New Approaches for Efficient Fully Homomorphic Encryption

by

Yarkın Doröz

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Electrical and Computer Engineering

by

24 May 2017

APPROVED:

Professor Wayne Burleson
Dissertation Committee
University of Mass., Amherst
ECE Department

Associate Professor Thomas Eisenbarth
Dissertation Committee
ECE Department

Professor Yehia Massoud
Head of Department
ECE Department

Professor Berk Sunar
Dissertation Advisor
ECE Department
Abstract

In the last decade, cloud computing became popular among companies for outsourcing some of their services. Companies use cloud services to store crucial information such as financial and client data. Cloud services are not only cost effective but also easier to manage since the companies avoid maintenance of servers. Although cloud has its advantages, maintaining the security is a big concern. Cloud services might not have any malicious intent, but attacks targeting cloud systems could easily steal vital data belong to the companies. The only protection that assures the security of the information is a strong encryption. However, these schemes only protects the information but prevent you to do any computation on the data. This was an open problem for more than 30 years and it has been solved recently by the introduction of the first fully homomorphic encryption (FHE) scheme by Gentry. The FHE schemes allow you to do arbitrary computation on an encrypted data by still preserving the encryption. Namely, the message is not revealed (decrypted) at any given time while computing the arbitrary circuit. However, the first FHE scheme is not practical for any practical application. Later, numerous research work has been published aiming at making fully homomorphic encryption practical for daily use, but still they were too inefficient to be used in everyday practical applications.

In this dissertation we tackle the efficiency problems of fully homomorphic encryption (FHE) schemes. We propose two new FHE schemes that improve the storage requirement and runtime performance. The first scheme (Doröz, Hu and Sunar) reduces the size of the evaluation keys in existing NTRU based FHE schemes. In the second scheme (F-NTRU) we designed an NTRU based FHE scheme which is not only free of costly evaluation keys but also competitive in runtime performance.
We further proposed two hardware accelerators to increase the performance of arithmetic operations underlying the schemes. The first accelerator is a custom hardware architecture for realizing the Gentry-Halevi fully homomorphic encryption scheme. This contribution presents the first full realization of FHE in hardware. The architecture features an optimized multi-million bit multiplier based on the Schönhage-Strassen multiplication algorithm. Moreover, a number of optimizations including spectral techniques as well as a precomputation strategy is used to significantly improve the performance of the overall design. The other accelerator is optimized for a class of reconfigurable logic for somewhat homomorphic encryption (SWHE) based schemes. Our design works as a co-processor: the most compute-heavy operations are offloaded to this specialized hardware. The core of our design is an efficient polynomial multiplier as it is the most compute-heavy operation of our target scheme. The presented architecture can compute the product of very-large polynomials more efficiently than software implementations on CPUs.

Finally, to assess the performance of proposed schemes and hardware accelerators we homomorphically evaluate the AES and the Prince block ciphers. We introduce various optimizations including a storage-runtime trade-off. Our benchmarking results show significant speedups over other existing instantiations. Also, we present a private information retrieval (PIR) scheme based on a modified version of Doröz, Hu and Sunar’s homomorphic scheme. The scheme is capable of privately retrieving data from a database containing 4 billion entries. We achieve asymptotically lower bandwidth cost compared to other PIR schemes which makes it more practical.
Acknowledgements

I successfully completed my PhD thanks to the fundings by National Science Foundation (NSF) awards CNS-1319130, CNS-1117590 and CNS-1561536.

First, I would like to thank my advisor Professor Berk Sunar for his continuous support on my journey to complete my dissertation. His advise and guidance not just helped me to achieve my goals, but also to attain the best of my skills. I am lucky to have the privilege to work with him. Throughout my PhD, he was not just my mentor but also he was my friend. I hope that our friendship will continue in the future.

I would also like to thank my dissertation committee members Professor Wayne Burleson, Professor Yehia Massoud and Professor Thomas Eisenbarth. They spend their valuable time to read and improve my dissertation. I am grateful for their guidance.

I would like to thank Professor William J. Martin for always being available to discuss theoretical and mathematical background. I thoroughly enjoyed our discussions. He helped me to understand the mathematics behind the cryptographic constructions.

I am grateful for help and support of Vernam Lab members, whom I regard as dear friends. Particularly, I would like to thank Michael Moukarzel for being there for me when I needed. He helped me a lot after I arrived to US for the first time and to get used to American culture. I also thank my lab mates Gizem Selcan Çetin and Wei Dai. We worked on many papers together. I hope they enjoyed working with me, as much as I did working with them. With Gizem, we went to the same university so we are coming from similar culture. She always invited us to her home for dinner and breakfast. She is a wonderful cook and she kept the Vernam Lab people together by throwing out parties. Wei Dai is more than my lab
mate. I see him as a close friend with whom our frequencies match, which does not happen quite often. He even visited me and my family in Turkey where we had so much fun together. I hope we will have other opportunities to travel together in the future. My old friend from high school, Metin Evren Kurtuluş, was always there for me when I needed. He spent many hours with me when I was stressed out. I am grateful for his patience and support.

My parents Ali Tunç Doröz and Ash Doröz have always believed in me, never spared their support, they respected my decisions and wished the best for me. I hope that I was able to fulfill their expectations and become someone whom they can be proud of. I know that they will always provide me the strength and motivation in my life when I need. My grandmother Kamuran Giray raised me and always believed in me, supported me for my decisions to pursue a PhD. I lost her during the first year of my PhD and I still miss her a lot. I wish that she was with me in this journey and see that I am a Dr. now. I always appreciated her and hope that she is in a better place now (RIP).

Finally, I would like to thank my fiancé Yonca Betül Karadeniz for being in my life. She is my best friend, my soulmate and my love. She is an amazing person, she knows how to touch my soul and make me happy. Throughout these years, together we were able to overcome all the obstacles we encountered. We never gave up on each other and find a way to make it work. I feel that I am really fortunate to have her in my life.
Contents

1 Introduction 1
   1.1 Contributions .................................. 3

2 Background 8
   2.1 Overview of Gentry’s Approach on FHE ................. 8
      2.1.1 Gentry’s FHE Scheme ............................ 8
      2.1.2 The Gentry-Halevi FHE Scheme .................. 9
   2.2 NTRU Based FHE Schemes ............................. 12
      2.2.1 Stehlé and Steinfeld’s NTRU Variant ............ 13
      2.2.2 LTV FHE Scheme ................................. 13
      2.2.3 YASHE Scheme ................................. 16
   2.3 LWE Based Fully Homomorphic Encryption Schemes ....... 19
      2.3.1 BGV Scheme ..................................... 19
      2.3.2 GSW Scheme ..................................... 22

3 Proposed New FHE Schemes 24
   3.1 DHS FHE Scheme .................................... 24
      3.1.1 Optimizations .................................... 25
      3.1.2 Description ..................................... 31
      3.1.3 Coping with Noise ............................... 33
5 FHE Hardware Designs

5.1 Implementation of Gentry’s FHE in Hardware

5.1.1 Background

5.1.2 Overview of Our Architecture

5.1.3 Large Integer Architecture

5.1.4 FHE Primitives

5.1.5 Implementation Results

5.1.6 Comparison

5.2 Implementation of DHS FHE in Hardware

5.2.1 Background

5.2.2 Architecture Overview

5.2.3 $2^n \times 2^n$ Polynomial Multiplier

5.2.4 Implementation Results

5.2.5 Comparison

6 Conclusion
List of Figures

3.1 Homomorphic XOR operation ............................ 55
3.2 Homomorphic AND operation ............................ 56

4.1 The Prince cipher ........................................ 83

5.1 Overview of the Full Architecture ....................... 106
5.2 Overview of The Large Integer Multiplier ............... 109
5.3 Stage Reconstruction Unit .............................. 114
5.4 ALU of The Stage Reconstruction Unit .................. 115
5.5 Encryption Architecture ............................... 118
5.6 Encryption Processing Element ........................ 122
5.7 Binary Computation Unit .............................. 126
5.8 Modular Computation Unit ............................ 128
5.9 Recryption Architecture .............................. 130
5.10 Recryption Processing Element ......................... 131
5.11 Modular Adder/Subtractor Circuits ..................... 145
5.12 Multiplier Circuit ..................................... 147
5.13 Architecture for 32-bit Modular Multiplier ............ 149
5.14 Construction of the 4 \times 4 NTT circuit from 2 \times 2 circuits ...... 152
5.15 Construction of the 8 \times 8 NTT circuit iteratively ... 153
5.16 NTT Circuit ................................................................. 154

5.17 The architecture for NTT transformation of a polynomial of degree

\[ N \text{ over } F_p, \text{ where } \lceil \log_2 p \rceil = 32. \] ........................................... 157
List of Tables

3.1 Worst case and average case number of bits log(1/K) required to cut 41
3.2 Hermite Factor estimates for various dimensions n and sizes of q . . . 48
3.3 Estimated security level with BKZ ................................. 49
3.4 Hermite factor δ estimates for security level sec reported in [1]. . . . 50
3.5 Derivation of Row 0 of product ciphertext ........................... 57
3.6 Derivation of Row 1 of product ciphertext ........................... 57
3.7 Comparison of F-NTRU and YASHE ................................. 69
3.8 Parameters (log (n), log (q)) to support depth L evaluations with ω =
  16. .................................................................................. 70
3.9 Multiplicative depth L for security parameter λ = 80-bit security . . 70
3.10 Multiplicative depth L for security parameter λ = 128-bit security . . 71
3.11 Homomorphic multiplication times (msec) for radix selection ω = 16.
  C denotes the number of threads. ................................. 71
3.12 Evaluation key and ciphertext sizes to support depth L evaluation . . 72
4.1 The two settings under which we evaluated AES and timing results
  on Intel Xeon @ 2.9 GHz. .............................................. 77
4.2 Sizes of public-key in various representations with and without opti-
  mization for the two selected parameter settings. ........................ 78
4.3 Number of multiplications and evaluation key sizes for constructions in [2] and ours. ......................................................... 82
4.4 Comparison of the complexity of common lightweight block ciphers in number of rounds ................................................. 86
4.5 Parameters for the AES and Prince implementations. .............. 88
4.6 Performance comparison of Prince against AES implementations. . 89
4.7 Hermite factor and supported circuit depth ($\gamma, d$) for various $q$ and $n$. 97
4.8 Polynomial parameters and Query/Response sizes necessary to support various database sizes $N$. .............................. 101
4.9 Index comparison and data aggregation times per entry ............ 101
4.10 Comparison of query sizes for databases upto $2^{32}$, $2^{16}$ and $2^{8}$ entries . 102
5.1 Assignment Table ......................................................... 112
5.2 Clock Cycle Counts of Functional Blocks .............................. 116
5.3 Arithmetic Operation Timings ............................................ 133
5.4 Area of Hardware Blocks (Millions of Gates) ........................... 135
5.5 Time–Area trade–off ..................................................... 135
5.6 Times in msec (top) and in million cycles (bottom) ................... 136
5.7 Powers of $w$ needed in different levels of NTT circuit ............... 159
5.8 Details of NTT computation in our architecture for 32768 coefficients and 256 multiplier units. ................................. 160
5.9 Virtex-7 XC7VX690T device utilization of the multiplier .......... 162
5.10 Timing results for 32-bit coefficient polynomial multiplier for various degree $N$ sizes .................................................. 163
5.11 Primitive operation timings including I/O transactions. ............. 167
5.12 Comparison of multiplication, relinearization times and AES estimate 168
5.13 Comparison of multiplication, relinearization times and Prince estimate

......................................................... 168
Chapter 1

Introduction

The notion of fully homomorphic encryption (FHE) was introduced by Rivest et al. [3] in 1978 and its existence stayed an open question until recently. In the meantime, numerous schemes were proposed that are called partially homomorphic encryption schemes. These schemes feature limited homomorphic functionality e.g. restriction to evaluate a non-universal set of gates, or prohibitive growth in noise or ciphertext sizes restricting the evaluation depth. Prominent examples of partially homomorphic schemes include Goldwasser-Micali cryptosystem [4] additive homomorphism in $\mathbb{Z}_2$, Elgamal encryption system [5], which provides additive homomorphism in $\mathbb{Z}_q$, Paillier cryptosystem [6] additive homomorphism in $\mathbb{Z}_q$ and RSA cryptosystem [7] multiplicative homomorphism on $\mathbb{Z}_q$.

