Embedded System Design Seminar

Hardware Security

Seminar Report

Crypto-processor Design

Shubham Paul

Supervised By

Prof. Dr. Rainer Leupers

Chair for Software for Systems on Silicon (SSS)

Institute For Communication Technologies And Embedded Systems (ICE)

RWTH Aachen

Contents

Chapter 1

Introduction

Increased usage of cloud platforms to outsource processing of data is great way to reduce local infrastructure and maintenance cost. This has become the need of the day with increasing software complexity and requirement of a high-performance machine for rapid prototyping. With this increased use of cloud computers comes the essential requirement of privacy and security. Although, we have encryption algorithms that provide a certain amount of security from indirect attacks (i.e. attackers don't have a direct access to the machines e.g. man-in-the-middle attack), they provide no security if the attacker has direct access (e.g. employees of a cloud services company) to the machine since they have access to the key-pairs of the encryption algorithm.

In order to make the outsourcing of data secure from the cloud platform itself, there is a need for a certain type of hardware processing unit that can directly perform computations on encrypted data without needing to decrypt it first. Such a type of processing unit is called a homomorphic Crypto-processing-unit and the type of encryption algorithms that allow such computations is called Homomorphic encryption.

This report explores the literature of the hardware designs and performance of such (homomorphic) crypto-processors based on traditional encryption algorithms (e.g. variants of AES) and designs based on Homomorphic encryptions.

Chapter 2

Related work

[1] Gentry, made the breakthrough by using a method of "bootstraping" a 'Somewhat homomorphic encryption' (SWHE or SHE) to evaluate its own decryption circuit and hence reduce the 'noise' which resulted in a working 'Fully homomorphic encryption' (FHE). The first generation implementations (e.g. by IBM [13]) were processing 1-bit operations per second with a million bit block which were quite impractical.

Since then, there have been software optimizations in form of libraries such as HElib[2], FV-NFLlib[20] and also different variants of new SWHE and FHE algorithms such has the FV-SHE scheme[3] and TFHE[21] have been developed.

Since these were purely algorithmic desgins and were genreally slow, there is an ongoing effort at making the desgin faster by optimizaing the software and using readily available set of CPUs and GPUs as a high-performance machine. This is shown in [4] where the authors use a variant of FHE called TFHE[21] and optimise it for multi-core CPUs and then using a parallel framework, port the implementation for GPUs to achieve performance gain.

There have been a focus on the Hardware architures (of homomorphic crypto-processors) either by using a variant of FHE or SHE or by choosing a 'pseudo-homomorphic' approach. In this report, we would explore two architecures, one[11] that uses a pseudo-homomorphic processor desgin and implements a KPU based on Rijndael-64[6] algorithm and another[5] that uses a FPGA based custom parallel archictecture and uses a SWHE based on FV-SHE scheme[3].

A practical (and cost effective compared to GPUs) cloud FPGA implementation in Amazon AWS F1 based on [5] is shown in [22] that benchmarks the homomorphic computation of an artificial neural network in a simple forecasting algorithms for SmartGrids.

Chapter 3

Pseudo-Homomorphic Processor Design

The crypto-processing unit (or as the authors call it, the 'KPU') design shown in [11] achieves the property of performing ALU operations on encrypted data by adding certain hardware units in a RISC CPU[9]. These hardware units are called 'codecs'. In 2013, [10] gave the general architecture of embedding so called 'codecs' in ALU and Memory for decryption of data internally, processing via the ALU and then followed by a re-encryption. The authors of [10] then implemented a KPU design based on the original paper[10] in [11] and the design is simulated to show the performance compared to an unmodified RISC pipeline in the same CPU.

