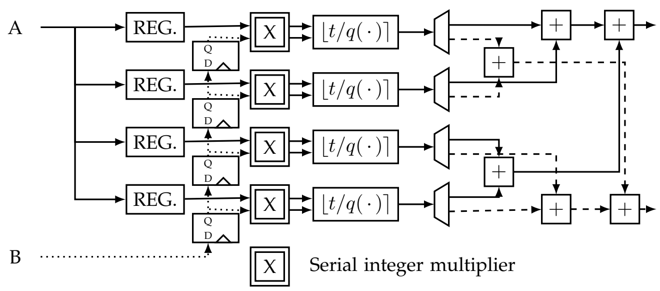
Post-computation follows the same approach than pre-computation and is constructed by a recursive architecture. Figure 10(a) shows elementary operations required for a post-computation stage, and Figure 10(b) the related architecture. As it can be noticed, much more operations are required compared to pre-computations, comprising six coefficient additions and 14 coefficient subtractions.

4.2.8 *Adapting FV.Relin in hardware*

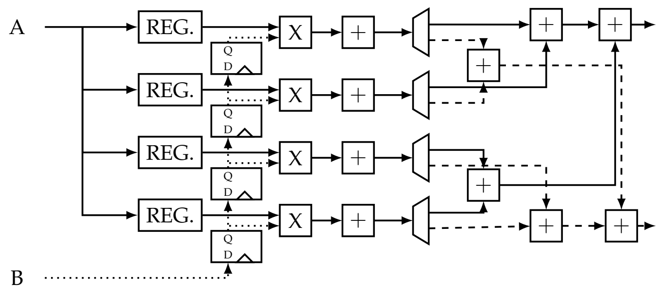
Several modifications to our design are required to use Karatsuba for relinearization. First, one needs a pre-computation and a pre-crossbar for each relinearization key. Second, the integer multiplier needs to be adapted. However, no modifications are needed for post-computations. By sending polynomials as before, the FV.WordDecomp$_{w,q}$ operation is already done. Because the first relinearization key must be multiplied by the first segment of the polynomial to be relinearized, the second key with the second

Inputs		Elementary operations							Outputs	
a_0	b_0	a_0b_0							r_0	
a_1	b_1	a_0b_1	$+$	a_1b_0					r_1	
a_2	b_2	a_0b_2	$+$	a_1b_1	$+$	a_2b_0			r_2	
a_3	b_3	a_0b_3	$+$	a_1b_2	$+$	a_2b_1	$+$	a_3b_0	r_3	
a'_0	b'_0	$a'_0b'_0$		a_1b_3	$+$	a_2b_2	$+$	a_3b_1	r_4	r'_0
a'_1	b'_1	$a'_0b'_1$	$+$	$a'_1b'_0$		a_2b_3	$+$	a_3b_2	r_5	r'_1
a'_2	b'_2	$a'_0b'_2$	$+$	$a'_1b'_1$	$+$	$a'_2b'_0$		a_3b_3	r_6	r'_2

(a) Elementary operations scheduling



(b) Polynomial multiplier architecture



(c) Integer multiplier architecture

Fig. 9: Overview of polynomial and integer multipliers, elementary operations schedule 9(a) and the associated architectures 9(b) & 9(c).

segment and so on, relinearization keys are not sent at the same time but shifted, as shown in Figure 11(a). Like for the standard polynomial multiplier, both architectures of the polynomial multiplier and the integer multiplier are visible in Figures 11(b) and 11(c). The polynomial relinearizer multiplier is quite similar to the standard one except of the left operand operations that are performed on each relinearization key. The integer multiplier has now as many inputs as the number of relinearization keys, and FIFOs are added after DSPs in order to realign coefficients as it can be seen in Figure 11(a) at the bottom. By adding simple switches before and after DSPs, the serial polynomial multiplier and the polynomial relinearizer multiplier can share the same logic, limiting as much as possible hardware resources consumption.