Later, the first working FHE scheme was constructed by Gentry [8, 9] in 2009. Gentry introduced a crucial operation called bootstrapping which is used to restore the noise to a certain level in a ciphertext to operate one more level of multiplicative circuit. However, the implementation was lacking in performance, i.e. it is taking 32 seconds to run a single bootstrapping. This meant a single AND operation taking half a minute to complete. Besides, the scheme required multi-million-bit operands.
The efficiency bottleneck prevented the use of Gentry’s scheme to match any practical applications. After the introduction of the first FHE scheme, many researchers worked on new constructions to improve the efficiency of the FHE schemes. The integer-based FHE constructions may be found in [10, 11, 12], R/LWE based FHE constructions may be found in [13, 14, 15] and NTRU based FHE constructions may be found in [16, 17, 18].

These recent constructions brought impressive optimization techniques that can reduce the cost of expensive bootstrapping operations and noise growth. Brakerski and Vaikuntanathan (BV) introduced such a method in [13] which is called \textit{modulus switching}. The method is used to mitigate noise growth in ciphertexts by applying it at each multiplicative level to cut down a fixed amount of noise. This reduces the noise growth to single exponential from double exponential with respect to the number of multiplicative levels. This method is adopted in many integer, R/LWE and NTRU based FHE schemes. One such example scheme is presented by López-Alt, Tromer, Vaikuntanathan (LTV) in [16]. The scheme is based on the construction of the modified NTRU [19] scheme by Stehlé and Steinfeld [20]. The authors use the modulus switching along with the key switching technique called \textit{relinearization}, which is also introduced in [13], to mitigate noise growth and prevent the dependency of the secret key on the depth of the circuit. However, the relinearization method brings significant memory requirement since it requires to store evaluation keys to perform the operation.

Another NTRU based FHE, which is called \textbf{YASHE}, is proposed by Bos et al. in [2]. The scheme adopts tensor products (another optimization technique) from [17] to limit the noise growth in the FHE scheme. The optimization is also referred to as the scale-invariant method. Using the method, they were able to remove Decisional Small Polynomial Ratio (DSPR) assumption from the security setting of LTV.
side effect of using the scale invariant method, is that it requires prohibitively large evaluation keys which restricted a practical implementation. Specifically, required evaluation key size is cubic $O(\ell^3)$ with respect to the bit-size of the modulus, i.e. $\ell = \log q$. In order to solve the issue, they implemented another variant YASHE’ where they reduce the evaluation key size to $O(\ell)$, but this modification re-introduces the DSPR assumption.

Recently, Gentry, Shai and Waters [21] proposed a new scheme based on a new approach to LWE-based encryption, i.e. the approximate eigenvector method. The scheme introduces flattening, an alternative to existing noise management techniques, e.g. relinearization, modulus switching, bootstrapping. The system is constructed on matrix additions and multiplications which make it asymptotically faster. The flattening operation decomposes the matrix entries into bits to control the noise growth. For a circuit of depth $L$ with $B$-bounded secret key entries and (flattened) ciphertexts with 0/1 coefficients, the error magnitude is at most $((n + 1) \log q + 1)LB$ where $n$ being equal to the lattice dimension. Although the noise management is better, ciphertexts still take a considerable space $O(n^2 \log(q)^2)$ and as noted by GSW [21] the scheme may not be as efficient as existing leveled schemes in practice.

1.1 Contributions

In this dissertation we try to tackle the efficiency problems of the FHE schemes that we discussed above. Each scheme lacks in performance in some aspects, e.g. where one requires significant memory for evaluation keys, whereas the other takes a long time to complete even a single AND operation. In order to overcome the efficiency problems, we introduce two new FHE schemes. Each scheme targets to
solve different efficiency bottlenecks. We also introduce two hardware accelerators to increase the processing speed of the existing FHE schemes. Moreover, we implement our FHE schemes in software and build various applications for benchmarking. Here is a summary of the contributions.

• **DHS FHE Scheme**

  - We modified the LTV scheme [16] and implemented an optimized bit-sliced version. The parameter selection issue is taken care of by theoretical and experimental results in the field of lattice reduction.

  - A specialized ring structure is introduced to simplify modulus switching, to reduce the size of the public key and to eliminate the need for key switching.

  - The noise growth of the proposed scheme is analyzed over the levels of computation. We developed a simple formula for estimating the number of bits one needs to cut during each modulus switching step.

• **F-NTRU Scheme**

  - We introduced the first NTRU based FHE implementation that uses the flattening technique of GSW [21] by converting the NTRU ciphertexts into a matrix structure.

  - Similar to the YASHE construction in [2], we used a wide key distribution so that the security only relied on standard lattice reductions as in [20] (without DSPR assumption). Thus, our scheme is immune to the Subfield Lattice Attack by Albrecht, Bai and Ducas [22].

  - By employing flattening, we are able to multiply ciphertexts with only linear increase in magnitude of noise. Our scheme does not use any
expensive noise reduction techniques such as relinearization.

- Our construction does not require evaluation keys. In contrast, YASHE evaluation keys grow as $O(L^4)$ where $L$ represents the evaluation depth. This makes it impossible to use YASHE in deep evaluations.

- F-NTRU achieved a significantly smaller ciphertext size compared to YASHE for the same multiplicative depth.

- We introduced a variant F-NTRU’ of the scheme that reduces the ciphertext size and improves the homomorphic evaluation speed at the expense of the DSPR assumption.

- Using the noise asymmetry property of our scheme, we are able to provide small parameters which results in fast homomorphic multiplications in range of milliseconds even for large multiplicative levels, e.g. 34.3 msec for $L = 30$.

- Finally, via a simple polynomial to integer mapping our scheme becomes able to support homomorphic integer arithmetic. Featuring a very large message space, the integer version of F-NTRU was capable of supporting a wide range of applications where such arithmetic is required. We presented a noise analysis for this scheme.

- **Software Applications**

  - We homomorphically evaluated the full 128-bit AES circuit in a bit-sliced implementation to demonstrate the scalability of the introduced techniques. Our implementation is 5.8 times faster than the byte sliced implementation and 47 times faster than the SIMD implementation in [23].
- We presented a leveled homomorphic implementation of the Prince cipher. Specifically, we optimized the Prince cipher for shallow circuit implementation, and based on the depth characteristics, choose optimal but secure parameters for the library to evaluate Prince efficiently. With the chosen parameters, the batched implementation evaluates 1024 blocks in 57 minutes, with 3.3 seconds per block amortization.

- We constructed a PIR scheme from a batched leveled SWHE implementation based on the NTRU encryption scheme. Our scheme has excellent bandwidth performance compared to previous implementations (more than 1000 times smaller). The computational cost of our implementation is higher than previous proposals for databases containing a small number of bits in each row. However, this cost is amortized as the database rows become wider.

• Implementation of Gentry’s FHE in Hardware

- We presented the first ASIC realization of the full scheme (excluding key generation) to the best of our knowledge. Our hardware architecture supports encryption, decryption and recryption (bootstrapping) primitives for a 2048–dimension instantiation of the Gentry–Halevi scheme [9].

- We utilized a number of optimizations, including reformulation of the operations, use of spectral techniques and precomputations to speed up the arithmetic operations.

- We synthesized the proposed hardware using 90 nm TSMC library and achieved 10 times speedup for recryption operation.

- Another contribution of independent interest is the number theoretical transform based fast million-bit multiplier, which underlies the core of all
the primitives.

- **Implementation of DHS FHE in Hardware**

  - We presented an FPGA architecture to accelerate NTRU based FHE schemes. Our architecture may be considered as a proof-of-concept implementation of an external FHE accelerator that will speed up homomorphic evaluations taking place on a CPU.

  - The architecture manages to evaluate a full polynomial multiplication efficiently, for large degrees, i.e. $2^{14}$ and $2^{15}$, by utilizing a number theoretical transform based approach.

  - FPGA core can evaluate multiplication of $2^{14}$ degree polynomial 72 times faster than a CPU and 25.7 times faster than a GPU implementations. In the case of $2^{15}$ degree polynomials, it can evaluate the multiplications 102 and 36.5 times faster than a CPU and a GPU, respectively.

  - Furthermore, by including data transfer overhead, our hardware can evaluate a full 10 round AES circuit in under 440 ms per block. In the case of Prince, our hardware achieves amortized run time of 52 ms per block.
Chapter 2

Background

In this chapter, we introduce three main Fully Homomorphic Encryption (FHE) schemes. First, we give background information on the first FHE scheme. Than, we introduce NTRU based FHE schemes. Last, we give information on Learning With Errors (LWE) based FHE schemes.

2.1 Overview of Gentry’s Approach on FHE

In this section we give an outline on the first Fully Homomorphic Encryption Scheme (FHE) introduced by Gentry [24]. First, we give a high level description of the proposed Gentry’s FHE scheme. Later, we give a more detailed description on Gentry-Halevi’s FHE implementation. The implementation makes some changes on the original scheme to make it practical and run on a PC.

2.1.1 Gentry’s FHE Scheme

A high-level description of Gentry’s scheme is as follows. The scheme is based on identifying ideals $I$ in polynomial quotient rings $\mathbb{Z}[x]/(f(x))$ (with $\deg(f) = n)$
with Euclidean lattices $\mathbb{L}_I \subseteq \mathbb{R}^n$ by mapping each residue polynomial $r(x) = a_0 + \cdots + a_{n-1}x^{n-1}$ to its vector of coefficients $(a_0, \ldots, a_{n-1})$. Gentry calls these objects ideal lattices. Ideal lattices provide additive and multiplicative homomorphisms modulo a public key ideal. We obtain an encryption procedure $\text{Encrypt}$ such that $\text{Encrypt}(x_1) + \text{Encrypt}(x_2) = \text{Encrypt}(x_1 + x_2)$ and $\text{Encrypt}(x_1) \cdot \text{Encrypt}(x_2) = \text{Encrypt}(x_1 \cdot x_2)$. Therefore, any circuit $C$ with efficient description can be evaluated homomorphically. However, this somewhat fully homomorphic scheme (SWHE) is not perfect. Due to the noisy nature of the scheme, with each homomorphic gate evaluation the noise term in the partial result grows. After the evaluation of only a logarithmic depth circuit, the decryption fails to recover the correct result. To make the scheme work, Gentry uses a number of tricks. He introduces a re-encryption procedure called $\text{Recrypt}$ that takes a noisy ciphertext and returns a noise-reduced version. In a brilliant move, Gentry manages to obtain $\text{Recrypt}$ again from the SWHE scheme by simply homomorphically evaluating the decryption circuit using encrypted secret key bits on the noisy ciphertext. To make this work, the SWHE needs to be able to handle circuits that are deeper than its own decryption circuit before the level of noise becomes too large. SWHE schemes with this property are called bootstrappable.

2.1.2 The Gentry-Halevi FHE Scheme

Smart and Vercauteren specialized Gentry’s scheme to principal-ideal lattices, and forced the determinant of the lattice to be a prime number [25]. While this specialization improves the efficiency, it does not allow construction of the full scheme including bootstrapping and $\text{Recrypt}$ for practical key sizes [25]. Gentry and Halevi remove the primality restriction by introducing a special hermitian normal form for the bases. Further optimizations such as choosing sparse polynomials, batching poly-
nomial evaluations, customized resultant and inversion algorithm for \( f(x) = x^2l \pm 1 \) allowed the first software implementation of an FHE scheme. Here we give a high-level description of the primitives as follows. Let \([N]\) denote the round to nearest integer operation, \([N]_d = (N \mod d) - d\), and \([N] = \{0, 1, \ldots, N - 1\}\).