3.1 Architecture

3.1.1 Modified ALU and Codecs

The general idea as shown in [11] is to modify a 32-bit RISC CPU ALU core to work on Rijndael-64 encrypted data. There are two different kinds of hardware codecs, one for encryption and one for decryption and each codec has a secret key of encryption/decryption embedded in it. Although this is a point of security concern, an idea of using a Smartcard[12] like technology has been raised. The idea is to have a physical overlay of the processor's codecs (the part that contains the encryption/decryption algorithm and the secret keys) which when tampered with a physical probe cause the processor to stop working permanently. These codecs (named D and E) are added to the ALU's input (i.e. before it receives the data) and at its output (i.e after it has computed on the received

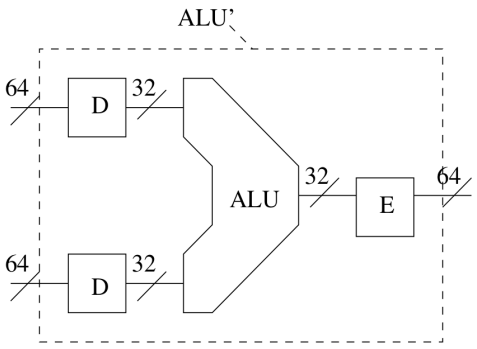

Figure 3.1: Modified 32-bit ALU with 'D' and 'E' codecs [11].

data) respectively as shown in Fig. 3.1. Data addresses and Data in the memory (heap and registers) are always encrypted. The instructions are of 32-bit, processing encrypted data of size 64-bit that consists of 32-bit of actual data and 32-bit of padding.

Two modes named 'user' (encrypted) and 'supervisor' (unencrypted) are used. The supervisor mode has the standard 5-stage RISC pipeline[9] and is used to compare the performance against the user mode. The user mode uses a modified (extended) pipeline structure to improve the performance. The structure is shown in Fig. 3.2.

3.1.2 Dual-config. pipeline

There are two configurations available with this pipeline. Config. 'A' and Config. 'B' has codec used just before the write stage and right after the 'decode' stage respectively. There is also a set of 'shadow registers' available in the ALU (only visible for user-mode) that store the unencrypted version of data available in the general purpose registers. When an instruction doesn't need a codec then Config. 'A' is used due to its early execution capabilities. When processing consecutive chain of arithmetic operations using the pipeline modes and the shadow registers, only a start of the sequence is processed with a decryption and the end of the sequence is processed with an encryption. This results in gain of performance due to less frequent codec calls. Since data addresses are encrypted and 1-n nature of Rijndael-64[6], encrypting the same memory location can result in two different ciphertexts and the memory unit maps it to different locations, which can lead to problem of 'hardware aliasing' as discussed in [11]. Therefore the encryption has to processed in a deterministic way[11] when encrypting data addresses or data in general.

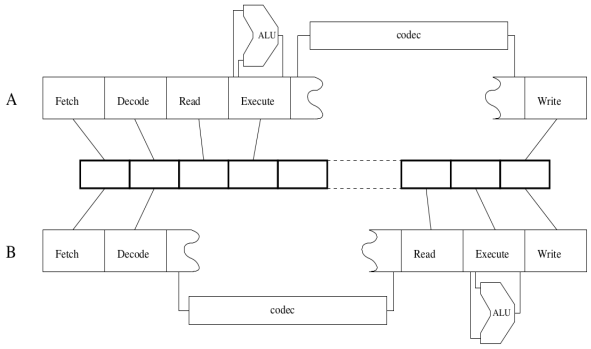

Figure 3.2: Pipeline configurations 'A' and 'B' for user-mode instructions [11].

3.2 Performance

The KPU is simulated on a modified OpenRISC 'or1ksim' simulator (github.com/openrisc /or1ksim). When using a clock of 1GHz, there is a 90% pipeline usage at 892Kips in supervisor mode. In user mode, with the same 1Ghz clock there is a 54.9% pipeline usage at 549Kips. Hence the performance of the user (encrypted) mode is 61.6% of the supervisor (unencrypted) mode.

