Securing the Cloud with Reconfigurable Computing: An FPGA Accelerator for Homomorphic Encryption

Alessandro Cilardo and Domenico Argenziano
DIETI - University of Naples Federico II
Naples, Italy, Email: acilardo@unina.it

***Abstract*—A hot topic in current cloud security research, homomorphic encryption is a recently introduced technique allowing computation to take place on encrypted data. This work presents the architecture and implementation of a dedicated FPGA-based accelerator addressing the prohibitive computing demand of homomorphic encryption. In particular, the accelerator targets the most time consuming operation used by the encryption primitive, large integer multiplication. Based on an Altera's Stratix V FPGA platform, the prototype implementation achieves significant improvements in terms of execution time – under a comparable hardware cost– against alternative solutions previously presented in the technical literature.**

I. Introduction

Homomorphic encryption (HE) [1] is a promising answer to the security concerns raised by cloud computing since it allows data to be stored and manipulated remotely in an encrypted form, effectively preventing the server from accessing the information being processed. Ideally, HE can serve as an enabling tool for a number of different applications, including multiparty computation, medical applications, financial computing, electronic voting, etc. Unfortunately, in spite of the large variety of alternative HE schemes available today [1], [2], [3], HE suffers from extremely high computational costs, which currently prevents its practical use. In that respect, the availability of dedicated FPGA-based acceleration in server settings might play a key role. In fact, recent trends clearly suggest that reconfigurable hardware, i.e. FPGA devices, may potentially play a key role in datacenters and cloud servers for certain classes of applications [4], [5]. In addition to user application acceleration, FPGAs have a large potential for security-related processing as well. FPGAs have proved to be an effective platform for cryptographic processing [6] as cryptoalgorithms have peculiar characteristics, like integer computation, bit-level manipulation, etc., that make standard platforms like CPUs and GPUs less competitive. On the other hand, hardware reconfigurability allows the designer to customize the system possibly based on specific parameters, e.g. a cryptographic key, making FPGAs an ideal platform for cryptographic acceleration [7], [8], [9] as well as for cryptanalytic purposes [10], [11], [12]. FPGA platforms have also been explored as a secure compute/storage environment [13], [14], [15] as well as for implementing special security-related features like Physically Unclonable Functions [16].

This paper presents a dedicated FPGA-based accelerator implementing ultralong integer multiplication, the main performance bottleneck in most homomorphic encryption schemes. The work describes an implementation based on an Altera's Stratix V FPGA platform. The experimental results collected from the hardware synthesis show significant improvements in terms of execution time –under a comparable hardware cost– against alternative solutions previously presented in the technical literature.

The paper is structured as follows. Section II recapitulates the current state of the art in the implementation of the HE primitives. Section III introduces the algorithm implemented by the proposed solution. Section IV describes the architecture and optimizations adopted for the implemented FPGA-based system. Section V presents the main experimental results collected from hardware synthesis. Section VI concludes the paper with some final remarks.

II. Previous works

Cloud computing has emerged as an important paradigm shift for a large class of applications [17], [18], [19]. Security is a major concern in cloud settings, pointing out the importance of advanced cryptographic techniques like homomorphic encryption, allowing computation to take place on encrypted data on the server side. In particular, this work addresses the so-called *Fully Homomorphic Encryption* (FHE), introduced by Gentry's seminal work [1] just a few years ago. Beside Gentry's scheme, based on the properties of ideal lattices, various alternative solutions have been proposed, the most relevant being the van Dijk, Gentry, Halevi, and Vaikuntanathan's (DGHV) scheme over the integers, and the Brakerski and Vaikuntanathan's scheme [2] based on the Learning with Errors (LWE) and Ring Learning with Errors (RLWE) problems [3].