**Key Generation.** The key generation phase is rather involved but can be summarized with the following steps:

1. Set \( f_n(x) = x^n \pm 1 \). Choose a random \( n = 2^\theta\)-dimensional integer lattice represented by a randomly chosen polynomial \( v(x) \) where \( v_i \) are chosen from the set of \( t\)-bit signed integers.

2. Compute \( w(x) \) such that \( w(x)v(x) = d \ (\mod f_n(x)) \) where \( d \) represents a constant integer. This task may be achieved by using the polynomial version of the Extended Euclidean Algorithm \(^1\).

3. Compute \( r = w_0/w_1 \ (\mod d) \) and check if \( w_i = w_{i+1}r \ (\mod d) \) for all \( i = 1, \ldots, n - 2 \). If the inverse of \( w_1 \) does not exist restart the key generation procedure by picking a new random \( v(x) \) polynomial.

4. In order to facilitate re-encryption, randomly choose bit-vectors \( \sigma_i \) for \( i = 0, \ldots, S - 1 \) where each vector has Hamming weight one. Choose \( w' \) as any one of the odd coefficients of \( w(x) \). Randomly choose \( x_i \in \mathbb{Z}_d \) for \( i = 0, \ldots, s - 1 \) such that \( \sum_{j=0}^{S-1} \sum_{i=0}^{S-1} \sigma_i(j)x_j R^i \ (\mod d) = w' \). The parameter \( R \in \mathbb{Z} \) may be chosen as a power of 2.

5. Let \( l = \lceil 2\sqrt{S} \rceil \). For re-encryption, pick bits \( \eta_{i,j} \) for \( i \in [s] \), \( j \in [l] \) where \( \eta_{i,j} \) has Hamming weight 2 when viewed as an \( l\)-dimensional vector. Then encrypt each to obtain \( \beta_{i,j} = \text{Encrypt}(\eta_{i,j}) \).

\(^1\)Note that Section 4 of [25] presents a significantly more efficient technique for computing \( w(x) \).
6. The public key is $PK = (r, d, \{x_i : i \in [s]\}, \{\beta_{i,j} : i \in [s], j \in [l]\})$ and the secret key is $SK = (w', \sigma_0, \sigma_1, \ldots, \sigma_{S-1})$.

**Encryption.** To encrypt a bit $m \in \{0, 1\}$ first choose an $n$-th degree sparse random polynomial $u(x)$ with coefficients from $\{0, 1, -1\}$, for which probability of a coefficient being 0 is $\rho$. Using the PK parameters $(r, d)$ encryption is computed as follows: $Encrypt(m) = [m + 2 \sum_{i=0}^{n-1} u_ir^i]_d$. When multiple bits are to be encrypted, one may *batch* the computation yielding a significant speedup, e.g. $k$ encryptions may be computed with only $O(\sqrt{k})$ times more costly than a single bit encryption using simultaneous polynomial evaluation.

**Decryption.** We decrypt $c \in \mathbb{Z}_d$ using the secret key $SK = (w_i)$ simply by computing a modular multiplication as $Decrypt(c) = [cw_i]_d \pmod{2}$.

**Recryption.** The goal of recryption is to remove the noise buildup experienced during homomorphic circuit evaluations. We may only evaluate circuits to a constant (small) depth depending on the specific choice of parameters. To continue homomorphic evaluations we apply the recrypt procedure. Informally, recrypt works by homomorphically decrypting the ciphertext using encrypted secret key bits. A given ciphertext $c$ is recrypted by taking the following steps:

1. Compute $y_{j,i} = cx_jR_i \pmod{d}$ for $i = 0, \ldots, S - 1$ and $j = 0, \ldots, s - 1$.

2. Compute $z_{j,i} = y_{j,i}/d$ as the $p = \lceil \log_2(s + 1) \rceil$ bit approximation to the right of the binary point.

3. For $j \in [s]$ compute the quotients $q_j = \sum_{a \in [l]} \beta_{j,a} \left( \sum_{b \in [l]} \beta_{j,b}z_{j,i(a,b)} \right) \pmod{d}$ where the index function is defined as $i(a, b) = al - \left(\frac{a+1}{2}\right) + (b - a)$. Note that the $\beta_{j,a}z_{j,i}$ products are realized as conditional additions in $\mathbb{Z}_d$ (since $z_{j,i}$ are
bits in cleartext) and only the product of the result of the inner summation with $\beta_{j,i}$ requires multiplication in $\mathbb{Z}_d$.

4. Finally, the re-encryption of $c \in \mathbb{Z}_d$ is achieved by homomorphically evaluating the decryption circuit on $c$ in encrypted form. After a number of optimizations, the decryption operation $\text{Decrypt}_SK(c)$ is expressed in the following form:

$$\left[ \sum_{j \in [s]} \sum_{i \in [S]} \sigma_j(i)z_{j,i} \right] + \sum_{j \in [s], i \in [l]} \sigma_j(i)(y_{j,i}) \pmod{2}.$$  

Note that the inputs $d$ and $y_{j,i}$ are in cleartext form while the secret key $\sigma_j$ are in encrypted form during the evaluation of the decryption circuit. The first summation is homomorphically computed on the individual bits (in encrypted form) via grade school addition of $s$ fixed point numbers expressed using $p$ bits. Therefore, in the computation of the actual recrypt operation, $\sigma_j(i)$ are replaced with their recoded and encrypted form, i.e. $\beta_{j,i}$ and inner summation of the first term with $q_j$. During homomorphic evaluation, the (mod 2) additions and multiplications turn into additions and multiplications in $\mathbb{Z}_d$, respectively. The depth of the circuit evaluating the carry output may be shown to be bounded by $O(s^2)$. Hence, we end up computing in the order of $O(s^2)$ multiplications in $\mathbb{Z}_d$ to figure out the carry bit and reflect it to the LSB in encrypted form using a $\mathbb{Z}_d$ addition. The second sum multiplies bits by ciphertexts in $\mathbb{Z}_d$.

2.2 NTRU Based FHE Schemes

In this section we introduce FHE schemes that are based on the NTRU encryption system [19]. First, we introduce Stehlé and Steinfeld variant which is a modified
version of NTRU. Later, we summarize López-Alt, Tromer and Vaikuntanathan’s (LTV) FHE and YASHE schemes which are based on Stehlé and Steinfeld’s NTRU variant.

### 2.2.1 Stehlé and Steinfeld’s NTRU Variant

In [20] Stehlé and Steinfeld introduced a modification to the NTRU [19] scheme to make the scheme secure under the assumed quantum hardness of the standard worst-case lattice problem. The scheme fixes the ring to $R = \mathbb{Z}[x]/(x^n + 1)$ where $n$ is a power of 2 and chooses $q \leq \text{Poly}(n)$. Also, chooses a message space $\mathbb{Z}_p$. Using a discrete Gaussian distribution random samples $f', g$ are chosen where the secret key becomes $f = pf' + 1$. Set public key $h = pf^{-1}g$. A message $\mu$ is encrypted by computing $c = hs + pe + \mu \mod q$ where $s, e$ are Gaussian distribution samples. To decrypt we simply compute $\mu = c \cdot f \mod p$. Their modification is the addition of error term $e$ which makes the scheme derive CPA security under the hardness of a variant of R-LWE.

### 2.2.2 LTV FHE Scheme

López-Alt, Tromer and Vaikuntanathan introduce a FHE scheme [16] by using Stehlé and Steinfeld’s NTRU variant. Although the scheme can support multi-key evaluations, for simplicity, we summarize the single key setting. The scheme sets its parameters by using the security parameter $\eta$ as:

- an integer $n = n(\eta)$,
- a prime number $q = q(\eta)$,
- a degree-$n$ polynomial $\phi(x) = \phi_{\eta}(x) = x^n + 1$,
• a $B(\eta)$-bounded error distribution $\chi = \chi(\eta)$ over the ring $R = \mathbb{Z}[x]/\langle \phi(x) \rangle$.

The primitives of the public key encryption scheme defined as follows:

• **KeyGen**: We choose a decreasing sequence of primes $q_0 > q_1 > \cdots > q_d$, a polynomial $\phi(x) = x^n + 1$ and set $\chi$ as a truncated discrete Gaussian distribution that is $B$-bounded. For each $i$, we sample $u(i)$ and $g(i)$ from distribution $\chi$, set $f(i) = 2u(i) + 1$ and $h(i) = 2g(i) (f(i))^{-1}$ in ring $R_{q_i} = \mathbb{Z}_{q_i}[x]/\langle \phi(x) \rangle$. (If $f(i)$ is not invertible in this ring, re-sample.) We then sample, for $i = 0, \ldots, d$ and for $\tau = 0, \ldots, \lfloor \log q_i \rfloor$, $s^{(i)}_\tau$ and $e^{(i)}_\tau$ from $\chi$ and publish evaluation keys $\left\{ s^{(i)}_\tau \right\}_\tau$ where $s^{(i)}_\tau = h(i)s^{(i)} + 2e^{(i)} + 2\tau (f(i-1))^2$ in $R_{q_{i-1}}$. Then set; secret key $sk = (f(i))$ and public keys $pk = (h(i), s^{(i)}_\tau)$.

• **Encrypt**: To encrypt a bit $b \in \{0, 1\}$ with a public key $(h(0), q_0)$, Encrypt first generates random samples $s$ and $e$ from $\chi$ and sets $c^{(0)} = h(0)s + 2e + b$, a polynomial in $R_{q_0}$.

• **Decrypt**: To decrypt the ciphertext $c$ with the corresponding private key $f(i)$, Decrypt multiplies the ciphertext and the private key in $R_{q_i}$ then compute the message by modulo two: $m = c^{(i)}f(i) \pmod{2}$

• **Eval**: Arithmetic operations are performed directly on ciphertexts as follows: Suppose $c^{(0)}_1 = Encrypt(b_1)$ and $c^{(0)}_2 = Encrypt(b_2)$. Then XOR is effected by simply adding and AND is effected by simply multiplying the ciphertexts:

$$b_1 \oplus b_2 = Decrypt(c^{(0)}_1 + c^{(0)}_2) \quad \text{and} \quad b_1 \cdot b_2 = Decrypt(c^{(0)}_1 \times c^{(0)}_2) .$$

Polynomial multiplication incurs a much greater growth in the noise, so each
multiplication step is followed by a modulus switching. First, we compute
\[
\tilde{c}^{(0)}(x) = c_1^{(0)} \cdot c_2^{(0)} \pmod{\phi(x)}
\]
and then perform Relinearization, as described below, to obtain \( \tilde{c}^{(1)}(x) \) followed by modulus switching \( \text{Encrypt}(b_1 \cdot b_2) = \left\lfloor \frac{q_1}{q_0} \tilde{c}^{(1)}(x) \right\rfloor_2 \) where the subscript 2 on the rounding operator indicates that we round up or down in order to make all coefficients equal modulo 2. The same process hold for evaluating with \( i^{th} \) level ciphertexts, e.g. computing \( \tilde{c}^{(i)}(x) \) from \( c^{(i-1)}_1 \) and \( c^{(i-1)}_2 \).

- **Modulus Switch**: The modulus switching technique is used to transform the ciphertext in modulo \( q \) into a different ciphertext with a smaller modulo \( q' \) while preserving the parity. This way they reduce the noise while preserving the correctness of the ciphertext for a key. They simply compute it by scaling the coefficients by \( q'/q \) and round the number number to match the parity.

\[
\frac{q'}{q} \tilde{c} = c \pmod{2}
\]

A more detailed proof can be found in [26].