Chapter 4

Homomorphic Encryption

In order to understand the algorithm on which [5] is based on, a brief overview of homomorphic encryption is needed. Homomorphic encryption is a type of encryption that allows mathematical computations (additions and/or multiplications) directly on ciphertexts without the need of decrypting the data back into plaintext. As an example if we have a plaintext x and another plaintext y and the corresponding ciphertext with E (homomorphic encryption) are $E(x)$ and $E(y)$ then there would be a function F_* such that:

$$F_*(E(x), E(y)) = E(x * y) \tag{4.1}$$

Where $*$ operation symbolizes addition or multiplication. Hence applying decryption D we get:

$$D(E(x * y)) = x * y \tag{4.2}$$

We say that the operation $*$ is performed homomorphically.

4.1 Types

There are three (main) types[7] of homomorphic encryptions:

- Partially homomorphic encryptions (PHE): These encryptions allow unlimited number of either addition or multiplication operations. For example, RSA[8] is homomorphic with multiplication and Pallier encryption[9] is homormorphic with addition.
- Somewhat homomorphic encryptions (SWHE): These allow a limited number of both addition and multiplcations operations. Going beyond the number results in the increment of noise to such level that decryption becomes impossible.
- Fully homomorphic encryptions (FHE): These allow unlimited number of both addition and multiplcations operations. One of the ways these can be obtained is from a SWHE using a process called 'bootstrapping'[1].

4.2 Lattice based cryptography

The security of all encryptions are based on the 'hardness' of a mathematical property. For example, RSA [8] is strong based on the fact that it is difficult to factor the product of two large prime number using classical computers in polynomial time. Although it could be broken with Shor's algorithm[10] in polynomial time using a quantum computer.

When cryptographic algorithms use 'lattices' (group theory) in their constructions or as a basis to 'hard' problems solvable with 'lattices' as their security measure then such algorithms come under the category of Lattice based cryptography.

Certain (for e.g. FV-SHE scheme[3]) FHEs or SWHE are based on Learning with errors (LWE) (or LWE over a ring, RLWE) problem which with lattice reductions comes down to a problem called Shortest Vector Problem (SVP)[14] which is 'hard' to solve. Lattice based cryptographic algorithms are also a candidate of post-quantum cryptography[15], which means that even a quantum computer cannot break the security in polynomial time.

A basic LWE problem is:

Given a random uniformly chosen matrix A with elements chosen as a modulo integer q. That is $A \in Z_q[n \times m]$. Also, s and e are short vectors, then:

$$b = LWE_A(s, e) = As + e \ (mod\ q) \tag{4.3}$$

Hence, given $[A, b]$, it is 'hard' to recover (s, e). For encryption s is used as a secret key and e is the added noise. A simple secret key encryption on message m is done using:

$$E_s(m; [A, e]) = [A, b + m] \tag{4.4}$$

Decryption is noisy as:

$$D_s([A, b + m]) = (b + m) - As = m + e \ (mod\ q) \tag{4.5}$$

The decrypted message has a noise e added to m (corrupts the low-order bits) and can be recovered by scaling m followed by rounding, provided e is not too large. RLWE is a LWE with polynomial rings. [5](FPGA based Crypto-processor desgin) uses FV-SHE[3] which is based on RLWE.

Chapter 5

FPGA based Coprocessor Design

The crypto-processor design shown in [5] implements a custom parallel architecture on a FPGA+ARM heterogeneous platform. The architecture exploits the parallelism of the polynomial ring LWE based FV-SHE[3] algorithm with residue number system based modulo arithmetic and Number theoretic transform for large integer polynomial multiplications. This design skips the bootstrapping procedure (used to convert SWHE/SHE to FHE), which is computationally expensive and provides an SWHE based crypto-processor that has a multiplicative depth (maximum number of homomorphic multiplications after which the ciphertext becomes too noisy to decrypt) of four.