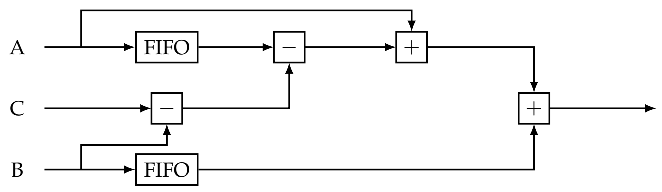
5 Discussion

5.1 Implementations Results

Table 4 provides implementation results and compares the proposed solution to the FFT implementation in [17], which uses NWC. Even if the FFT implementation performs a

Inputs			Elementary operations									Output
a_0			a_0									r_0
a_1			a_1									r_1
a_2			a_2									r_2
a_3			a_3									r_3
a_4	b_0	c_0	a_4	$+$	c_0	$-$	b_0	$-$	a_0			r_4
a_5	b_1	c_1	a_5	$+$	c_1	$-$	b_1	$-$	a_1			r_5
a_6	b_2	c_2	a_6	$+$	c_2	$-$	b_2	$-$	a_2			r_6
	b_3	c_3			c_3	$-$	b_3	$-$	a_3			r_7
	b_4	c_4			c_4	$-$	b_4	$-$	a_4	$+$	b_0	r_8
	b_5	c_5			c_5	$-$	b_5	$-$	a_5	$+$	b_1	r_9
	b_6	c_6			c_6	$-$	b_6	$-$	a_6	$+$	b_2	r_{10}
											b_3	r_{11}
											b_4	r_{12}
											b_5	r_{13}
											b_6	r_{14}

(a) Elementary operations schedule



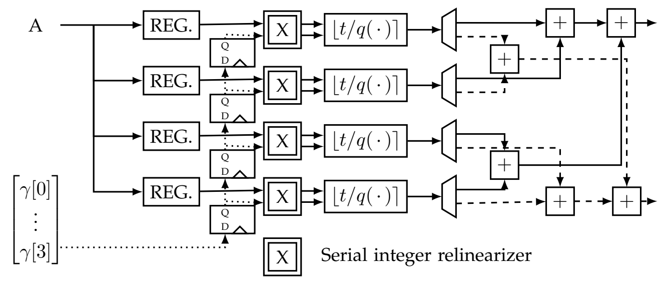
(b) Architecture

Fig. 10: Overview of post-computation elementary operations scheduling 10(a) and the associated architecture 10(b).

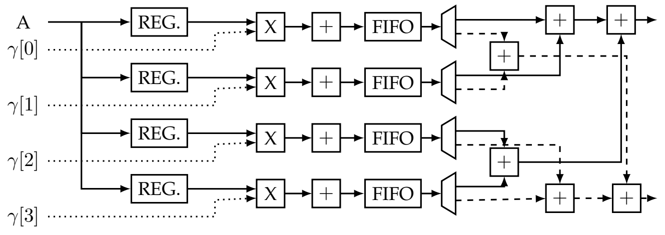
larger polynomial multiplication, these two designs are functionally equivalent. Indeed, to multiply two degree 2560 polynomials, one needs a 8192-FFT or a 4096-NWC. The only difference is the fact that our design supports the batching technique. However, even in the case of the NWC, our accelerator reduces computation time by 23% for the homomorphic multiplication in FV, and reduces ALMs by 57%, registers by 46%, embedded memory by 99.95% and DSPs by 30%. This is due to the fact that our accelerator is hardware/software co-designed and some computations are deported to the software, when FFT requires an autonomous implementation in hardware. The large memory saving is also the consequence that our accelerator runs as a flow and so does not require to store large banks of coefficients. Moreover, even if Karatsuba needs to send more coefficients than FFT to the hardware accelerator, the transfer is hidden during Karatsuba hardware computations.
By considering the polynomial multiplication only, FFT implementation has a lower latency than our accelerator, due to the lower complexity of FFT. However, for FV.Relin, Karatsuba becomes very efficient, because it can be cleverly adapted to this step as stated in Section 4.2.8, compared to the FFT which fails in flexibility. To the best of our knowledge, our solution is the first one which allows batching operations for multiplicative depth up to 4.
As one can see, our solution has f_{max} upper than the frequency provided by the PCIe, that is to say 250 MHz. The computation times provided are based on this restricted frequency only.
By carefully examining software results, the only critical operation is the post-computation. To reduce the dependence