An implementation of a variant of the original scheme [1] is proposed by Gentry and Halevi [20]. Their solution, despite various optimizations and small-size security parameters, takes more than one second for encrypting a single bit on an Intel Xeon server. Recent software implementations include [21], [22]. An open-source library, hcrypt, is available on-line [23], while [24] contains an optimized implementation reaching a significant speed-up over the previous solutions. Several research works concerning FHE computing platforms have looked for alternative architectures, particularly GPUs and FPGAs. GPUs offer high throughput and efficiency for data intensive computing, such as vector and linear algebra problems. FHE schemes can benefit from this architecture, since they are highly parallelizable with respect to data. FPGA technology offers, on the other hand, the flexibility of implementing a custom and targeted architecture at a low cost, as opposed to Application Specific Integrated Circuits (ASICs). Moreover, several FPGAs include built-in optimized blocks for multiply-and-accumulate operations, which can be effectively exploited

when implementing large multiplication. Recent GPU implementations include [25], [26], [27]. An FPGA implementation is presented in [28], which compares FPGAs and GPUs, namely Altera Stratix V and NVIDIA Tesla C2050 devices. The solution is fundamentally focused on the FFT multiplier building block. The authors conclude that the FPGA version is at least twice as fast as the GPU one, with lower power consumption. [29] and [30] propose a full custom ASIC implementation of large-operand multiplication. For example, in [29] a single multiplication is performed in 7.7 ms at 666 MHz. The authors of [31] build a custom hardware implementation of the cryptographic primitives of Gentry-Halevi's FHE scheme. The design includes optimizations previously introduced in [25] to reduce the number of FFT computations. Last, [32] proposes an FFT-based large integer multiplier, along with a Barrett reduction module. The design is implemented on a Xilinx Virtex-7 FPGA and includes the encryption primitive of Coron *et al.* FHE scheme [33], [34]. The results show a remarkable speed-up compared to existing software implementations.

III. Algorithm

The accelerator presented in this work targets the most time consuming operation used by the encryption primitive, integer multiplication on very large operands, in the order of millions of bits. Other mathematical operations involved in the primitive can either be reduced to a combination of multiplications, or are not the main bottleneck. Ultralong multiplication, in fact, plays a central role in different fully homomorphic schemes, such as the integer-based approach and solutions based on Lattice problems and Learning with Errors, which may thus be implemented on top of the accelerator, e.g. as software routines. We addressed the efficient implementation of asymptotically faster (but inherently more complex) multiplication algorithms in place of usual schemes used for moderately large operands (thousands of bits). An asymptotically efficient multiplication algorithm is the Schönhage-Strassen algorithm (SSA), which exploits the properties of the Number-Theoretic Discrete Fourier Transform and is advantageous for operands of at least $100,000$ bits. In essence, the algorithm computes $c = a \cdot b$ as follows:

- decompose operands a and b into groups of m bits and consider such groups as polynomial coefficients;
- perform the integer FFT of a and b, hence getting A and B;
- compute $C = A \cdot B$ component-wise, which can be easily parallelized;
- compute the Inverse FFT of C, namely c';
- compute the final result c performing the shifted sum of the components of c'.

The computational complexity of SSA is $O(n \cdot \log n \cdot \log\log n)$. The most time consuming operation in SSA is the computation of the FFT (and Inverse FFT). Instead of the more common binary recursive splitting approach relying on a radix-2 transform, we adopted the original Cooley-Tukey general FFT decomposition, with higher radices.

Decomposing N as $N = N_1 \cdot N_2$, the input and output vectors can be split into N_1 sub-sequences of length N_2. Let $n = N_2 n_1 + n_2$ and $k = N_1 k_2 + k_1$ with $n_1, k_1 \in \{0, 1, \ldots, N_1 - 1\}$ and $n_2, k_2 \in \{0, 1, \ldots, N_2 - 1\}$. Then the DFT can be written as:

$$F\left[N_1 k_2 + k_1\right] = \sum_{n=0}^{N-1} f[n]\omega_N^{kn} = \\ = \sum_{n_2=0}^{N_2-1}\left[\left(\sum_{n_1=0}^{N_1-1} f\left[N_2 n_1 + n_2\right]\cdot\omega_{N_1}^{n_1 k_1}\right)\cdot\omega_N^{n_2 k_1}\right]\cdot\omega_{N_2}^{n_2 k_2} \tag{1}$$

We choose to perform the computation in the finite field $\mathbb{Z}/p\mathbb{Z}$, with prime p. By selecting a proper prime p, the modular multiplication in the finite field can be computed rapidly as a sequence of shifts. In our implementation, we choose the Solinas prime number $p = 2^{64} - 2^{32} + 1$. We assume to deal with operands of $786,432$ bits, which correspond to the small security parameter setting for DGHV adopted in various research papers. Operands are decomposed into 32K coefficients of 24 bits.