- **Relinearize**: We will show the general process that computing \( \tilde{c}^{(i)}(x) \) from \( \tilde{c}^{(i-1)}(x) \). We expand \( \tilde{c}^{(i-1)}(x) \) as an integer linear combination of 1-bounded polynomials \( \tilde{c}^{(i-1)}(x) = \sum_\tau 2^{\tau} \tilde{c}^{(i-1)}(x) \) where \( \tilde{c}^{(i-1)}(x) \) takes its coefficients from \( \{0, 1\} \). We then define \( \tilde{c}^{(i)}(x) = \sum_\tau \zeta^{(i)}_\tau(x) \tilde{c}^{(i-1)}_\tau(x) \) in \( R_q \).
To see why relinearization works, observe that simple substitution gives us

\[
\tilde{c}^{(i)}(x) = h^{(i)}(x) \left[ \sum_{\tau=0}^{\left\lfloor \log q_i \right\rfloor} s^{(i)}_\tau(x) \tilde{c}^{(i-1)}_\tau(x) \right] + 2 \left[ \sum_{\tau=0}^{\left\lfloor \log q_i \right\rfloor} c^{(i)}_\tau(x) \tilde{c}^{(i-1)}_\tau(x) \right] \\
+ \left[ f^{(i-1)} \right]^2 \sum_{\tau=0}^{\left\lfloor \log q_i \right\rfloor} 2^\tau \tilde{c}^{(i-1)}_\tau(x) \\
= h^{(i)}(x)S(x) + 2E(x) + \left[ f^{(i-1)} \right]^2 \tilde{c}^{(i-1)}(x) \\
= h^{(i)}(x)S(x) + 2E(x) + \left[ f^{(i-1)} c^{(i-1)}_1(x) \right] \left[ f^{(i-1)} c^{(i-1)}_2(x) \right] \\
= h^{(i)}(x)S(x) + 2E'(x) + m_1m_2
\]

modulo \( q_{i-1} \) for some pseudorandom polynomials \( S(x) \) and \( E'(x) \). This ensures that the output of each gate takes the form of a valid encryption of the product \( m_1m_2 \) of plaintexts with reduced noise.

### 2.2.3 YASHE Scheme

The BLLN scheme was introduced by Bos et al. in [2]. BLLN is based of the scheme proposed by Stehlé and Steinfeld [20]. The authors create a scheme called YASHE by using the construction of [20] and modify it by applying the tensor product technique of [17] to curb the noise growth in multiplications. With this it becomes possible to use a high level of noise in encryptions with which the schemes becomes DSPR hard as in [20]. However, this brings large evaluation keys into the scheme and makes it difficult to use in practice: the evaluation key consists of \( \ell^3 = \mathcal{O}(\left(\log (q)\right)^3) \) ciphertexts. To mitigate this problem, in the same reference the authors introduced another scheme called YASHE'. They discard the tensor product and decrease the size of the evaluation keys, i.e. \( \ell \) ciphertexts. However, they have to reduce the
noise levels on fresh ciphertexts which brings back the DSPR security assumption. In the following, we first give the Basic scheme and later explain YASHE and YASHE’ constructions using the Basic scheme.

Basic. We set the ring as \( R = \mathbb{Z}[X]/\langle x^n + 1 \rangle \), \( t \) as the plaintext modulus, \( \chi_{\text{err}} \) and \( \chi_{\text{key}} \) as the Gaussian distribution for sampling. The scheme has the following primitive functions:

- **ParamsGen.** For security parameter \( \lambda \) we create \( n = n(\lambda), q = q(\lambda), \chi_{\text{key}} = \chi_{\text{key}}(\lambda) \) and \( \chi_{\text{err}} = \chi_{\text{err}}(\lambda) \).

- **KeyGen.** We sample \( f', g \in \chi_{\text{key}} \). Set secret key \( f = tf' + 1 \) and public key \( h = tgf^{-1} \).

- **Encrypt.** To encrypt message \( \mu \), we sample \( s, e \in \chi_{\text{err}} \) and compute \( c = \left\lfloor \frac{q}{t} \right\rfloor \mu + e + hs \).

- **Decrypt.** To decrypt a message simply compute \( \mu = \left\lceil \frac{t}{q} fc \right\rceil \) where \( \left\lceil \cdot \right\rceil \) is rounding to the nearest.

There are 3 notations that need explanation before we summarize the YASHE scheme. The authors use \( \otimes \) for the tensor product. The notation \( P_{w,q} \) is used to convert a number to an array with powers of \( w \), i.e. \( P_{w,q}(x) \rightarrow (x, xw, xw^2, \ldots, xw^{\ell_{w,q}-1}) \) for \( \ell_{w,q} = (\log w q + 2) \). Last notation is \( D_{w,q} \) which decomposes a value into its word sizes, i.e. \( D_{w,q}(x) \rightarrow (x_0, x_1, \ldots, x_{\ell_{w,q}-1}) \) where \( x = \sum_{i=0}^{\ell_{w,q}-1} x_i w^i \).

YASHE. The Basic scheme explained above is used to construct a leveled FHE scheme. The primitive operations of the scheme is as follows:

- **ParamsGen.** The parameters are selected the same way as in Basic scheme.
• **KeyGen.** The public and secret key pairs are selected using the `Basic.KeyGen` routine, i.e. \( h, f \leftarrow \text{Basic.KeyGen} \). Sample \( e, s \in \chi_{\text{err}}^{\ell_{w,q}} \). Compute the evaluation keys

\[
\zeta = \left[ f^{-1}P_{w,q}(D_{w,q}(f) \otimes D_{w,q}(f)) + e + hs \right] \in R^{\ell_{w,q}}.
\]

• **Encrypt.** Encrypt message \( \mu \) as in `Basic` scheme.

• **Decrypt.** Decrypt message as in `Basic` scheme.

• **KeySwitch.** Compute \( \langle D_{w,q}(c), \zeta \rangle \) where \( c \) is a ciphertext.

• **Addition.** Addition of two ciphertexts \( c_1 \) and \( c_2 \) is \( c = c_1 + c_2 \).

• **Multiplication.** Multiplication of two ciphertexts is computed as

\[
c = \left[ \left\lfloor \frac{t}{q} P_{w,q}(c_1) \otimes P_{w,q}(c_2) \right\rfloor \right]
\]

and later apply **KeySwitch** to \( c \) and output.

**YASHE’.** The primitive operations are as follows:

• **ParamsGen.** The parameters are selected the same way as in `Basic` scheme.

• **KeyGen.** The public and secret key pairs are selected using the `Basic.KeyGen` routine, i.e. \( h, f \leftarrow \text{Basic.KeyGen} \). Sample \( e, s \in \chi_{\text{err}}^{\ell_{w,q}} \). Compute the evaluation keys

\[
\zeta = \left[ f^{-1}P_{w,q}(D_{w,q}(f) \otimes D_{w,q}(f)) + e + hs \right] \in R^{\ell_{w,q}}.
\]

• **Encrypt.** Encrypt message \( \mu \) as in `Basic` scheme.
• **Decrypt.** Decrypt message as in Basic scheme.

• **KeySwitch.** Compute \( D_{w,q}(c), \zeta \) where \( c \) is a ciphertext.

• **Addition.** Addition of two ciphertexts \( c_1 \) and \( c_2 \) is \( c = c_1 + c_2 \).

• **Multiplication.** Multiplication of two ciphertexts is computed as

\[
    c = \left[ \begin{array}{c} \frac{t}{q} - c_1 c_2 \\
    \end{array} \right]
\]

and later apply KeySwitch to \( c \) and output.

### 2.3 LWE Based Fully Homomorphic Encryption Schemes

The learning with errors (LWE) problems first introduced by Regev in [27]. However, it is after the Gentry’s first FHE construction that lead many FHE schemes that is based on LWE. There are many LWE based FHE constructions which are usually progress of their predecessors. Here in this section we first introduce a LWE scheme that is constructed by Brakerski, Gentry and Vaikuntanathan [26]. Later, we introduce another scheme (GSW) that is based on LWE which eliminates the necessities of relinearization and evaluation keys.

#### 2.3.1 BGV Scheme

Brakerski, Gentry and Vaikuntanathan proposed this new FHE scheme based on the LWE problem. Their scheme can evaluate a circuit homomorphically with depth \( L \) without the bootstrapping. First we give a general LWE scheme setup and later we give details on the BGV scheme setup.
• **E.Setup.** For depth $L$ and security parameter $\lambda$, set $d = d(\lambda, L)$ and $n = n(\lambda, L)$ for ring $R = \mathbb{Z}[x]/(x^d + 1)$ and set Gaussian distribution $\chi = \chi(\lambda, L)$. Also, choose a decreasing ladder of moduli $q_i$ and set $N = (2n + 1) \log q$

• **E.KeyGen.** For each level $L$, draw $s' \leftarrow \chi^n \ sk = s_j \leftarrow (1, s'[1], \ldots, s'[n]) \in R^{n+1}$.

Generate matrix for each level $A'_j \leftarrow R^{N \times n}$ and a vector $e_j \leftarrow \chi^N$ and set $b_j \leftarrow A'_j s'_j + 2e_j$. Set $pk = A_j$ to be the $(n + 1)$-column matrix consisting of $b_j$ followed by the $n$ columns of $A'_j$.

• **E.Encrypt.** To encrypt a message $m$ for level $i$, set $s \leftarrow (m, 0, \ldots, 0) R^{n+1}$, sample $r \leftarrow R_N^2$ and output ciphertext $c \leftarrow m + A^T_i r \in R^{n+1}$.

• **E.Decrypt.** To decrypt a ciphertext at level $i$, compute $m \leftarrow \langle c, s_i \rangle_2$.

Using the setup above, the proposed FHE scheme is constructed as

• **FHE.Setup**($1^\lambda, 1^L, b$): Takes as input the security parameter, a number of levels $L$, and a bit $b$. Use the bit $b \in \{0, 1\}$ to determine whether we are setting parameters for a LWE-based scheme. Let $\mu = \mu(\lambda, L, b) = \theta(\log \lambda + \log L)$. For $j = L$ to 0, run $params_j \leftarrow \text{E.Setup}(1^\lambda, 1^{(j+1)^\mu}, b)$ to obtain a ladder of decreasing moduli from $q_L((L + 1)\mu)$ down to $q_0(\mu)$. For $j = L - 1$ to 0, replace the value of $d_j$ in $params_j$ with $d = d_L$ and the distribution $\chi_j$ with $\chi = \chi_L$. (That is, the ring dimension and noise distribution do not depend on the circuit level, but the vector dimension $n_j$ still might.)

• **FHE.KeyGen**(\{params\}): For $j = L$ down to 0, do the following:

  - Run $s_j \leftarrow \text{E.SecretKeyGen}(params_j)$ and $A_j \leftarrow \text{E.PublicKeyGen}(params_j, s_j)$. 

20
- Set $s'_j \leftarrow s_j \otimes s_j \in R_{q_j}^{(n_j+1)/2}$. That is, $s'_j$ is a tensoring of $s_j$ with itself whose coefficients are each the product of two coefficients of $s_j$ in $R_{q_j}$.

- Set $s''_j \leftarrow \text{BitDecomp}(s'_j, q_j)$.

- Run $\tau_{s''_j \rightarrow s_j} \leftarrow \text{SwitchKeyGen}(s''_j, s_{j-1})$. (Omit this step when $j = L$.)

The secret key $sk$ consist of the $s_j$’s and the public key $pk$ consist of the $A_j$’s and $\tau_{s''_j \rightarrow s_j}$’s.

- **FHE.Enc(params, pk, m):** Take a message in $R_2$. Run $\text{E.Enc}(A_L, m)$.

- **FHE.Dec(params, sk, c):** Suppose the ciphertext is under key $s_j$. Run $\text{E.Dec}(s_j, c)$.

- **FHE.Add(pk, c_1, c_2):** Takes two ciphertexts encrypted under the same $s_j$.
  Set $c_3 \leftarrow c_1 + c_2 \pmod{q_j}$. Interpret $c_3$ as a ciphertext under $s'_j$ and output:
  \[c_4 \leftarrow \text{FHE.Refresh}(c_3, \tau_{s''_j \rightarrow s_{j-1}}, q_j, q_j-1)\]

- **FHE.Mult(pk, c_1, c_2):** Takes two ciphertexts encrypted under the same $s_j$.
  First, multiply: the new ciphertext, under the secret key $s'_j = s_j \otimes s_j$, is the coefficient vector $c_3$ of the linear equation $L_{c_1, c_2}^{\text{long}}(x \otimes x)$. Then, output:
  \[c_4 \leftarrow \text{FHE.Refresh}(c_3, \tau_{s''_j \rightarrow s_{j-1}}, q_j, q_j-1)\]

- **FHE.Refresh(c_3, \tau_{s''_j \rightarrow s_{j-1}}, q_j, q_j-1):** Takes a ciphertext encrypted under $s'_j$, the auxiliary information $\tau_{s''_j \rightarrow s_{j-1}}$ to facilitate key switching, and the current and next moduli $q_j$ and $q_{j-1}$. Do the following:
  - Expand: Set $c_1 \leftarrow \text{Powersof}(c, q_j)$. 