Inputs	$a[0]_0$	$a[1]_0$	$a[2]_0$	$a[3]_0$	$a[0]_1$	$a[1]_1$	$a[2]_1$	$a[3]_1$
	$\gamma[0]_0$	$\gamma[0]_1$	$\gamma[0]_2$	$\gamma[0]_3$	$\gamma[0]_4$	$\gamma[0]_5$	$\gamma[0]_6$	$\gamma[0]_7$
		$\gamma[1]_0$	$\gamma[1]_1$	$\gamma[1]_2$	$\gamma[1]_3$	$\gamma[1]_4$	$\gamma[1]_5$	$\gamma[1]_6$
			$\gamma[2]_0$	$\gamma[2]_1$	$\gamma[2]_2$	$\gamma[2]_3$	$\gamma[2]_4$	$\gamma[2]_5$
				$\gamma[3]_0$	$\gamma[3]_1$	$\gamma[3]_2$	$\gamma[3]_3$	$\gamma[3]_4$
$\times$	$a[0]_0$	$a[0]_0$	$a[0]_0$	$a[0]_0$	$a[0]_1$	$a[0]_1$	$a[0]_1$	$a[0]_1$
	$\gamma[0]_0$	$\gamma[0]_1$	$\gamma[0]_2$	$\gamma[0]_3$	$\gamma[0]_4$	$\gamma[0]_5$	$\gamma[0]_6$	$\gamma[0]_7$
Output	$r[0]_0$	$r[0]_1$	$r[0]_2$	$r[0]_3$	$r[0]_4$	$r[0]_5$	$r[0]_6$	$r[0]_7$
$\times$		$a[1]_0$	$a[1]_0$	$a[1]_0$	$a[1]_1$	$a[1]_1$	$a[1]_1$	$a[1]_1$
		$\gamma[1]_0$	$\gamma[1]_1$	$\gamma[1]_2$	$\gamma[1]_3$	$\gamma[1]_4$	$\gamma[1]_5$	$\gamma[1]_6$
Output		$r[1]_0$	$r[1]_1$	$r[1]_2$	$r[1]_3$	$r[1]_4$	$r[1]_5$	$r[1]_6$
$\times$			$a[2]_0$	$a[2]_0$	$a[2]_1$	$a[2]_1$	$a[2]_1$	$a[2]_1$
			$\gamma[2]_0$	$\gamma[2]_1$	$\gamma[2]_2$	$\gamma[2]_3$	$\gamma[2]_4$	$\gamma[2]_5$
Output			$r[2]_0$	$r[2]_1$	$r[2]_2$	$r[2]_3$	$r[2]_4$	$r[2]_5$
$\times$				$a[3]_0$	$a[3]_1$	$a[3]_1$	$a[3]_1$	$a[3]_1$
				$\gamma[3]_0$	$\gamma[3]_1$	$\gamma[3]_2$	$\gamma[3]_3$	$\gamma[3]_4$
Output				$r[3]_0$	$r[3]_1$	$r[3]_2$	$r[3]_3$	$r[3]_4$
$+$				$r[0]_0$	$r[0]_1$	$r[0]_2$	$r[0]_3$	$r[0]_4$
				$r[1]_0$	$r[1]_1$	$r[1]_2$	$r[1]_3$	$r[1]_4$
				$r[2]_0$	$r[2]_1$	$r[2]_2$	$r[2]_3$	$r[2]_4$
				$r[3]_0$	$r[3]_1$	$r[3]_2$	$r[3]_3$	$r[3]_4$
Output				r_0	r_1	r_2	r_3	r_4

(a) Elementary operations schedule



(b) Polynomial relinearizer architecture



(c) Integer relinearizer architecture

Fig. 11: Overview of polynomial and integer multipliers, elementary operations schedule 11(a) and the associated architectures 11(b) & 11(c).

from the CPU load, extra post-computations in hardware may be a good solution.