We need to apply FFT on 64K points, in order to accommodate the multiplication result. By using Equation 1 recursively on the 64K-point DFT, it can be computed with three stages using radix-64 and radix-16 sub-transforms:

$$\sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1}\left[\sum_{n_3=0}^{N_3-1}\left(a[n]\omega_{N_3}^{n_3 k_3}\right)\omega_{N_2 N_3}^{n_2 k_3}\omega_{N_2}^{n_2 k_2}\right]\omega_N^{n_1 k_2'}\omega_{N_1}^{n_1 k_1} = \\ = \sum_{n_1=0}^{15}\sum_{n_2=0}^{63}\left[\sum_{n_3=0}^{63}\left(a[n]\omega_{N_3}^{n_3 k_3}\right)\omega_{N_2 N_3}^{n_2 k_3}\omega_{N_2}^{n_2 k_2}\right]\omega_N^{n_1 k_2'}\omega_{N_1}^{n_1 k_1} \tag{2}$$

where $n = N_1 N_2 n_3 + N_1 n_2 + n_1$, $k_2' = k_3 + N_3 k_2$.

Each stage can be efficiently parallelized, according to the available computing resources.

IV. Architecture of the FPGA-based accelerator

An essential design objective we set for the proposed accelerator was the inherent support for scalability to ultralong operands which, unlike many cryptographic primitives in different contexts, may require a flexible and composable design solution applicable either to on- or off-chip scenarios, possibly in multi-FPGA settings available on the server side. Consequently, for the implementation of the 64K-point FFT building block, we devised a flexible distributed approach, relying on several nodes connected in a hypercube topology, which matches exactly the logical topology of the distributed FFT

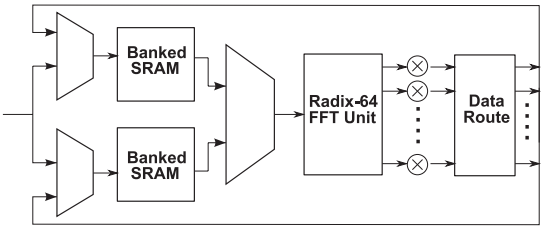

Fig. 1. Architecture of a 64K FFT processing element.

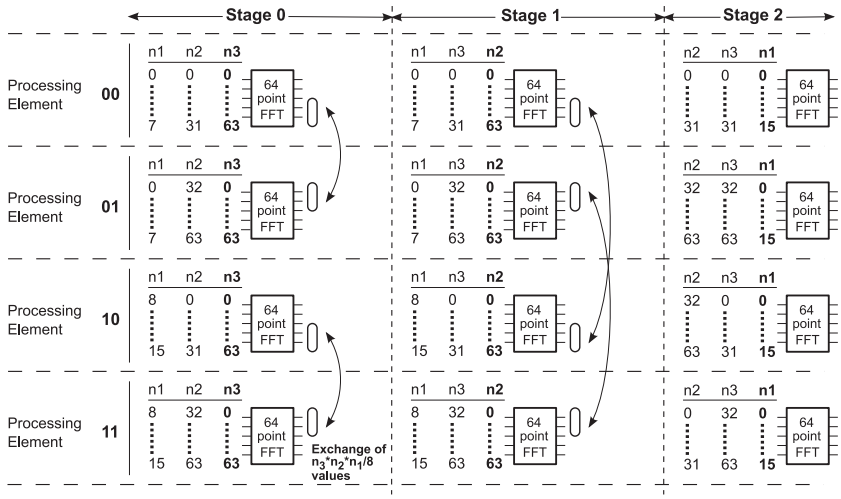

Fig. 2. Data distribution.

algorithm. The solution was initially prototyped on a multi-board platform based on low-end devices (Altera Cyclone V) then extended to a hybrid on-/off-chip solution relying on a larger device, i.e. an Altera Stratix V FPGA. The distributed approach, distinguishing our proposal from previous related works like [28], matches very well the FFT computation and ensures several advantages compared to shared memory approaches, such as better scalability and reduced use of global routing resources, which may be a major performance bottleneck especially on FPGAs. Using a hypercube topology, the number of communication stages for FFT computation is the hypercube dimension d. In each stage, a node communicates only with one of its d neighbors, one for each stage. The number of computation stages l instead depends on the FFT decomposition, as previously shown. We must have $l > d$ in order to correctly interleave computation and communication. If $l > d + 1$, communication takes places only after the first d computation stages while the subsequent stages are computation only. The overall architecture of a single node, called here *Processing Element*, is shown in Figure 1.