5.1 Algorithm

The FV-SHE[3] algorithm has computations in a polynomial ring $R = \mathbb{Z}[x]/(x^{\mathrm{n}} + 1)$, where n is a power of 2 for a n-bit message. The message can be viewed as a polynomial with $0, 1$ as coefficeints[23]. The encryption process is similar to LWE but now all the variables are represented using polynomial rings and uses public-key cryptography. The ring is modulo-q ring (reduced by q) and shown as R_q. The encrytption and decryption process is shown in Fig. 5.1. All the variables are polynomials with coefficients in R_q and are of degree $n-1$. The public key is (p_0, p_1) and the secret key is s. The encryption process invloves encoding of a n-bit message m and sampling of error(or noise) polynomials (u, e_1, e_2) to generate ciphertext polynomial pair (c_0, c_1). The decryption is noisy as with LWE algorithms.

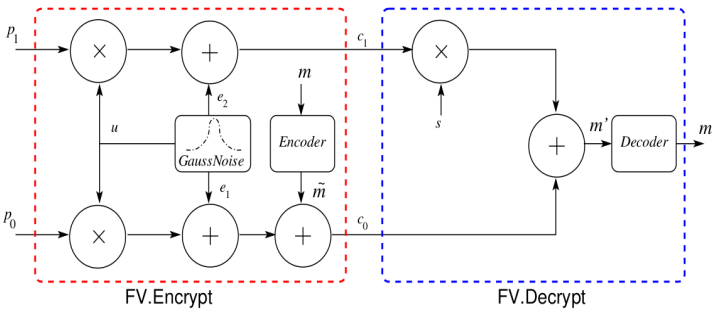

Figure 5.1: Encryption and Decryption under FV-SHE [5].

FV-SHE[3] has special 'Add' and 'Mult' functions (compared to RLWE public key encryption) that allow homomorphic additions and multiplications on encrypted data. Since the ciphertexts are polynomials[3][5], addition is a simple polynomial addition. Assuming the ciphertexts polynomials are $c_0 = (c_{0,0}, c_{0,1})$ and $c_1 = (c_{1,0}, c_{1,1})$. Addition results in $c_{addition} = (c_{0,0} + c_{1,0}, c_{0,1} + c_{1,1})$ The polynomial multiplication of two ciphertexts is as shown in the Fig. 5.2.

The incoming ciphertexts c_0 and c_1 have polynomial coefficients in the mod-q ring (R_q) which are then lifted to a larger modulus Q ring (R_Q) in the $Lift_{q \to Q}$ to have sufficiently large modulo for the polynomial multiplication. The incoming coefficients (R_q) are in RNS (residue number system) representation with six 30-bit coprimes (q is the product of six primes and hence it is 180-bit) hence they need to be merged using the 'Chinese Remainder Theorem (CRT)' and then brought to RNS of Q (R_Q) by computing the modulo with the additional seven coprimes. Therefore Q is a 390-bit in size (product of 13 primes). Since using traditional CRT is expensive, a different method[16], referred in [5] as the 'HPS method' is used to compute approximate CRT which avoid long integer arithmetic. After the lifting the polynomials, Number Theoretic Transform (similar to Fast Fourier transform but performs inter arithmetic) is applied which reduces polynomial multiplication (similar to FFT-based method[17]) to simple coefficient-wise multiplications, since numbers are in RNS mod-Q, long integer multiplications reduces to small integer modulo multiplications which can be performed in parallel. INTT (Inverse Number Theoretic Transform) is then applied and $Scale_{Q \to q}$ scales down the result back to R_q. The polynomials uses 4096 coefficients.

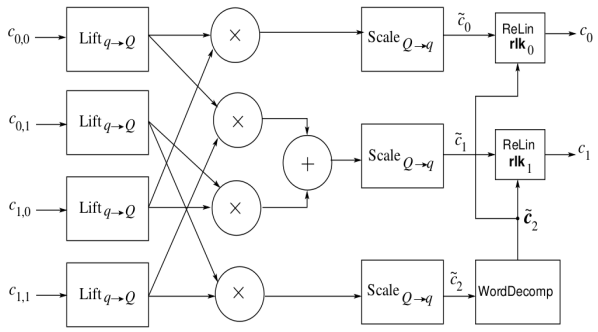

Figure 5.2: Homomorphic multiplication of ciphertexts c_0 and c_1 under FV-SHE [5].