5.2 Comparison to Software Implementation of FV

Recent work on pure software implementation of FV in [22] provides very promising computation times for low level multiplicative depths. For parameter set ($n = 4096$, $\log_2 q = 168$), authors of [22] can achieve an homomorphic multiplication in 7.68 ms. They implement a full RNS variant of FV to reduce the size of coefficients, then compute fast polynomial arithmetic using NWC FFT. The modulus is larger than ours because, for efficiency, authors of [22] set ω to 62 bits, inferring a larger modulus but a much smaller relinearization key.

TABLE 4: Implementations results compared to FFT implementation from [17].

		Our design	FFT [17]
Setup (n, $\log_2 q$)		(2560, 125)	(4096, 125)
Resources	ALM	29 534	69 058
	Registers	76 823	144 747
	Memory bits	4 131	8 031 568
	DSPs	100	144
	f_{max}	331.13 MHz	100 MHz
Polynomial $\times$	Software pre-computation	488 us	
	Hardware accelerator	583.2 us	
	Software post-computation	1.37 ms	
	Total	2.44 ms	1.96 ms
Relin	Software pre-computation	336 us	
	Hardware accelerator	583.2 us	
	Software post-computation	1.37 ms	
	Total	2.29 ms	4.79 ms
YASHE' $\times$		4.73 ms	6.75 ms
FV $\times$		11.9 ms	15.46 ms

Compared to the FFT hardware implementation in [17] with parameters set ($n = 4096$, $\log_2 q = 125$), the pure software solution is two times faster. Compared to our approach, because our design can use the batching technique and not the optimized software implementation, our accelerator remains interesting. To allow batching, the NWC requires to double the size of the FFT. Thus to fairly compare the two approaches, software timing results for parameters ($n = 8192$, $\log_2 q = 168$) would be required. Due to the lack of software implementation results, we just provide an estimation of computation time based on the NTT complexity provided in [22]. Because NTT has a complexity of $\mathcal{O}(n \log_2 n)$, increasing the NTT from size 4096 to 8192 increases the complexity by a factor of 2.16. Thus, we can estimate the computation time of the full RNS software implementation with batching to 7.68 ms $\times$ 2.16 = 16.58 ms, and so our accelerator remains competitive. Moreover, the size of ciphertexts is smaller in our case due to a smaller ω, but also because polynomials degree is smaller. Furthermore, several optimizations can be made on our accelerator, in particular on the software part, in order to further improve its competitiveness.

5.3 Scalability of the Proposed Accelerator

Our implementation results demonstrate that for the proposed homomorphic scenario, that is to say circuits with multiplicative depths up to 4, our accelerator reduces both computation times and hardware resources on the FPGA compared to the FFT. However, a main concern is to evaluate the scalability of the architecture for higher multiplicative depths. Due to the asymptotic complexity of the FFT, it is clear that Karatsuba will fail in competitiveness after a certain degree.

TABLE 5: Turning point of Karatsuba compared to FFT.

		Our design	FFT [17]
Setup (n, $\log_2 q$)		(6144, 512)	(16384, 512)
Resources	ALM	153 642	141 090
	Registers	329 235	391 773
	Memory bits	301 645	17 626 400
	DSPs	456	577
	f_{max}	262.47 MHz	66 MHz
Polynomial ×	Software pre-computation	2.74 ms	
	Hardware accelerator	7.98 ms	
	Software post-computation	8.68 ms	
	Total	19.39 ms	27.88 ms
Relin	Software pre-computation	1.95 ms	
	Hardware accelerator	23.94 ms	
	Software post-computation	8.68 ms	
	Total	34.58 ms	20.80 ms
YASHE′ ×		52.7 ms	47.54 ms
FV ×		124.36 ms	122.30 ms

TABLE 6: Example of polynomial multiplication degree achievable with Karatsuba.

lowest sub-polynomial degree	Karatsuba recursions	polynomial multiplication degree
5	9	2560
6	9	3072
7	9	3584
8	9	4096
9	9	4608
5	10	5120
6	10	6144
7	10	7168
8	10	8192
9	10	9216
5	11	10240
6	11	12288
7	11	14336
8	11	16384
9	11	18432