The core computing element is the Radix-64/16 FFT unit, which computes the basic sub-transforms. Since in our distributed scheme communication will indeed overlap with computing, double buffering is used: while a buffer is feeding current input values, the other one is filled with new values coming partly from the same node and partly from one of its neighbors. At the end of a computation stage, the roles of the buffers are swapped. Buffers are based on a banked architecture which uses the SRAM primitive blocks of the underlying FPGA architecture. Additionally, we also need a group of modular multipliers for twiddle factor multiplications, required between two consecutive FFT computation stages. The data route component is responsible for the proper ordering of FFT output points before writing to the memory buffers.

a) Data distribution and exchange pattern: Below we consider the computation of a 64K-point FFT with four processing elements. In the initial data distribution phase, the 64K-element vector is partitioned among the four processing elements, also considering the proper decomposition reordering. Then, computing and data exchange stages take place in an interleaved, but partially overlapped way. During a computing stage, each node can execute autonomously.

We recall also that, according to the previous formula, we can decompose the 64K FFT as per Equation 2. Figure 2 summarizes the sequence of computing and communication stages: bold style is used to highlight the index (one of n_1, n_2, and n_3) involved in the current sub-FFT computing and subsequent data exchange.

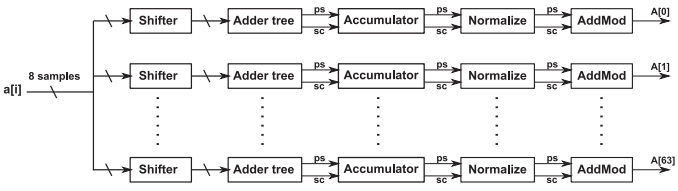

Fig. 3. Architecture of the baseline Radix-64 unit [28].

b) FFT-64 unit: The Radix-64 unit (or FFT-64) is the basic building block which is capable of computing the sub-transforms making up the overall FFT. It can be easily extended for computing Radix-16 FFT as well, though this will not be shown here. In the chosen finite field, the 64^{th} root of unity is 8, so multiplications in the FFT formula become simple shifts, as follows:

$$A[k] = \sum_{i=0}^{63} a[i]\omega_{64}^{i \cdot k} = \sum_{i=0}^{63} a[i]8^{i \cdot k} \pmod{p} \qquad (3)$$

Since $8^{64} \pmod{p} = 2^{192} \pmod{p} = 1$, no intermediate value can exceed 192 bits.

The unit proposed here builds on a baseline scheme [28] shown in Figure 3. Input samples are read 8-by-8 and are fed to 64 separated computing chains, one for each *frequency* component. Each chain comprises a shifter bank, where the eight samples are multiplied by their respective twiddle factor. Shifted values are summed by an adder tree to produce a partial sum. To avoid the latency of long carry chains, a carry save solution is adopted. The output is then made up of two vectors, which are not merged until the very last block (*AddMod*). The accumulator will sum up the partial sums in