5.2 Architecture

5.2.1 Residue Polynomial Arithmetic Unit (RPAU)

This unit leverages the inherent parallelism in RNS by computing multiple parallel modulo multiplications within each RPAU. There are 7 (instead of 13) RPAUs for performing small modulo multiplications for every prime q_i in RNS representation of a number in order to increase hardware reuse. A R_q number takes a single batch while a R_Q number takes two batches to finish one computation.

5.2.2 Memory

A BRAM (block RAM) with 1024 elements of size 36-bit per element is used. Two ports are used simultaneously, one for writing and one for reading. The 'butterfly operation' of the iterative NTT operation mentioned in [17][5] uses a pair of coefficients and generates a new pair of coefficients. Hence, using 2-NTT butterfly cores, two pairs of coefficients are read, processed and generated every cycle. The final memory architecture uses a stacked 2-over-2 1024 blocks (two BRAMS are aligned on top of each other with the same memory address range), resulting in a upper and lower block with each block having 1024*2 (2048) words. A single read automatically reads in a pair of coeffients from the stacked memory block. The 2-NTT cores always read/write data from different blocks. Fig. 5.3 shows the memory access by the dual core NTT with R and R' showing reads by different NTT cores for the respective iteration of the iterative NTT algorithm[17][5]. For iteration $m = 2048$, the memory address of the second core is inverted to prevent conflicts.

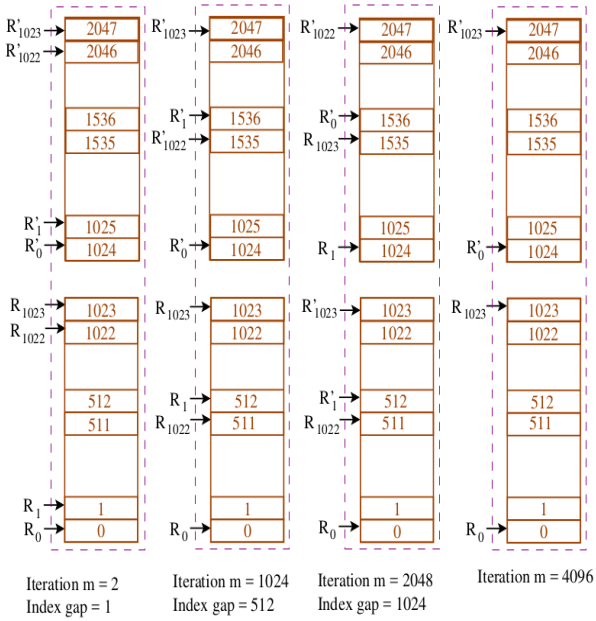

Figure 5.3: Memory access by the two NTT cores [5].

5.2.3 NTT-core

The NTT butterfly core used can perform forward and inverse NTT and arithmetic modulo operations. The core is similar to the one presented in [18] with minor differences. One such difference is loading the constant twiddle factors from memory instead of calculating them on the fly leading to an increase in performance. The integer multiplier (implemented using DSP slices) produces a 60-bit integer and it is then reduced using modular reduction (by prime q_i). A generic modular reduction circuit is implemented using a reduction table and an iterative sliding window. The arithmetic units and the multiplier are also pipelined. A single NTT core is shown in Fig. 5.4