To estimate that degree, we have implemented various configurations of our accelerator until matching to an existing FFT implementation both in terms of hardware resources consumption and computation time. We found a turning point of our Karatsuba approach for degree 6144 polynomials with 512 bits coefficients. With such parameters, an FFT using the batching technique must be of size 16384 with 512 bits coefficients. Table 5 provides implementation results of our accelerator for parameters set ($n = 6144$, $\log_2 q = 512$) compared to FFT implementation in [17] with parameters set ($n = 16384$, $\log_2 q = 512$). As one can see, the hardware resources consumption is equivalent with comparable computing time. The main limit to Karatsuba scalability is clearly the relinearization. Compared to the hardware computation time of the polynomial multiplication, the relinearization takes 3 times longer. Indeed, due to the limited bandwidth of the PCIe, we are not able to send the complete relinearization key. Because the PCIe is equivalent to a 250 MHz bus with 256 bits width on the FPGA side, and because our polynomials coefficients are split into 27 bits segments, we can only send 9 polynomials in parallel ($9 \times 27 = 243 < 256$). Thus, when the number of relinearization sub-keys exceeds 8, we need to start again the hardware relinearization process with the remaining sub-keys. For parameters set ($n = 6144$, $\log_2 q = 512$), the number of relinearization sub-keys is 19, requiring 3 hardware relinearizations. The software computation time of post recursions is also an important issue, but can be compensated by additional efforts on the software part.

5.4 Pros and Cons of Karatsuba Compared to FFT

As settled in Section 4, Karatsuba can be more efficient than the FFT for both computation time and resources utilization for specific parameters. Karatsuba has several advantages compared to FFT, despite its highest asymptotic complexity. First, Karatsuba is a simple algorithm, with basic pre- and post-computations and so can be easily implemented. Second, Karatsuba can perform polynomial multiplications with non-power of two degrees, allowing to fit more precisely to the required parameters. Table 6 provides examples of polynomial multiplications achievable with Karatsuba. Third, the modulus can be freely selected compared to FFT, reducing the complexity of the division and rounding operation required by the FV scheme to a simple shift. Moreover, the division and rounding in the FFT case is reported to be an important bottleneck in [17]. Fourth, thanks to the use of Karatsuba, several computations can be hidden during transfers. For our accelerator, the sub-polynomials multiplication performed on the FPGA is hidden by the transfers through the PCIe. Fifth, the relinearization can be efficiently adapted to Karatsuba as explained in Section 3.2. Karatsuba has also some limitations. First, Karatsuba requires a software/hardware co-design approach to meet competitive computation times, which is not the case for FFT. Second, as stated in Section 5.3, Karatsuba is a good alternative to FFT only until a certain degree. We estimate this degree to 6144 subject to change if improvements are made on Karatsuba or FFT implementations. Third, because the polynomial multiplication degree achievable by FFT is often over-sized for a given multiplicative depth, changing the multiplicative depth only requires to change the modulus, assuming that the degree does not exceed the size of the FFT. For Karatsuba, each multiplicative depth requires a specific configuration, inferring a substantial modification of the hardware accelerator to change the lowest sub-polynomial multiplication degree.

6 Conclusion

In this paper, we demonstrate that for some cases, especially when the polynomial degree is just upper than a power of 2 and less than 6144, Karatsuba algorithm can be a good alternative to FFT. The study provides a complete implementation of a software/hardware co-design approach of Karatsuba for degree 2560 polynomials with 135 bits coefficients, allowing homomorphic operations on the FV scheme

for algorithms with a multiplicative depth up to 4. We also provide information on the scalability of our approach and an estimation of the degree when Karatsuba becomes less efficient than FFT. Compared to previous state of the art contributions, and especially implementation in [17], our accelerator can perform an homomorphic multiplication of FV in 11.9 ms, when a functionally equivalent design using FFT requires about 15.46 ms for a multiplicative depth up to 4, and halves the hardware resources consumption. Moreover, our approach goes in the right direction considering that recently published Homomorphic Encryption schemes have a lower polynomial degree than previous ones [1].
Future work will consist on evaluating the proposed solution to a more constraint architecture. We will also investigate how to improve the design scalability.