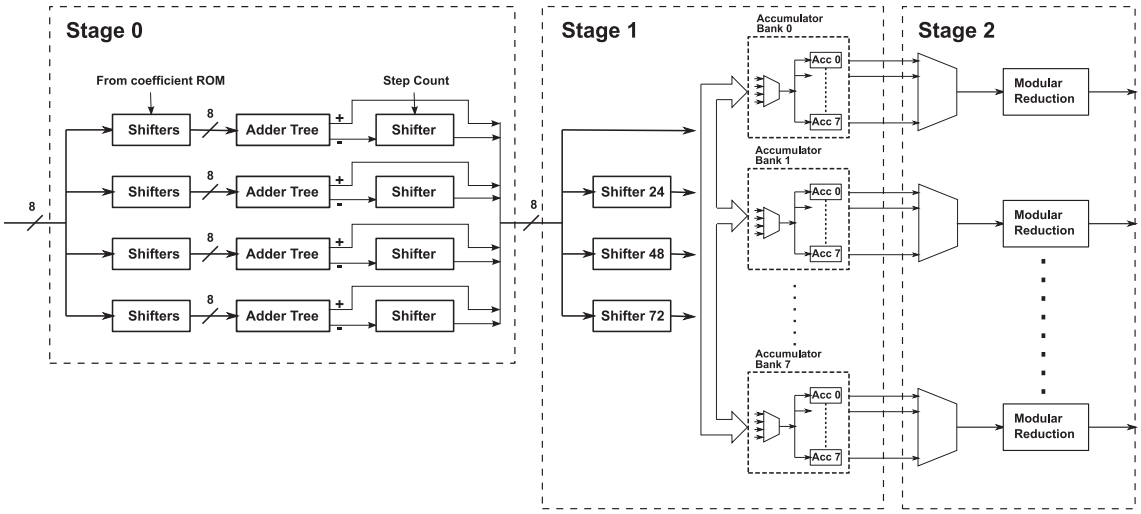

Fig. 4. Architecture of the FFT64 unit.

eight consecutive clock cycles. After the eighth clock cycle, the transform is complete and the value in the accumulator is modular reduced. The *Normalize* block computes a first coarse reduction by using Equation 4, which applies to 128-bit numbers and exploits the properties of the chosen modulus:

$$a \cdot 2^{96} + b \cdot 2^{64} + c \cdot 2^{32} + d = 2^{32}(b + c) - a - b + d \quad (4)$$

The result will require at most one extra addition or subtraction with the modulus p. This last operation is performed in the *AddMod* component.

The baseline scheme is somehow redundant since much work can be shared among the different components. Our architecture avoids redundancy through several structural solutions which, as shown in the implementation section, result in improved parallelism and area efficiency. In order to exploit common work between the FFT components, we apply Equation 1 to the 64-point FFT:

$$\sum_{i=0}^{63} a_i \omega^{i \cdot k} = \sum_{j=0}^{7} \left[\left(\sum_{i=0}^{7} a_{i \cdot 8 + j} \omega_8^{i \cdot k_1} \right) \cdot \omega_{64}^{j \cdot k_1} \right] \cdot \omega_8^{j \cdot k_2} = \\ = \sum_{j=0}^{7} \left(\sum_{i=0}^{7} a_{i \cdot 8 + j} \omega_8^{i \cdot k_1} \cdot \omega_{64}^{j \cdot k_1} \right) \cdot \omega_8^{j \cdot k_2} \quad (5)$$

The expression between parentheses is computed by the first stage in Figure 4, where eight samples from the memory are shifted and summed. This is done only for eight frequency components (denoted by k_1). Then, such partial sums are multiplied by the twiddle factor $\omega_8^{j \cdot k_2}$ while the external sum is performed by each accumulator. The multiplication by $\omega_8^{j \cdot k_2}$ leads to eight possible shifts, but they can be reduced to four if we consider that one half of the twiddle factors are the opposite of the other half (the partial sum will be subtracted in the accumulator instead of being summed). The four factors needed are then 2^0, 2^{24}, 2^{48}, and 2^{72} (respectively, no shift, shift by 24, 48, and 72 bits). Accumulators can be thought of as being partitioned into eight blocks of eight accumulators (block number corresponds to k_2 in Equation 5). Each block contains a multiplexer selecting which of the four shifts is needed, according to the block number and the current computing step (respectively, index k_2 and j). Each block receives also a *subtract* signal (not shown in figure).

The first stage itself is optimized by computing only four of the eight components by relying on the following property:

$$\sum_{i=0}^{7} a_{i \cdot 8 + j} \omega_8^{i \cdot k_1} \cdot \omega_{64}^{j \cdot k_1} = \sum_{i=0}^{7} a_{i \cdot 8 + j} \omega_8^{i \cdot (k_1' + 4)} \cdot \omega_{64}^{j \cdot (k_1' + 4)} = \\ = \sum_{i=0}^{7} a_{i \cdot 8 + j} \omega_8^{i \cdot k_1'} \cdot \omega_2^{i} \cdot \omega_{64}^{j \cdot k_1'} \cdot \omega_{16}^{j} \text{ for } k = 4, 5, 6, 7$$

We can see that components 4 to 7 can be computed similarly to the first four, except for the multiplication by a factor ω_{16}^{j} and the fact the in the summation odd terms are taken with negative sign. This is done by modifying the adder tree so that it outputs also the difference between the sums of even and odd terms (such modification adds little complexity to the adder tree).