21
Switch Moduli: Set $c_2 \leftarrow \text{Scale}(c_1, q_j, q_{j-1}, 2)$, a ciphertext under the key $s''_j$ for modulus $q_{j-1}$.

Switch Keys: Output $c_3 \leftarrow \text{SwitchKey}(\tau_{s''_j \rightarrow s_{j-1}}, c_2, q_{j-1})$, a ciphertext under the key $s_{j-1}$ for modulus $q_{j-1}$.

2.3.2 GSW Scheme

Gentry, Sahai and Waters [21] proposed this new scheme based on the hardness of the approximate eigenvector problem. The construction consists of simple matrix addition and multiplication operations to perform homomorphic addition and homomorphic multiplication. The advantage of the scheme is that it eliminates the need of relinearization, storage of evaluation keys and even modulus switching. As many other homomorphic schemes the security is based on the LWE problem.

We explain the scheme in four primitive functions in below. Before the explanations we should note some of the preliminaries that is used in the primitive functions. A vector $\vec{a} = (a_0, \ldots, a_{k-1})$ is split into its bits using the function $\text{BitDecomp}(\vec{a}) = (a_{0,0}, \ldots, a_{0,\ell-1}, \ldots, a_{k-1,0}, \ldots, a_{k-1,\ell-1})$. We are able to reconstruct the elements from the bit representations using the inverse of the $\text{BitDecomp}$ as $\text{BitDecomp}^{-1}(\vec{a}) = (\sum 2^j a_{0,j}, \ldots, \sum 2^j a_{k-1,j})$. The most important function that the scheme uses is $\text{flattening}$. It keeps the ciphertexts bounded so that the noise increase after multiplicative operations are limited. We simply evaluate it as $\text{Flatten}(\vec{a}) = \text{BitDecomp}(\text{BitDecomp}^{-1}(\vec{a}))$. The last function is $\text{Powersof2}(\vec{a}) = (a_0, 2a_0, \ldots, 2^{\ell-1}a_0, \ldots, a_k, 2a_k, \ldots, 2^{\ell-1}a_k)$ which multiplies the vector elements with powers of two. Using these functions, the encryption scheme is defined with the following primitives:

- **Setup.** We select $\lambda$ as the security parameter and $L$ as multiplicative depth.
Then compute lattice dimension \( n = n(\lambda, L) \), error distribution \( \chi = \chi(\lambda, L) \), parameter \( m = m(\lambda, L) = O(n \log q) \). Also, set \( \ell = \lceil \log q \rceil + 1 \) and \( N = (n + 1)\ell \).

- **KeyGen.** We sample \( \tilde{t} \leftarrow \mathbb{Z}_q^n \) and compute \( \tilde{s} \leftarrow (1, -t_1, -t_2, \ldots, -t_n) \). Then, set the secret key \( \tilde{v} = \text{Powersof2}(\tilde{s}) \). The public key matrix is computed by first generating uniform matrix \( B \leftarrow \mathbb{Z}_q^{m \times n} \) and error vector \( \tilde{e} \leftarrow \chi^m \). We set \((n + 1)\) column matrix \( A \) having \( \tilde{b} = B \cdot \tilde{t} + \tilde{e} \) as the first column and \( \tilde{b} \) in rest of the columns as the public key.

- **Encrypt.** A message \( \mu \) is encrypted by simply computing \( C = \text{Flatten}(\mu I_N + \text{BitDecomp}(R \cdot A)) \in \mathbb{Z}_q^{N \times N} \). In the equation \( R \) is selected as a uniform matrix \( R \in \{0, 1\}^{N \times m} \).

- **Decrypt.** Select a row of the matrix, i.e. \( C_i \) as the \( i \)-th row of matrix \( C \). Compute \( x_i \leftarrow \langle C_i, \tilde{v} \rangle \) and the message as \( \mu' = \lfloor x_i / v_i \rfloor \).

The beauty of the GSW scheme is that straightforward addition and multiplication of ciphertext matrices suffice to compute homomorphic additions and multiplications. First, lets observe that the construction holds the property \( C \cdot \tilde{v} = \mu \cdot \tilde{v} + \tilde{e} \), since secret key \( \tilde{v} \) is the approximate eigenvector related to the ciphertext. Therefore, adding ciphertext matrices has the effect of adding the corresponding eigenvalues (messages): \( (C_1 + C_2) \cdot \mathbf{v} = (\mu_1 + \mu_2) \cdot \mathbf{v} + (\mathbf{e}_1 + \mathbf{e}_2) \). Similarly the matrix product of the ciphertexts (in any order) multiplies the eigenvalues: \( C_1 \cdot C_2 \cdot \mathbf{v} = C_1 (\mu_2 \cdot \mathbf{v} + \mathbf{e}_2) = \mu_2 (\mu_1 \cdot \mathbf{v} + \mathbf{e}_1) + C_1 \cdot \mathbf{e}_2 = \mu_1 \mu_2 \cdot \mathbf{v} + \mathbf{e} \).
Chapter 3

Proposed New FHE Schemes

In this chapter we introduce two new FHE schemes that we constructed. In Section 3.1, we introduce a FHE construction that we based on LTV scheme by applying various optimizations. Later, in Section 3.2, we use the noise management technique from GSW construction [21] on top of the NTRU encryption scheme to create a new FHE scheme that is resilient against the subfield attacks on NTRU.

3.1 DHS FHE Scheme

In this section, we describe our proposed FHE scheme which is based on the LTV scheme. We start by describing the optimizations that we implemented on top of the LTV scheme. Later, we state the operations of our homomorphic scheme, present an analysis of the noise growth and show a noise coping mechanism. Last, we evaluate concrete parameters to make our FHE scheme secure.
3.1.1 Optimizations

3.1.1.1 Batching

*Batching* has become an indispensable tool for boosting the efficiency of homomorphic evaluations [28]. In a nutshell, batching allows us to evaluate a circuit, e.g. AES, on multiple independent data inputs simultaneously by embedding them into the same ciphertext. With batching multiple message bits belonging to parallel data streams are packed into a single ciphertext all undergoing the same operation similarly as in the single instruction multiple data (SIMD) computing paradigm.

The LTV scheme we use here permits the encryption of binary polynomials as messages. However a simple encoding where each message polynomial coefficient holds a message bit is not very useful when it comes to the evaluation of multiplication operations. When we multiply two ciphertexts (evaluate an **AND**) the resulting ciphertext will contain the product of the two message polynomials. However, we will not be able to extract the parallel product of message bits packed in the original ciphertext operands. The cross product terms will overwrite the desired results. Therefore, a different type of encoding of the message polynomial is required so that **AND** and **XOR** operations can be performed on batched bits fully in parallel. We adopted the technique presented by Smart and Vercauteren [28]. Their technique is based on an elegant application of the Chinese Remainder Theorem on a cyclotomic polynomial $\Phi_m(x)$ where $\deg(\Phi_m(x)) = \phi(m)$. An important property of cyclotomic polynomials with $m$ odd is that it factorizes into same degree factors over $\mathbb{F}_2$. In other words, $\Phi_m$ has the form

$$
\Phi_m(x) = \prod_{i \in [\ell]} F_i(x),
$$

where $\ell$ is the number of factors irreducible in $\mathbb{F}_2$ and $\deg(F_i(x)) = d$ and $d = N/\ell$. 

25
The parameter $d$ is the smallest value satisfying $m | (2^d - 1)$. Each factor $F_i$ defines a message slot in which we can embed message bits. Actually we can embed elements of $\mathbb{F}_2[x]/\langle F_i \rangle$ and perform batched arithmetic in the same domain. However, in this paper we will only embed elements of $\mathbb{F}_2$ in the message slots. To pack a vector of $\ell$ message bits $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_{\ell-1})$ into a message polynomial $a(x)$ we compute the CRT inverse on the vector $\mathbf{a}$

$$a(x) = \text{CRT}^{-1}(\mathbf{a}) = a_0 M_0 + a_1 M_1 + \cdots + a_{\ell-1} M_{\ell-1} \pmod{\Phi_m}.$$

The values $M_i$ are precomputed values that are shown as:

$$M_i = \frac{\Phi_m}{F_i(x)} \left( \left( \frac{\Phi_m}{F_i(x)} \right)^{-1} \pmod{F_i(x)} \right) \pmod{\Phi_m}.$$

The batched message can be extracted easily by performing modular reduction on the polynomial, e.g. $a_i = a(x) \pmod{F_i(x)}$. Due to the Chinese Remainder Theorem multiplication and addition of the message polynomials carry through to the residues: $a_i \cdot b_i = a(x) \cdot b(x) \pmod{F_i(x)}$ and $a_i + b_i = a(x) + b(x) \pmod{F_i(x)}$.

### 3.1.1.2 Reducing the Public Key Size

To cope with the growth of noise, following Brakerski et al [26] we introduce a series of decreasing moduli $q_0 > q_1 > \ldots > q_{t-1}$; one modulus per circuit level. Modulus switching is a powerful technique that exponentially reduces the growth of noise during computations. Here we introduce a mild optimization that allows us to reduce the public key size drastically. We require that $q_i = p^{t-i}$ for $i = 0, \ldots, t-1$ where $p \in \mathbb{Z}$ is a prime integer. Therefore, $\mathbb{Z}_{q_i} \supset \mathbb{Z}_{q_j}$ for any $i < j$. We also require the secret key $f \in \mathbb{Z}_{q_0}/\langle \Phi(x) \rangle$ to be invertible in all rings $\mathbb{Z}_{q_i}$. Luckily, the following lemma from [29] tells us that we only need to worry about invertibility in $\mathbb{Z}_p = \mathbb{F}_p$.  

26
Note that the lemma is given for $R'_p = \mathbb{Z}_p[x]/\langle x^n - 1 \rangle$ however the proof given in [29] is generic and also applies to the $R_p$ setting.

**Lemma 1** (Lemma 3.3 in [29]). Let $p$ be a prime, and let $f$ be a polynomial. If $f$ is a unit in $R'_p$, (or $R_p$) then $f$ is a unit in $R'_{p^k}$ (or $R_{p^k}$) for every $k \geq 1$.

Under this condition the inverse $f^{-1} \in \mathbb{Z}_{q_i}/\langle \Phi(x) \rangle$ which is contained in the public key $h$ will persist through the levels of computations, while implicitly being reduced to each new subring $\mathbb{Z}_{q_i+1}/\langle \Phi(x) \rangle$ when $q_{i+1}$ is used in the computation. More precisely, let $f^{(i)}(x) = f(x)^{-1} \pmod{q_i}$. Then we claim $f^{(i)} \pmod{q_{i+1}} = f^{(i+1)}$ for $i = 0, \ldots, t-1$. To see why this works, note that by definition it holds that $f(x)f^{(t-1)}(x) = 1 \pmod{p}$ which allows us to write $f(x)f^{(t-1)}(x) = 1 - pu(x)$ for some $u(x)$ and form the geometric (Maclaurin series) expansion of $f(x)^{-1}$ w.r.t. modulus $q_{t-k} = p^{k-1}$ for any $k = 1, \ldots, t$ as follows

$$f(x)^{-1} = f^{(t-1)}(x)(1 - pu(x))^{-1}$$

$$= f^{(t-1)}(x)(1 + pu(x) + p^2u(x)^2 + \cdots + p^{k-2}u(x)^{k-2}) \pmod{p^{k-1}}.$$

Then it holds that $f^{(i)} \pmod{q_{i+1}} = f^{(i+1)}$ for $i = 0, \ldots, t-1$. This means that to switch to a new level (and modulus) during homomorphic evaluation the public key we simply compute via modular reduction. The secret key $f$ remains the same for all levels. Therefore, key switching is no longer needed. Also we no longer need to store a secret-key/public-key for each level of the circuit. With this approach we can still take advantage of modulus switching without having to pay for storage or key switching.