Acknowledgment

Authors would like to thank the reviewers for their precious time and constructive comments which helped us to further improve the quality of the manuscript.
This study has been partially funded by the french Direction Générale de l'Armement (DGA).

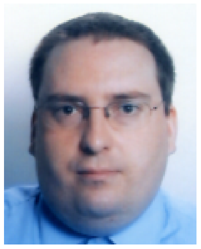
References

[1] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds," in *Proc. of ASIACRYPT 2016*, ser. LNCS.
[2] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Proc. of Advances in Cryptology — EUROCRYPT 1999*, ser. LNCS, no. 1592, pp. 223–238.
[3] R. L. Rivest, A. Shamir, and L. M. Adleman, "A Method for Obtaining Digital Signatures and Public-key Cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.
[4] C. Gentry, "A Fully Homomorphic Encryption Scheme," Ph.D. dissertation, Stanford University, 2009.
[5] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully Homomorphic Encryption over the Integers," in *Proc. of Advances in Cryptology – EUROCRYPT 2010*. Springer, pp. 24–43.
[6] J.-S. Coron, T. Lepoint, and M. Tibouchi, "Scale-Invariant Fully Homomorphic Encryption over the Integers," in *Proc. of Public-Key Cryptography – PKC 2014*. Springer, pp. 311–328.
[7] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved Security for a Ring-Based Fully Homomorphic Encryption Scheme," in *Proc. of Cryptography and Coding: 14th IMA International Conference – IMACC 2013*. Springer, pp. 45–64.
[8] Y. Doröz and B. Sunar, "Flattening NTRU for Evaluation Key Free Homomorphic Encryption," Cryptology ePrint Archive, Report 2016/315, 2016.
[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) Fully Homomorphic Encryption without Bootstrapping," in *Proc. of the 3rd Innovations in Theoretical Computer Science Conference – ITCS 2012*, pp. 309–325.
[10] Z. Brakerski, "Fully homomorphic Encryption without Modulus Switching from Classical GapSVP," in *Proc. of Advances in Cryptology – CRYPTO 2012*.
[11] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," Cryptology ePrint Archive, Report 2012/144, 2012.
[12] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Proc. of Advances in Cryptology – CRYPTO 2013*, pp. 75–92.
[13] Z. Brakerski and V. Vaikuntanathan, "Lattice-Based FHE as Secure as PKE," in *Proc. of the 5th Conference on Innovations in Theoretical Computer Science – ITCS 2014*. ACM, pp. 1–12.
[14] A. Khedr, G. Gulak, and V. Vaikuntanathan, "SHIELD: Scalable Homomorphic Implementation of Encrypted Data-Classifiers," *Accepted to IEEE Transactions on Computers*, 2015.
[15] M. Albrecht, S. Bai, and L. Ducas, "A Subfield Lattice Attack on Overstretched NTRU Assumptions: Cryptanalysis of Some FHE and Graded Encoding Schemes," Proc. of Advances in Cryptology – CRYPTO 2016.
[16] D. Stehlé and R. Steinfeld, "Making NTRUEncrypt and NTRUSign as Secure as Standard Worst-Case Problems over Ideal Lattices," Cryptology ePrint Archive, Report 2013/004, 2013.
[17] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating Homomorphic Evaluation on Reconfigurable Hardware," in *Proc. of Cryptographic Hardware and Embedded Systems – CHES 2015*. Springer, pp. 143–163.
[18] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular Hardware Architecture for Somewhat Homomorphic Function Evaluation," in *Proc. of Cryptographic Hardware and Embedded Systems – CHES 2015*. Springer, pp. 164–184.
[19] T. Lepoint and M. Naehrig, "A Comparison of the Homomorphic Encryption Schemes FV and YASHE," in *Proc. of AFRICACRYPT 2014*, ser. LNCS, vol. 8469, pp. 318–335.
[20] W. Hart, "FLINT library," http://www.flintlib.org/, 2016, [Online; accessed 31-May-2016].
[21] C. Aguilar-Melchor, J. Barrier, S. Guelton, A. Guinet, M.-O. Killijian, and T. Lepoint, "NFLlib: NTT-Based Fast Lattice Library," in *SA Conference Cryptographers' Track*, 2016.
[22] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, "A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes," in *Selected Areas in Cryptography - SAC*, Aug. 2016.
[23] J. Pollard, "The Fast Fourier Transform in a Finite Field," in *Mathematics of Computation*, 1971, vol. 25, no. 90, pp. 365–374.
[24] J. A. Solinas, "Generalized Mersenne Prime," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 509–510.
[25] A. Karatsuba and Y. Ofman, "Multiplication of Multi-Digit Numbers on Automata (in Russian)," *Doklady Akad. Nauk SSSR*, vol. 145, no. 2, p. 293–294, 1962, translation in Soviet Physics-Doklady, 44(7), 1963, pp. 595-596.
[26] D. Catalano and D. Fiore, "Using Linearly-Homomorphic Encryption to Evaluate Degree-2 Functions on Encrypted Data," in *Proc. of Computer and Communications Security – CCS 2015*.
[27] N. P. Smart and F. Vercauteren, "Fully Homomorphic SIMD Operations," *Designs, Codes and Cryptography*, vol. 71, no. 1, pp. 57–81, 2014.
[28] V. Shoup, "NTL library," http://www.shoup.net/ntl/, 2016, [Online; accessed 31-May-2016].
[29] T. Granlund, "GMP library," https://gmplib.org/, 2015, [Online; accessed 31-May-2016].