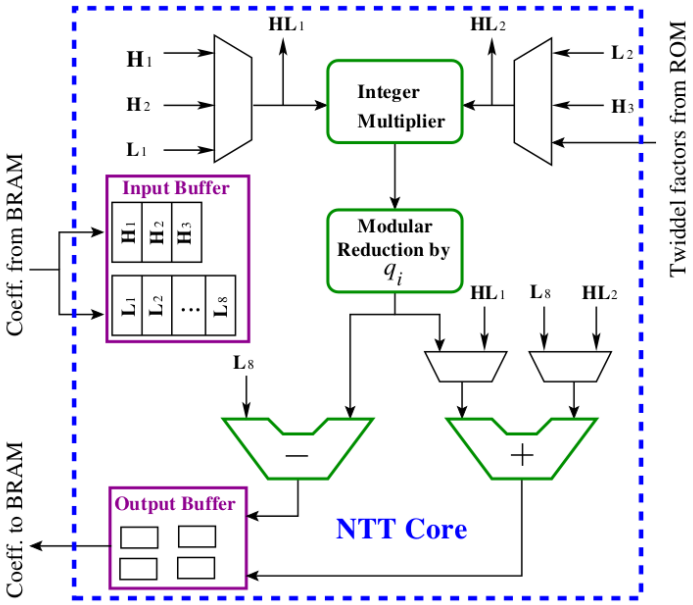

Figure 5.4: Architecture of a NTT core [5].

5.2.4 $\text{Lift}_{q\to Q}$, $\text{Scale}_{Q\to q}$

The $\text{Lift}_{q\to Q}$ and $\text{Scale}_{Q\to q}$ blocks use the implementations from [16] to avoid long integer arithmetic. The independent SOP (sum of products) expressions are computed in parallel using MAC (multiply and accumulate) blocks. A generic block is implemented to be used with accumulation or without accumulation as shown in Fig. 5.5 to increase hardware utilization.

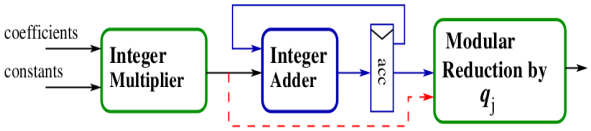

Figure 5.5: Architecture of generic multiplication block with or without accumulation [5].

5.2.5 Overall Architecture

Within a coprocessor, there are 7 RPAUs with dual NTT cores and memories (eight for storing coefficients and 1 filled with zeros). The $\text{Lift}_{q\to Q}$ and $\text{Scale}_{Q\to q}$ blocks are used by all the 7 RPAUs. Fig. 5.6 shows the architecture of a coprocessor.

Two coprocessor instances run on a FPGA. Two ARM cores are used to run baremetal (without an OS) application software and offload the homomorphic computation to the respective coprocessors. The third ARM core is used to manage network connectivity between clients and the application cores. DMA is used to achieve high speed continuous data access between the hardware (FPGA) and the memory (DDR). Fig. 5.7 shows the high level architecture.

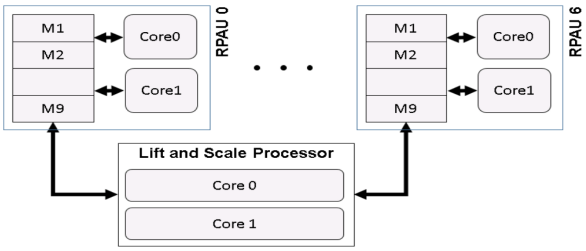

Figure 5.6: Architecture of a coprocessor [5].

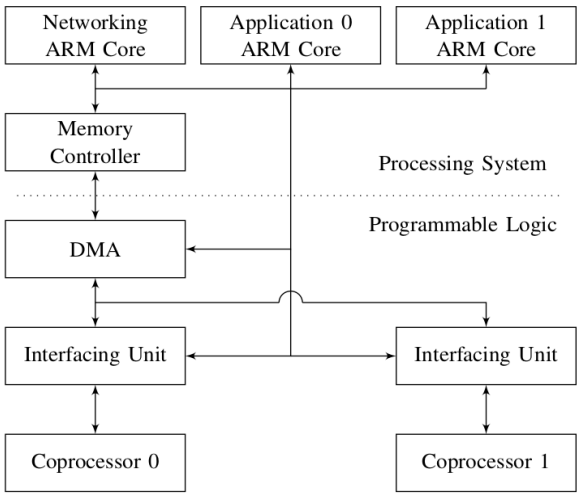

Figure 5.7: High level Architecture of the heterogeneous FPGA+ARM platform [5].