In the original scheme, the public key size is quadratically dependent on the number of the levels the instantiation can support. Also the number of evaluation keys needed in a level is dependent to the bit size of the modulus at that level, i.e.
log \(q_i\). Having a polynomial size \(n \log q_i\) at each level, the public key size can be written as

\[
|\text{PK}| = \sum_{i=0}^{t-2} n(\log q_i)^2.
\]

In our modified scheme we only need the key for the first level \(q_0 = p^t\) which is progressively reduced as the evaluation proceeds through the levels, and therefore

\[
|\text{PK}'| = n(\log q_0)^2.
\]

To understand the impact of this restriction on key generation and on the size of the key space we invoke an earlier result by Silverman [29]. In this study, Silverman analyzed the probability of a randomly chosen \(f \in R'_q = \mathbb{Z}_q[x]/\langle x^n - 1 \rangle\) to be invertible in \(R'_q\).

**Theorem 1** ([29]). Let \(q = p^k\) be a power of a prime \(p\), and let \(n \geq 2\) be an integer with \(\gcd(q, n) = 1\). Define \(w \geq 1\) to be the smallest positive integer such that \(p^w = 1 \mod n\) and for each integer \(d|w\), let

\[
\nu_d = \frac{1}{d} \sum_{e|d} \mu \left( \frac{d}{e} \right) \gcd(n, p^e - 1)
\]

then

\[
\frac{|R'_{q^*}|}{|R'_q|} = \prod_{d|w} \left( 1 - \frac{1}{p^d} \right)^{\nu_d}
\]

Silverman [29] noted that the large class of noninvertible polynomials \(f \in R'_q\) such that \(f(1) = 0\) can be avoided by “intelligently choosing” \(f\). He further restricts the selection \(f \in R'_q\) such that \(f(1) = 1\) and derives an approximation on the probability
of picking an invertible $f$ which simplifies for large and prime $n$ as follows

$$\frac{|R'_q(1)|}{|R'_q(1)|} \approx 1 - \frac{n - 1}{wp^w}.$$  

Here $R'_q(1) = \{ f \in R'_q \mid f(1) = 1 \}$ and $R'_{pq}(1) = \{ f \in R'_{pq} \mid f(1) = 1 \}$ and $\exists g \in R'_q \mid g \cdot f = 1 \}$. Note that the failure probability may be made negligibly small by picking appropriate $p$ and $n$ values.

In this paper we are working in a much simpler setting, i.e. $R_q = \mathbb{Z}_q[z]/\langle \Phi_m(x) \rangle$. The uniform factorization of the cyclotomic polynomial $\Phi(x)$ allows us to adapt Silverman’s analysis [29] and obtain a much simpler result. Assuming $\text{gcd}(n, p) = 1$, the cyclotomic polynomial factors into equal degree irreducible polynomials $\Phi_m(x) = \prod_{i=1}^\ell F_i(x)$ over $\mathbb{Z}_p$, where $\deg(F_i(x)) = w$, $\ell = \phi(m)/w$ and $w \geq 1 \in \mathbb{Z}$ is the smallest integer satisfying $p^w = 1 \pmod{\phi(m)}$. Therefore

$$\mathbb{F}_p[x]/\langle \Phi_m(x) \rangle \cong \mathbb{F}_p[x]/\langle F_1(x) \rangle \times \cdots \times \mathbb{F}_p[x]/\langle F_\ell(x) \rangle \cong (\mathbb{F}_{p^w})^\ell$$

and for $R_q$ we have $\nu_d = \ell$. With this simplification the probability of randomly picking an invertible $f \in R_q$ given in Theorem 1 simplifies to

$$\frac{|R'_q|}{|R_q|} = \left( 1 - \frac{1}{p^w} \right)^{\ell}.$$  

When $p$ is large $|R'_q|/|R_q| \approx 1 - \ell p^{-w}$.

With the new restriction imposed by the selection of the moduli we introduce a modified KeyGen procedure as follows.

**Modified KeyGen.** We use the chosen decreasing moduli $q_0 > q_1 > \cdots > q_{t-1}$ where $q_i = p^{t-i}$ for $i = 0, \ldots, t - 1$. We further set the $m^{th}$ cyclotomic polynomial $\Phi_m(x)$ as our polynomial modulus and set $\chi$ as a truncated discrete Gaussian distri-
bution that is $B$-bounded. We sample $u$ and $g$ from distribution $\chi$, set $f = 2u + 1$ and $h = 2gf^{-1}$ in ring $R_{q_0} = \mathbb{Z}_{q_0}[x]/\langle \Phi_m(x) \rangle$. We then sample, for $\tau = 0, \ldots, \lfloor \log q_0 \rfloor$, $s_\tau$ and $e_\tau$ from $\chi$ and publish evaluation key $\{\zeta^{(i)}(\tau)\}$ where $\zeta^{(i)}(\tau) = hs_\tau + 2e_\tau + 2\tau f$ in $R_{q_0}$. Then using Lemma 1, we can evaluate rest of the evaluation keys for a level $i$ by simply computing $\zeta^{(i)}(\tau) = \zeta^{(0)}(\tau) \mod q_i$. Then set; secret key $sk = (f)$ and public key $pk = (h, \zeta)$. 

3.1.1.3 Optimizing Relinearization

In homomorphic circuit evaluation using LTV by far the most expensive operation is relinearization. Therefore, it becomes essential to optimize relinearization as much as possible. Recall that the relinearization operation computes a sum of encrypted shifted versions of a secret key $f(x)$ and polynomials $\tilde{c}_\tau(x)$ with coefficients in $\mathbb{F}_2$ extracted from the ciphertext $c$.

$$\tilde{c}(x) = \sum_\tau \zeta_\tau(x) \cdot \tilde{c}_\tau(x)$$

For simplicity we dropped the level indices in superscripts. The ciphertext $\zeta_\tau(x) \in R_q[x]/\langle \Phi(x) \rangle$ values are full size polynomials with coefficients in $R_q$ and do shrink in size over the levels of evaluation after each modulus switching operation. In contrast $\tilde{c}_\tau(x) \in \mathbb{F}_2[x]/\langle \Phi(x) \rangle$ where $\tau$ ranges $\log(q)$. We may evaluate the summation, by scanning the coefficients of the current $\tilde{c}_\tau(x)$ and conditionally shifting and adding $\zeta_\tau(x)$ to the current sum depending on the value of the coefficient. With this approach the computational complexity of relinearization becomes $O(n \log(q))$ polynomial summations or $O(n^2 \log(q))$ coefficient, i.e. $\mathbb{Z}_q$, summations. This approach is useful only for small $n$.

In contrast, if we directly compute the sum after we compute the products we
obtain a more efficient algorithm. The number of polynomial multiplications is $O(\log(q))$ each having a complexity of
$O(n \log(n) \log \log(n))$ with the Schönhage Strassen algorithm [30]. The algorithm simply uses Number Theoretic Transform (NTT) and completes the polynomial multiplication in three steps; conversion of the polynomials to NTT form, digit-wise multiplications, conversion from NTT to polynomial form. After the multiplications, coefficient additions require $O(n \log(q))$ operations. The total complexity of relinearization becomes $O(n \log(n) \log \log(n) \log(q))$ coefficient operations.

Another optimization technique is to store the polynomials $\zeta_r(x)$ in NTT form. This eliminates the time needed for the conversions of $\zeta_r(x)$ at beginning of each multiplication operation. Furthermore, polynomial additions are also performed in NTT form to eliminate $\text{NTT}^{-1}$ conversions to polynomial form. Representing the precomputed NTT form of $\zeta_r(x)$ as $\zeta'_r(x)$ we can rewrite the relinearization operations as follows

$$\tilde{c}(x) = \text{NTT}^{-1}\left[\sum_r \zeta'_r(x) \cdot \text{NTT}[\tilde{c}_r(x)]\right].$$

With this final optimization, we eliminate $2/3$rd of the conversions in each relinearization and obtain nearly 3 times speedup.

### 3.1.2 Description

We give a formal description of the DHS scheme. It is the modified version of LTV scheme that is optimized using the techniques discussed above. The scheme has the operational ring $R_q = \mathbb{Z}_q[x]/\langle \Phi_m(x) \rangle$ where $\Phi_m(x)$ is the $m^{th}$ cyclotomic polynomial. The scheme uses a truncated Gaussian error distribution function $\chi$ for sampling polynomials. The truncated distribution $\chi$ is $B$-bounded which means coefficient
sizes are between range $[-B, B]$. The implementation has following four primitive functions:

- **KeyGen.** Using the security parameter $\lambda$, we create a sequence of modulus for each level as $q_i = q^{d-i}$ in which $q$ is a prime. Later, we sample polynomials $g \in \chi$ and $f' \in \chi$ and compute secret key $f = 2f' + 1$ and public key $h = 2gf^{-1}$ in $\mathbb{Z}_{q_0}$. In the next step, we compute the evaluation keys $\zeta_r^{(0)}(x) = hs_r + 2e_r + 2^\tau f$ which $\{s_r, e_r\} \in \chi$ and $\tau = [0, \lceil \log q_0 \rceil]$. Computed evaluation keys are for level index 0. For the rest of the levels we recycle the evaluation keys using the ring structure. To use the evaluation keys for level index $i$, we simply compute $\zeta_r^{(i)}(x) \equiv \zeta_r(x) \mod q_i$. This reduces the memory requirement significantly.

- **Encrypt.** In order to encrypt a message bit $\mu$ for level $i$, we compute $c^{(i)} = h^{(i)}s + 2e + b$ which $\{s, e\} \in \chi$ and $h^{(i)} = h^{(0)} \mod q_i$.

- **Decrypt.** In order to decrypt at level $i$, we compute $\mu = \lceil c^{(i)}f^{(i)} \rceil_{q_i} \mod 2$.

- **Evaluation.** Homomorphic evaluation requires relinearization and modulus switching after each multiplication or an addition operation. Since the additive operations do not create significant noise like multiplications, we apply the noise reduction techniques only after a multiplication. Relinearization is computed as $\tilde{c}^{(i)}(x) = \sum_\tau \zeta^{(i)}_r(x)\tilde{c}^{(i-1)}_r(x)$. The polynomials $\tilde{c}^{(i-1)}_r(x)$ have the form of $\tilde{c}^{(i-1)}(x) = \sum_\tau 2^\tau \tilde{c}^{(i-1)}_r(x)$. Relinearization is followed by modulus switching which we compute as $\tilde{c}^{(i)}(x) = \lceil \frac{q_i}{q_{i-1}}c^{(i)}(x) \rceil_2$. This reduces the noise level by $\log (q_i/q_{i-1})$ bits. In modulus switching we need to match the parity bits of the messages between the old and new moduli: $\lceil \cdot \rceil_2$. 

32
3.1.3 Coping with Noise

In this section, we describe our approach in managing the growth of noise over the homomorphic evaluation of levels of the circuit. The accumulation of noise from the evaluations of additions adds very little noise compared to that contributed by multiplication. Therefore, as long as we have a reasonably balanced circuit we can focus only on multiplications. Furthermore, in our analysis we focus on noise growth with regards to its effect on the correctness of the scheme. Our goal is to minimize the computational burden, i.e. minimize parameters $q$ and $n$, such that the scheme still correctly decrypts with very high probability.