After eight computing steps, the accumulators contain the FFT output which needs to be modular reduced. While the baseline scheme uses 64 modular reduction components, one for each accumulator, we observe that the maximum average throughput, even in a fully pipelined solution, is eight components per clock cycle. Consequently, we use only eight modular reductors, one for each accumulator block, preceded by a multiplexer which switches to a different component at each clock cycle. So we use exactly eight frequency components for each clock cycle. Our solution has a twofold advantage: First, it reduces the area occupancy of the FFT64 unit and the memory parallelism required (eight words vs. 64). Second, it realizes part of the work of the Data Route component, since at every clock cycle we produce eight values which are appropriately spaced out for memory writing.

We also identified a couple of minor optimizations. We merged carry-save vectors immediately after the adder tree, reducing area usage. The carry propagation latency penalty can be mitigated by adding a pipeline stage. Furthermore, before Stage 1, we reduce the bit-width of each value by applying Equation 4. This further decreases the area, particularly routing

resource usage. Last, we recall that the FFT-64 unit can be adapted, with minor modifications, to compute also Radix-8, Radix-16, and Radix-32 FFTs. This gives us greater flexibility in choosing an FFT order other than 64K.

c) Internal banked memory: Our internal memory needs to support the specific FFT memory access pattern yet guarantee an appropriate degree of parallelism. A simple *linear* banked memory ensures parallel read accesses (with consecutive words in a row mapped to different banks) but it would cause write accesses to collide on the same bank. To effectively tackle this issue, we adopted a two-dimensional scheme, shown in Figure 5. Each square is a memory bank, i.e. a dual port SRAM, with a depth of 256 words and word-width of 64 bits, implemented as a native FPGA memory block (namely, two Altera M20K hard core blocks). A 4x4 array of basic memory blocks yields a size of 256Kb which can hold a vector of 4096 points. For visual clarity, the scheme in the figure displays only one of the dual ports in the basic block. Read access is column-wise, while write access is row-wise. Access parallelism is eight words per clock cycle, either during reading or writing.

d) Modular multiplier: The output points of inner FFTs need to be multiplied by appropriate twiddle factors before they can be used by the external FFT. We chose to use DSP blocks for greater efficiency in terms of area and speed. To compute 64x64 multiplications we can split our operands in 32-bit components and use a basic 32x32-bit DSP multiplier, which requires only two DSP blocks. Using school-book multiplication, four 32x32-bit multipliers are needed; partial products are then summed and modular reduced by Equation 4.

e) Data route: The purpose of this component is to properly order the output points coming from the modular multipliers, ensuring their correct writing in memory as well as computing the correct addresses according to the current computation step. As mentioned earlier, the complexity of this component is greatly reduced since part of its job is performed by the FFT-64 unit. In fact, it is just a memory address generator.

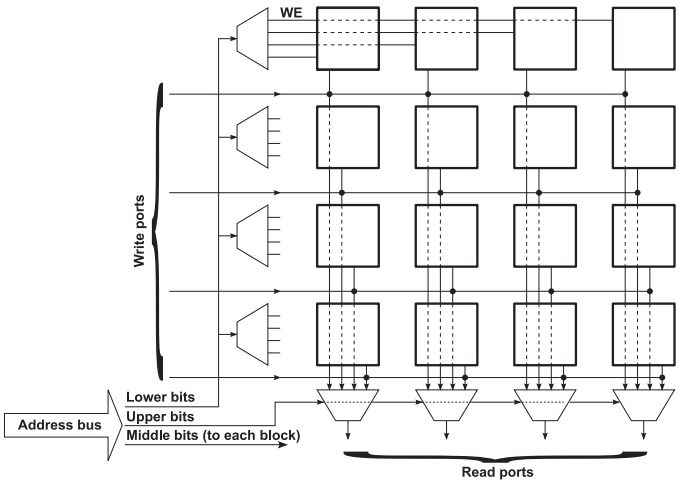

Fig. 5. Architecture of the banked memory buffer.