Vincent Migliore is a PhD student in Electrical and Computer Engineering with the Université Bretagne Sud, France. He received his M.Sc. degree from Ecole Normale Superieure de Rennes, France in 2014. His current research interests deal with the implementation of fast software/hardware computations for Fully/Somewhat Homomorphic Encryption.

Maria Méndez Real is a PhD student in Electrical and Computer Engineering with the Université Bretagne Sud (UBS). She received her M.Sc. degree in Electrical and Computer Engineering from UBS, Lorient, France in 2014. Her research interests deal with computer architecture, embedded systems, systems security and virtual prototyping. Her work focuses on the investigation of side-channel attacks and countermeasures for multi/many-core architectures.

Vianney Lapotre received his M.Sc. and his Ph.D. in Electrical and Computer Engineering from the University Bretagne Sud, France, in 2010 and 2013 respectively. In 2012 he spent six months as an invited researcher at the Ruhr-University of Bochum, Germany. From 2013 to 2014, he was a Postdoctoral at LIRMM, Montpellier, France. He was involved in the European Mont-Blanc project. He is currently Associate Professor at University Bretagne Sud, France. His research interests include reconfigurable and self-adaptive multiprocessor architectures, NoC-based architectures, high performance embedded systems and system security.



Arnaud Tisserand (M'00-SM'06) is Senior Researcher at CNRS in IRISA laboratory, Lannion, France. He received the following degrees in Computer Science: HDR (French Tenure, 2010, from University of Rennes), Ph.D. and M.Sc. (1997 and 1994, from Ecole Normale Superieure de Lyon) in France. His main research interests are in computer arithmetic, asymmetric cryptography, hardware security, low-power and/or reconfigurable circuit design.

Caroline Fontaine is a full-time researcher at CNRS (French National Research Institute). She has been working on content protection for more than 15 years. Her publications cover cryptography, steganography, digital watermarking, and active fingerprinting, with most of her articles tackling several of these domains at the same time. She has been and is involved in many research projects on these topics, and in the organization and program committees of many conferences and publications. She is with CNRS/Lab-STICC and Télécom Bretagne.

Guy Gogniat is a Professor in ECE with the Université Bretagne Sud, Lorient, France, where he has been since 1998. In 2005, he spent one year as an invited Researcher with the University of Massachusetts, Amherst, USA, where he worked on embedded system security using reconfigurable technologies. His work focuses on embedded systems design methodologies and tools. He also conducts research in the domain of reconfigurable and adaptive computing and embedded system security.