5.3 Performance

The FPGA+ARM platform used for [5] is Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit [19]. The FPGA coprocessor core runs at 200MHz and the ARM core runs at 1.2GHz. DMA is clocked at 250MHz. As shown in table 5.1 a single coprocessor computes a homomorphic multiplication operation in 4.458 miliseconds. Similary a homomorphic addition takes 0.026 miliseconds. Using two coprocessors paralelly gives around 2x of throughput. The setup computes 400 homomorphic multiplications per second which includes the communication overheads. This speed was achieved for a polynomial size of 4096 and a modulus q of 180-bit. This implementation of FV-SHE on FPGA (running at 200Mhz) has a 13x speedup over the use of FV-NFLlib[20] running on an Intel core-i5 CPU clocked at 1.8GHz.

Table 5.1: Performance with a single co-processor [5].

Operation	**Speed**	
	(cycles)	(msec)
Mult in HW	5,349,567	4.458
Add in HW	31,339	0.026
Add in SW	54,680,467	45.567
Send two ciphertexts to HW	434,013	0.362
Receive result ciphertext from HW	215,697	0.180

Chapter 6

Comparisons

The first desgin[11] compares the performance based on simulation results. Since, this particular processor design is not using any homomorphic encryption (FHE or SWHE) algorithm, it doesn't have to deal with 'encryption noise' which often limits the computation dept of SWHE or often requires a bootstrapped SWHE (i.e. a FHE), which is again expensive to calculate. Moreover, since the general design is not algorithm specific, a new algorithm can be implemented with minimal changes just by changing the codecs which makes it flexible to implement with modern encryption algorithms.

The second desgin[5] implements the design on a prototyping hardware and achieves a high-performance results that is practical to be deployed in cloud services. The design is based on an SWHE and therefore largely suffers from 'encryption noise' which restricts the number of homomorphic computations on a ciphertext. Even then, since the architecture is highly parallel, this designs offers a high number of concurrent and power efficient computations, about 400 homomorphic multiplications per second (for 2 parallel coprocessors). A practical implementation is also shown in [22] which uses 6 parallel coprocessors and achieves 600 homomorphic multiplications per second in a cloud platform. From the shown results one can observe that increasing the number of parallel coprocessors doesn't increase the throughput in a linear way and it is mostly due to the time sharing of the data-transfer interface. Regardless, the implementation is around 3 times more[22] cost effective than GPUs. It is bench-marked on a basic real world forecasting application.

[11] and [5] take very different approaches when building a crypto-processor and hence are not directly comparable except for the fact that using lattice-based crytography (the FV-SHE method used in [5]) makes the algorithm secure even from a quantum computer. It would be interesting to compare the improvement of performance results of both the designs if [11] is optimized further and is fabricated into an actual CPU and [5] is fabricated into specialized ASICs.

Chapter 7

Conclusion

This report highlighted two different classes of hardware based homomorphic crypto processors, a pseudo-homomorphic design and a SWHE (somewhat homomorphic encryption) based design which is then implemented on a FPGA. The custom pipelined hardware architecture (i.e the first design) was explored and the simulation performance was presented. A brief overview of different types of homomorphic encryptions were presented and the concept of lattice based cryptography was explained with an example of Learning with Errors (LWE) problem as this problem forms the basis of the second FPGA based design. Different architectural features such as memory management and different levels of parallelism were summarized and finally the performance comparison is shown. A feature comparison of both the designs is shown.

Since FHE/SWHE algorithms are lattice problems and are hence considered a candidate for post-quantum cryptography, they provide very high levels of security against attacks and hence the focus on improving the performance by improving the underlying hardware becomes an important criteria. With an increasing number of optimizations in software/hardware co-designs, homomorphic crypto processors are becoming faster and more efficient. Recent[22] implementations of such crypto processors on cloud platforms running basic real world applications have generated promising results. With such developments, outsourcing data processing on cloud would become more practical in terms of security.