Consider two plaintexts $m_1, m_2 \in \{0, 1\}$ and parameters $g, s \in \chi$ encrypted using a single user (single key) with no modulus switching specialization of the LTV scheme. The secret key is $f = 2f' + 1$ where $f' \in \chi$. the product of two given ciphertexts $c_1 = E(m_1) = hs_1 + 2e_1 + m_1$ and $c_2 = E(m_2) = hs_2 + 2e_2 + m_2$ yields:

$$c_1c_2 = h^2s_1s_2 + h(s_1m_2 + s_2m_1) + 2[h(s_1e_2 + s_2e_1) + e_1m_2 + e_2m_1 + 2e_1e_2] + m_1m_2$$

To decrypt the resulting ciphertext we compute

$$f^2c_1c_2 = 4g^2s_1s_2 + 2gf(s_1m_2 + s_2m_1) + 2[2gf(s_1e_2 + s_2e_1) +$$

$$+ f^2e_1m_2 + f^2e_2m_1 + 2f^2e_1e_2] + f^2m_1m_2$$

The accumulative noise in the ciphertext should satisfy the following condition and avoid any wraparound to prevent corruption in the message coefficients during de-
cryption:

\[
q/2 > 4n^3B^4 + 4n^3B^3(2B + 1) + 8n^3B^3(2B + 1) + 8n^3B^2(2B + 1)^2 + n^3B^2(2B + 1)^2 \\
> n^3(64B^4 + 48B^3 + 9B^2)
\]

Note that this is the worst case behavior of the norm and therefore decryption will work for most ciphertexts even with a somewhat smaller \(q\).

### 3.1.3.1 Modulus Switching

It will be impractical to evaluate a deep circuit, e.g. AES, using this approach since the norm grows exponentially with the depth of the circuit. To cope with the growth of noise, we employ modulus switching as introduced in [26]. For this, we make use of a series of decreasing moduli \(q_0 > q_1 > \ldots > q_t\); one modulus per level. Modulus switching is a powerful technique that exponentially reduces the growth of noise during computations. The modulus switching operation is done for a ciphertext \(c\) is shown as \(c_{new} = \left\lceil c \cdot q_{i+1}/q_i \right\rceil_2\). The ceil-floor represents the rounding operation and subscript 2 represents matching the parities of \(c\) and \(c_{new}\) in modulus 2. The modulus switching operation is performed by first multiplying the coefficients by \(q_{i+1}/q_i \approx \kappa\) and rounding them to the nearest integer. Later, a parity correction operation performed by adding a parity polynomial \(P_i\). As before, for \(c_1 = E(m_1) = hs_1 + 2e_1 + m_1\) and \(c_2 = E(m_2) = hs_2 + 2e_2 + m_2\) the product of the two ciphertexts gives

\[
c_1c_2 = h^2s_1s_2 + h(s_1m_2 + s_2m_1) + 2[h(s_1e_2 + s_2e_1) + e_1m_2 + e_2m_1 + 2e_1e_2] + m_1m_2
\]
After modulus switching, i.e. multiplication by \( q_1/q_0 \approx \kappa \) and correction of parities symbolized by \( P_i \in \mathbb{D}_{Z^n, \sigma} \) we obtain

\[
c_1c_2\kappa + P_1 = [h^2s_1s_2 + h(s_1m_2 + s_2m_1) + 2h(s_1e_2 + s_2e_1) + e_1m_2 + e_2m_1 + 2e_1e_2] + m_1m_2]\kappa + P_1
\]

After \( i \) levels the ciphertext products (for simplicity assume \( c = c_1 = \ldots = c_{2^i} \)) where each multiplication is followed by modulus switching and parity corrections (symbolized by the \( P_i \)) will be

\[
c_{2^i} = \ldots (((c^2\kappa + P_1)^2\kappa + P_2)^2 \ldots \kappa + P_{2^i})
\]

We may decrypt the result as follows:

\[
c_{2^i}f^{2^i} = \ldots (((c^2\kappa + P_1)^2\kappa + P_2)^2 \ldots \kappa + P_{2^i})f^{2^i}
\]

The correctness condition becomes \( \| c_{2^i}f^{2^i} \|_\infty < q/2 \). Note that due to the final multiplication with the \( f^{2^i} \) term we still have exponential growth in the norm with the circuit depth. Therefore, we need one more ingredient, i.e. relinearization [16], to force the growth into a linear function of the circuit depth. Intuitively, relinearization achieves to linearize the growth by homomorphically multiplying the current ciphertext by \( f \) right before modulus switching.
3.1.3.2 Relinearization and Modulus Switching

After each multiplication level we implement a relinearization operation which keeps the power of $f$ in the ciphertext under control and reduces the chances of wraparound before decryption. Assume we homomorphically evaluate a simple $d$-level circuit $C(m) = m^{2^d}$ by computing repeated squaring, relinearization and modulus switching operations on a ciphertext $c$ where $||c||_{\infty} = B_i$. Recall that for relinearization we compute

$$\tilde{c}^{(i)}(x) = \sum_{\tau} \zeta^{(i)}(x)\tilde{c}^{(i-1)}(x)$$

where each $\zeta^{(i)}(x)$ is of the form $\zeta^{(i)}(x) = h^{(i)}s^{(i)}_\tau + 2e^{(i)}_\tau + 2^\tau f^{(i-1)}$ in $R_{q_{i-1}}$. Substituting this value we obtain

$$\tilde{c}^{(i)}(x) = \sum_{\tau} [h^{(i)}s^{(i)}_\tau + 2e^{(i)}_\tau + 2^\tau (f^{(i-1)})] \tilde{c}^{(i-1)}(x)$$

$$= \sum_{\tau} [h^{(i)}s^{(i)}_\tau + 2e^{(i)}_\tau] \tilde{c}^{(i-1)}(x) + \sum_{\tau} 2^\tau (f^{(i-1)}) \tilde{c}^{(i-1)}(x).$$

Since we are only interested in bounding the growth of noise we assume $s^{(i)}_\tau = s \in \chi$, $g^{(i)}_\tau = g \in \chi$ and $e^{(i)}_\tau = e \in \chi$ and drop unnecessary indices from here on:

$$\tilde{c}^{(i)} = \sum_{\tau} (hs + 2e + 2^\tau f)\tilde{c}^{(i-1)}$$

$$= \sum_{\tau} (hs + 2e)\tilde{c}^{(i-1)} + \sum_{\tau} 2^\tau f \tilde{c}^{(i-1)}$$

$$= \sum_{\tau} (2gf^{-1}s + 2e)\tilde{c}^{(i-1)} + f \tilde{c}^{(i-1)}$$

$$= \sum_{\tau \in [\log(q)]} (2gf^{-1}s + 2e)\tilde{c}^{(i-1)} + \tilde{c}^{(i-1)} f$$
Also factoring in the modulus switching and parity correction steps and substituting 
\( \tilde{c}^{(i-1)} = c^2 \) we obtain the reduced noise ciphertext \( \tilde{c}' \) as

\[
\tilde{c}' = \left( \sum_{\tau \in [\log(q)]} (2gf^{-1}s + 2e)\tilde{c}_\tau + c^2 f \right) \kappa + P
\]

where \( P \in \{0, 1\} \) represents the parity polynomial. The distribution (and norm) of the left summand in the inner parenthesis is constant over the levels. To simplify the equation we use the shorthand \( X_0 = \sum_{\tau \in [\log(q)]} (2gf^{-1}s + 2e)\tilde{c}_\tau \) where the index is used to indicate the level. Assume we use \( Y_i \) to label the ciphertext (output) of evaluation level \( i \) then

\[
Y_1 = (fc^2 + X_0) \kappa + P_1.
\]

Assume we continue this process, i.e. squaring, relinearization, modulus switching and parity correction for \( d \) levels and then decrypt by multiplying the resulting ciphertext by \( f \) we obtain:

\[
Y_i = (fY_{i-1}^2 + X_{i-1}) \kappa + P_i, \quad \text{for } i = 1, \ldots, d-1
\]

To decrypt \( Y_{d-1} \) we need \( ||Y_{d-1}f||_\infty < q/2 \). Now first note that

\[
Y_i f = [(fY_{i-1}^2 + X_{i-1}) \kappa + P_i] f
\]

\[
= ((Y_{i-1}f)^2 + X_{i-1}f) \kappa + P_i f
\]
Therefore, $$||Y_i f||_\infty \leq ||(Y_{i-1} f)^2||_\infty \kappa + ||X_{i-1} f||_\infty \kappa + ||P_i f||_\infty$$ Also note that

$$||f X_i||_\infty = ||\sum_{\tau} (2g s_{\tau} + 2e_{\tau} f) \tilde{c}_{\tau}||_\infty$$

$$\leq ||\sum_{\tau} 2g s_{\tau} \tilde{c}_{\tau}||_\infty + ||\sum_{\tau} 2e_{\tau} f \tilde{c}_{\tau}||_\infty$$

Since $$||f||_\infty = 2B + 1$$, $$||\tilde{c}_{\tau}||_\infty = ||s_{\tau}||_\infty = ||e_{\tau}||_\infty = ||g||_\infty \leq B$$ and all polynomials have at most degree $$n$$ it follows that

$$||f X_i||_\infty \leq (2n^2 B^3 + 2n^2 B^2 (2B + 1)) \log(q_i)$$

$$\leq n^2 (6B^3 + 2B^2) \log(q_i)$$

Now let $$B_i$$ denote an upper bound on the norm of a decrypted ciphertext entering level $$i$$ of leveled circuit, i.e. $$B_i \geq ||f Y_i||_\infty$$. Using the equation $$B_i \geq ||f Y_i||_\infty$$ we can set the norm $$||(Y_{i-1} f)^2||_\infty$$ as

$$||(Y_{i-1} f)^2||_\infty \leq n ||(Y_{i-1} f)||_\infty \cdot ||(Y_{i-1} f)||_\infty$$

$$\leq n B_i^2.$$

The norm of the output grows from one level to the next including multiplication (squaring with our simplification), relinearization and modulus switching as follows

$$B_i \leq [n B_{i-1}^2 + n^2 (6B^3 + 2B^2) \log(q_i)] \kappa + n (2B + 1) . \quad (3.1)$$

Notice the level independent (fixed) noise growth term on the right summand of the recursion. In practice, $$\kappa$$ needs to be chosen so as to stabilize the norm over the levels of computation, i.e. $$B_1 \approx B_2 \approx \ldots \approx B_{d-1} < q_{d-1}/2$$. Finally, we can make
the accounting a bit more generic by defining circuit parameters $\nu_i$ which denote the maximum number of ciphertext additions that take place in evaluation level $i$. With this parameter we can bound the worst case growth simply by multiplying any ciphertext that goes into level $i+1$ by $\nu_i$ as follows

$$B_i \leq \left[ \nu_i^2 n B_{i-1}^2 + n^2 (6B^3 + 2B^2) \log(q_i) \right] \kappa + n(2B + 1). \tag{3.2}$$

### 3.1.3.3 Average Case Behavior

In our analysis so far we have considered worst case behavior. When viewed as a distribution, the product norm $||ab||_\infty$ will grow much more slowly and the probability that the norm will reach the worst case has exponentially small probability. To take advantage of the slow growth we can instead focus on the growth of the standard deviation by modeling each coefficient of $a$ and $b$ as a scaled continuous Gaussian distribution with zero mean and deviation $r = B$. The coefficients of the product $(ab)_i = \sum_{i=0,...,n-1} a_i b_{n-1-i}$, behave as drawn from a scaled chi-square distribution with $2n$ degrees of freedom, i.e. $\chi^2(2n)$. To see this just note each coefficient product can be rewritten as $a_i b_{n-1-i} = \frac{1}{4} (a_i + b_{n-1-i})^2 - \frac{1}{4} (a_i - b_{n-1-i})^2$. As $n$ becomes large $\chi^2(2n)$ becomes close to an ideal Gaussian distribution with variance $4n$. Thus $r((ab)_i) \approx \sqrt{n}B^2$ for large $n$. Therefore, a sufficiently good approximation of the expected norm may be obtained by replacing $n$ with $\sqrt{n}$ in Equation 3.2 as follows

$$B_{i,\text{avg}} \approx \left[ \nu_i \sqrt{n} B_{i-1,\text{avg}}^2 + n(6B^3 + 2B^2) \log(q_i) \right] \kappa + \sqrt{n}(2B + 1).$$

For practical values of $n$ and small $B$ the left-hand-side dominates the other terms in the equation. Further simplifying we obtain

$$B_{i,\text{avg}} \approx \left[ \nu_i \sqrt{n} B_{i-1,\text{avg}}^2 + n(6B^3 + 2B^2) \log(q_i) \right] \kappa. \tag{3.3}$$
Assuming nearly fixed $\nu_i \approx \nu$, if we set $1/\kappa = \epsilon \left[ \nu \sqrt{nB_{i-1,\text{avg}}^2} + n(6B^3 + 2B^2) \log(q_0) \right]$ for a small constant $\epsilon > 1$ we can stabilize the growth of the norm and keep it nearly constant over the levels of the evaluation. Initially recursive computation will start with $B_{0,\text{avg}} = 2$ and the noise will grow in a few steps until it is stable. Our experiments showed that after 5–6 levels of computation the noise stabilizes between $2^{10} < B_{i,\text{avg}} < 2^{15}$ for lattice of dimensions $2^{12} < n < 2^{17}$. By taking $B_{i,\text{avg}} = 2^{12}$ and $\nu = 1$ we tabulated the cutting sizes and attainable levels of homomorphic evaluation in Table 3.1.