V. Implementation and performance

To implement the proposed accelerator, we targeted a Stratix V 5SGSMD8N3F45I4 FPGA, as in [28], using VHDL as the design entry language. Most of the implementation effort was put on the Radix-64 unit, the banked memory, and modular multipliers. All of these components were optimized and extensively tested. By carefully pipelining some subsystems, the design could be synthesized at an operating frequency of 200 MHz. Based on this result, we could derive a performance estimate addressing a single FFT and the complete SSA multiplication. The FFT-64 unit is able to output an FFT every eight clock cycles, while an FFT-16 will take two clock cycles. A single 64K-point FFT takes:

$$T_{FFT} = 2 \cdot (T_C \cdot 8 \cdot 1024)/P + (T_C \cdot 2) \cdot 4096/P$$

where T_C is the clock period, i.e. 5 ns, while P is the number of Processing Elements (here, four). The first term refers to the first two stages, with 1024 FFTs on 64 points. The second term refers to the last stage where 4096 FFTs on 16 points are computed. By replacing the clock period and using four Processing Elements, we get:

$$T_{FFT} = 20480ns + 10240ns \approx 30.7\mu s.$$

A full SSA multiplication requires three FFTs (two direct FFTs for the inputs and one inverse FFT for the output). Furthermore, we need a component-wise multiplication on two vectors of 64K components and the final carry recovery addition on the inverse FFT components. The remaining resources can accommodate at least 32 additional modular multipliers for component-wise multiplication, yielding:

$$T_{DOTPROD} = T_C \cdot 65536/32 \approx 10.2\mu s.$$

The final carry recovery can be efficiently computed with an ad-hoc adder structure, not described here due to the lack of space. Its maximum delay is approximately 20μs. Hence, the overall time for a complete SSA multiplication is $\approx 122\mu s$.

Table I compares the proposed solution with [28] both in terms of absolute resource count and as a fraction of the total resources on the target FPGA. Notice that [28], based on the same device as our work, only presents quantitative results for the FFT operation. For our comparison, we conservatively assumed a zero difference for the remaining dot-product and carry recovery operations. Overall, the combination of the optimizations presented above results in around 60% saving in hardware costs. The unused resources might be used to achieve further performance improvements, although this was not exploited in this comparison. Table II presents the performance of our solution vs. [28], [30], a 90nm ASIC solution, and [26], [27], which are based on NVIDIA C2050 GPUs. The execution time of [28] is 3.32X larger than the time taken by our solution, while the other results are 1.69X larger, or more.

TABLE I. Comparison of resource usage.

	Proposed here	[28]
ALMs	104000 (40%)	231000 (88%)
Registers	116000 (11%)	336377 (31%)
DSP blocks	256 (13%)	720 (37%)
M20K SRAM	8 Mbit (20%)	–

TABLE II. Comparison of excution time.

	Proposed here	[28]	[30]	[26]	[27]
FFT (μs)	30.7	125	–	250	–
Multiplication (μs)	122	405	206	765	583

VI. Conclusion

This work presented an FPGA implementation of a dedicated hardware accelerator for ultralong-operand multiplication as a basic building block of a homomorphic encryption processor. The paper described the high-level architectural concept and implementation as well as the experimental data collected from hardware synthesis. The results point out the great potential posed by FPGA technologies in the acceleration of compute-intensive cryptographic algorithms.

Acknowledgments

The first version of the homomorphic encryption accelerator presented in this work, based on Altera Cyclone V devices, received the 2015 Altera Innovate Europe SoC award.

The work was partially supported by the European Commission in the framework of the H2020-FETHPC-2014 project n. 671668 - *MANGO: exploring Manycore Architectures for Next-GeneratiOn HPC systems*.

The authors are also thankful to Stefano Marano for supporting the development and test of the FFT unit.

References

[1] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[2] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," *EUROCRYPT*, pp. 24–43, 2010.

[3] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-LWE and security for key dependent messages," *CRYPTO*, pp. 505–524, 2011.

[4] K. Paranjape, S. Hebert, and B. Masson, "Heterogeneous computing in the Cloud: Crunching Big Data and democratizing HPC access for the life sciences," Intel Corporation, Tech. Rep., 2010.

[5] A. Putnam and other, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=212001

[6] Çetin K. Koç, *Cryptographic Engineering*. Springer Science & Business Medi, 2008.

[7] D. Suzuki, "How to maximize the potential of FPGA resources for modular exponentiation," in *Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, ser. LNCS, vol. 4727. Springer, 2007, pp. 272–288.

[8] G. de Meulenaer, F. Gosset, M. M. de Dormale, and J.-J. Quisquater, "Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware," in *Proceedings of the 15th Annual Symposium on Field-Programmable Custom Computing Machines (FCCM) 2007*. IEEE, 2007, pp. 197–206.