References

[1] C. Gentry, *Fully homomorphic encryption using ideal lattices*, New York, NY, USA: ACM, 2009, pp. 169–178.

[2] S. Halevi and V. Shoup, *Algorithms in helib*, in Annual Cryptology Conference. Springer, 2014, pp. 554–571.

[3] J. Fan and F. Vercauteren, *Somewhat practical fully homomorphic encryption*, Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/.

[4] T Morshed, Md Aziz and N. Mohammed, *CPU and GPU Accelerated Fully Homomorphic Encryption*, https://arxiv.org/abs/2005.01945, 2020.

[5] S. Roy, F. Turan, K. Järvinen, F. Vercauteren and I. Verbauwhede , *FPGA-Based High-Performance Parallel Architecture for Homomorphic Computing on Encrypted Data*, in IEEE International Symposium on High Performance Computer Architecture, 2019.

[6] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*, Springer Verlag, 2002.

[7] M. Ogburn, C. Turner and P. Dahal, *Homomorphic Encryption*, Procedia Computer Science 20(2013) 502–509.

[8] R. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Communications of the ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[9] P. Paillier, *Public-key cryptosystems based on composite degree residu- osity classes*, in Advances in cryptology – EUROCRYPT'99. Springer, 1999, pp. 223–238.

[10] Shor, P.W., *Algorithms for quantum computation: discrete logarithms and factoring*, Proceedings 35th Annual Symposium on Foundations of Computer Science. IEEE Comput. Soc. Press: 124–134, 1994.

[11] P. Breuer and J. Bowen, *A First Practical Fully Homomorphic Crypto-Processor Design*, https://arxiv.org/abs/1510.05278v2, 2016.

[12] O. Kömmerling and M. G. Kuhn, *Design principles for tamper-resistant smartcard processors*, in Smartcard '99, Chicago, Illinois, USA, May 10-11, 1999, May 1999, pp. 9–20.

[13] C. Gentry and S. Halevi, *Implementing Gentry's fully-homomorphic encryption scheme*, in Advances in Cryptology – EUROCRYPT 2011, ser. Lecture Notes in Computer Science, K. Paterson, Ed. Springer Berlin Heidelberg, 2011, vol. 6632, pp. 129–148.

[14] Oded Regev, *On lattices, learning with errors, random linear codes, and cryptography*, in Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (Baltimore, MD, USA: ACM, 2005), 84–93.

[15] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Y. Liu, *Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process*, Tech. Rep. 8240, National Institute of Standards and Technology, Jan. 2019.

[16] S. Halevi, Y. Polyakov, and V. Shoup, *An improved RNS variant of the BFV homomorphic encryption scheme*, 2018. https://eprint.iacr. org/2018/117.

[17] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, McGraw-Hill Higher Education, 2001.

[18] S. Sinha Roy, F. Vercauteren, N. Mentens, D. D. Chen, and I. Ver- bauwhede, *Compact ring-LWE cryptoprocessor*, in Cryptographic Hardware and Embedded Systems CHES 2014, Springer Berlin Hei- delberg, 2014.

[19] Xilinx, *ZCU102 Evaluation Board User Guide*, 2017. v1.3.

[20] CryptoExperts, *FV-NFLlib*, https://github.com/CryptoExperts/ FV-NFLlib, 2016.

[21] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachene, *Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds,*, in International Conference on the Theory and Application of Cryptology and Information Security. Springer, 2016, pp. 3–33.

[22] F. , S. Roy and I. Verbauwhede, *HEAWS: An Accelerator for Homomorphic Encryption on the Amazon AWS FPGA*, IEEE transactions on computers, vol. 69, no. 8, 2020.

[23] V. Lyubashevsky, C. Peikert and O. Regev, *On Ideal Lattices and Learning with Errors Over Rings*, Advances in Cryptology – EUROCRYPT 2010.