[9] S. Ghosh, D. Mukhopadhyay, and D. Roychowdhury, "High speed flexible pairing cryptoprocessor on FPGA platform," in *Proceedings of the 4th international conference on Pairing-based cryptography*. Springer-Verlag, 2010, pp. 450–466.

[10] T. Güneysu, T. Kasper, M. Novotný, C. Paar, and A. Rupp, "Cryptanalysis with COPACOBANA," *IEEE Transactions on Computers*, vol. 57, no. 11, pp. 1498–1513, 2008.

[11] (2011, Jan.) Rivyera S3-5000. [Online]. Available: http://www.sciengines.com/

[12] A. Cilardo and N. Mazzocca, "Exploiting vulnerabilities in cryptographic hash functions based on reconfigurable hardware," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 5, pp. 810–820, May 2013.

[13] A. Cilardo, M. Barbareschi, and A. Mazzeo, "Secure distribution infrastructure for hardware digital contents," *Computers Digital Techniques, IET*, vol. 8, no. 6, pp. 300–310, 2014.

[14] A. Cilardo, A. Mazzeo, L. Romano, and G. Saggese, "An FPGA-based key-store for improving the dependability of security services," in *Object-Oriented Real-Time Dependable Systems, 2005. WORDS 2005. 10th IEEE International Workshop on*, Feb 2005, pp. 389–396.

[15] M. Barbareschi, E. Battista, A. Mazzeo, and S. Venkatesan, "Advancing wsn physical security adopting TPM-based architectures," in *Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on*. IEEE, 2014, pp. 394–399.

[16] M. Barbareschi, P. Bagnasco, and A. Mazzeo, "Supply voltage variation impact on anderson PUF quality," in *Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2015 10th International Conference on*. IEEE, 2015, pp. 1–6.

[17] Y. K. Sinjilawi, M. Q. AL-Nabhan, and E. A. Abu-Shanab, "Addressing security and privacy issues in Cloud Computing," *Journal of Emerging Technologies in Web Intelligence*, vol. 5, no. 2, pp. 192–199, 2014.

[18] F. Moscato, F. Amato, A. Amato, and R. Aversa, "Model-driven engineering of cloud components in metamorp(h)osy," *International Journal of Grid and Utility Computing*, vol. 5, no. 2, pp. 107–122, 2014.

[19] F. Amato, A. Mazzeo, V. Moscato, and A. Picariello, "Exploiting cloud technologies and context information for recommending touristic paths," *Studies in Computational Intelligence*, vol. 511, pp. 281–287, 2014.

[20] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," *Proc. Advances in Cryptology-EUROCRYPT 2011*, pp. 129–148, 2011.

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

[22] J. Coron, T. Lepoint, and M. Tibouchi, "Batch fully homomorphic encryption over the integers," *IACR Cryptology ePrint Archive*, vol. 2013, p. 36, 2013.

[23] H. Perl, M. Brenner, and M. Smith. (2011) hcrypt. [Online]. Available: http://www.hcrypt.com/scarab-library/

[24] S. Halevi and V. Shoup. (2012) Helib, homomorphic encryption library. [Online]. Available: https://github.com/shaih/HElib

[25] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," vol. 99, p. 1, 2013.

[26] ——, "Accelerating fully homomorphic encryption using GPU," in *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, Sept 2012, pp. 1–5.

[27] ——, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, 2015.

[28] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," *ISCAS*, pp. 2589–2592, 2013.

[29] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," *Digital System Design (DSD),16th Euromicro Conference on*, 2013.

[30] W. Wang, X. Huang, N. Emmart, and C. Weems, "VLSI design of a large-number multiplier for fully homomorphic encryption," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, p. 18791887, 2014.

[31] Y. Doröz, E. Öztürk, and B. Sunar, "Accelerating fully homomorphic encryption in hardware," *draft, Under Review*, 2013.

[32] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High speed fully homomorphic encryption over the integers," *Workshop on Applied Homomorphic Cryptography, to appear*, 2014.

[33] J. Coron, A. Mandal, D. Naccache, and M. Tibouchi, "Fully homomorphic encryption over the integers with shorter public keys," *CRYPTO*, pp. 487–504, 2011.

[34] J. Coron, D. Naccache, and M. Tibouchi, "Public key compression and modulus switching for fully homomorphic encryption over the integers," *EUROCRYPT*, pp. 446–464, 2012.