We can simplify the noise equation further to gain more insight on the noise growth and subsequently on how the modulus $q$ and the dimension $n$ will be affected. Fix $B = 2$ and assume we are interested in evaluating a depth $t$ circuit and therefore $q_0 = p^{t+1}$. Also since with our $q_0 = p^{t+1}$ specialization $1/\kappa \approx p$ and since $p \gg \nu$ neglecting $\log(\nu)$ we can simplify our noise estimate as follows:

$$p \approx \nu \sqrt{nB_{i-1,\text{avg}}^2} + 56n(t + 1) \log p .$$

This nonlinear equation displays the relationship between the chosen dimension $n$ and depth of circuit we wish to support and the number of bits we need to cut in each level. However, $p$ and $n$ are not independent since $n$ and $q = p^{t+1}$ are tied through the Hermite factor $\delta = (\sqrt{q}/4)^{1/(2n)} = (\sqrt{p^{t+1}/4})^{1/2n}$ and $p = (4\delta^{2n})^{2/(t+1)}$. Substituting $p$ yields

$$(4\delta^{2n})^{2/(t+1)} \approx \nu \sqrt{nB_{i-1,\text{avg}}^2} + 56n(t + 1) \log (4\delta^{2n})^{2/(t+1)}$$

$$\approx \nu \sqrt{nB_{i-1,\text{avg}}^2} + 56n(t + 1)2/(t + 1)(\log 4 + \log(\delta^{2n}))$$

$$\approx \nu \sqrt{nB_{i-1,\text{avg}}^2} + 112n(2 + 2n \log(\delta)) .$$
By taking the logarithm and fixing $\nu$ and the security level $\delta$ we see that $t \sim O(n/\log(n))$.

<table>
<thead>
<tr>
<th>log($n$)</th>
<th>log($q$)</th>
<th>Worst Case</th>
<th>Average Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>log($1/K$)</td>
<td>#L</td>
<td>log($1/K$)</td>
<td>#L</td>
</tr>
<tr>
<td>12</td>
<td>155</td>
<td>36</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>311</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>622</td>
<td>39</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>1244</td>
<td>40</td>
<td>30</td>
</tr>
<tr>
<td>16</td>
<td>2488</td>
<td>42</td>
<td>58</td>
</tr>
<tr>
<td>17</td>
<td>4976</td>
<td>44</td>
<td>112</td>
</tr>
</tbody>
</table>

Table 3.1: Worst case and average case number of bits $\log(1/K)$ required to cut to correctly evaluate a pure multiplication circuit of depth $L$ with $B = 2$ and $\alpha = 6$ for $n$ and $q$ chosen such that $\delta(n,q) = 1.0066$.

### 3.1.3.4 Failure Probability

Equation 3.3 tells us that we can use a much smaller $q$ than that determined by the worst case bound in Equation 3.2 if are willing to accept a small decryption failure probability at the expense of a small margin. The failure probability is easily approximated. If we set $q/2 > \alpha B_{\text{avg}}$ where $\alpha > 1$ captures the margin, then $\alpha B_{\text{avg}}/\sigma$ determines how much of the probability space we cover in a Gaussian distribution $\mathcal{N}(\mu = 0, \sigma)$. The probability for the norm of a single coefficient to exceed a preset margin $\alpha \sigma$ becomes $\text{Prob}[\| (ab)_i \|_{\infty} > \alpha \sigma] \approx 1 - \text{erf} \left( \frac{\alpha}{\sqrt{2}} \right)$ where $\text{erf}$ denotes the error function. For the entire product polynomial we can approximate the worst case probability by assuming independent product coefficients as $\text{Prob}[\| ab \|_{\infty} > \alpha \sigma] \approx 1 - \text{erf} \left( \frac{\alpha}{\sqrt{2}} \right)^n$. Having dependent coefficients (as they really are) will only improve the success probability. For instance, assuming $n = 2^{14}$ and $\sigma = B$ with a modest margin of $\alpha = 7$ we obtain a reasonably small failure probability of $2^{-60}$. 

41
3.1.4 Parameter Selection in the LTV

A significant challenge in implementing and improving LTV is parameter selection. In [16] and [19] the security analysis is mostly given in asymptotics by reduction to the related learning with errors (LWE) problem [20]. In this section, after briefly reviewing the security of the LTV scheme we summarize the results of our preliminary work on parameter selection.

**DSPR Problem.** The scheme proposed by Stehlé and Steinfeld [20] is a modification to NTRU [19], whose security can be reduced to the hardness of the Ring-LWE (RLWE) problem. The reduction relies on the hardness of the Decisional Small Polynomial Ratio (DSPR_{\phi,q,\chi}) Problem defined as follows:

**Definition 1** ([20, 16] Decisional Small Polynomial Ratio (DSPR_{\phi,q,\chi}) Problem). Let \( \phi(x) \in \mathbb{Z}[x] \) be a polynomial of degree \( n \), \( q \in \mathbb{Z} \) be a prime integer, and let \( \chi \) denote a distribution over the ring \( R = \mathbb{Z}[x]/(\phi(x)) \). The decisional small polynomial ratio problem DSPR_{\phi,q,\chi} is to distinguish between the following two distributions:

- a polynomial \( h = gf^{-1} \), where \( f \) and \( g \) are sampled from the distribution \( \chi \) (with \( f \) invertible over \( R_q \)), and

- a polynomial \( h \) sampled uniformly at random over \( R_q \).

Stehlé and Steinfeld have shown that the DSPR_{\phi,q,\chi} problem is hard even for unbounded adversaries when \( n \) is a power of two, \( \phi(x) = x^n + 1 \) is the \( n \)-th cyclotomic polynomial, and \( \chi \) is the discrete Gaussian \( \mathcal{D}_{\mathbb{Z}^n,\sigma} \) for \( \sigma > \sqrt{q \cdot \text{poly}(n)} \). Specifically, the security reduction is obtained through a hybrid argument as follows

1. Recall that for the LTV scheme, the public key is of the form \( h = 2gf^{-1} \) where \( g, f \) chosen from a Gaussian distribution \( \chi \) where \( f \) is kept secret. If the DSPR problem is hard, we can replace \( h = 2gf^{-1} \) by some uniformly sampled \( h' \).
2. Once \( h \) is replaced by \( h' \), the encryption \( c = h's + 2e + m \) takes the form of the RLWE problem and we can replace the challenge cipher by \( c' = u + m \) with a uniformly sampled \( u \), thereby ensuring security.

In this way we can reduce the LTV scheme into a RLWE problem. However, the RLWE problem is still relatively new and lacks thorough security analysis. A common approach is to assume that RLWE follows the same behavior as the LWE problem [23]. Then, the second part of the reduction immediately suggest a distinguishability criteria in the lattice setting. The matrix \( \mathbf{H} \) is derived from the coefficients of the public key polynomial \( h \) as

\[
\mathbf{H} = \begin{pmatrix}
    h_0 & h_1 & \cdots & h_{n-1} \\
    -h_{n-1} & h_0 & \cdots & h_{n-2} \\
    \vdots & \vdots & \ddots & \vdots \\
    -h_1 & -h_2 & \cdots & h_0
\end{pmatrix}
\]

We can connect the \( q \)-ary lattice \( \Lambda_{\mathbf{H}} \) definition to the distinguishability of the masks in the LTV scheme. For simplicity we consider only a single level instantiation of LTV. To encrypt a bit \( b \in \{0, 1\} \) with a public key \((h, q)\), we first generate random samples \( s \) and \( e \) from \( \chi \) and compute the ciphertext \( c = hs + 2e + b \pmod{q} \). Here we care about the indistinguishability of the mask \( hs + 2e \) from a randomly selected element of \( R_q \). We can cast the encryption procedure in terms of the \( q \)-ary lattice \( \Lambda_{\mathbf{H}} \) as follows \( \mathbf{c} = \mathbf{H}s + 2\mathbf{e} + \mathbf{b} \) where we use boldface symbols to denote the vector representation of polynomials obtained trivially by listing the polynomial coefficients. Then the decisional LWE problem in the context of the LTV scheme is to distinguish between the following two distributions:

- a vector \( \mathbf{v} \) sampled randomly from \( \mathbb{Z}^n \), and
• a vector $v = Hs + 2e$ where $e$ and $s$ are sampled from the distribution $\mathbb{D}_{Z^n, \sigma}$, respectively.

Given a vector $v$ we need to decide whether this is a randomly selected vector or close to an element of the $q$-ary lattice $\Lambda_H$. The natural approach [31] to distinguishing between the two cases is to find a short vector $w$ in the dual lattice $\Lambda^*_H$ and then check whether $w \cdot v^T$ is close to an integer. If not then we decide that the sample was randomly chosen; otherwise we conclude that $v$ is a noisy lattice sample. Here we can follow the work of Micciancio and Regev [31] (Section 5.4) who considered the security of LWE distinguishability with the dual basis approach. They note that this method is effective as long as the perturbation of $v$ from a lattice point in the direction of $w$ is not much bigger than $1/||w||$. Since our perturbation is Gaussian, its standard deviation in the direction of $w$ is $r' = \sqrt{2}r$. Therefore we need $r \gg 1/(\sqrt{2}||w||)$. Micciancio and Regev note that restricting $r' > 1.5/(\sqrt{2}||w||)$ provides a sufficient security margin and derive the following criteria

$$r' \geq 1.5q \cdot \max \left( \frac{1}{q}, 2^{-2\sqrt{n \log(q) \log(\delta)}} \right)$$

This gives us another condition to satisfy once $q$, $n$ and $\delta$ are selected.

**Concrete Parameters.** Stehlé and Steinfeld reduced the security of their scheme to the hardness of the RLWE. Unfortunately, the reduction only works when a wide distribution $\mathbb{D}_{Z^n, \sigma}$, i.e. $\sigma > \sqrt{q} \cdot \text{poly}(n)$ is used. Due to noise growth with such an instantiation the LTV scheme [16] will not be able to support even a single homomorphic multiplication. Therefore [16] assumes the hardness of $\text{DSPR}_{\phi, q, \chi}$ for smaller $r$ values in order to support homomorphic evaluation. The impact of the new parameter settings to the security level is largely unknown and requires further research. However, even if we assume that the $\text{DSPR}_{\phi, q, \chi}$ problem is difficult, we
still need to ensure the hardness of the RLWE problem. As we discussed above, a common approach is to assume that it follows the same behavior as the LWE problem. Under this assumption only, we can select parameters. If we omit the noise, given the prime number $q$ and $k$-bit security level, the dimension is bounded as in [23] as $n \leq \log(q)(k + 110)/7.2$. This bound is based on experiments run by Lindner and Peikert [32] with the NTL library. The bound is rather loose since it is not exactly clear how the experiments will scale to larger dimensions. For instance, [32] ran experiments with the NTL library but extrapolates a running time which grows as $2^{O(k)}$ where $k$ is the block size, whereas NTL’s enumeration implementation grows as $2^{O(k^2)}$. Another issue is the assumption of $\delta_0 = O(2^{1/k})$ which should be $\delta_0 = O(k^{1/k})$. On the positive side, these simplifications yield a loose upper bound and should not negatively affect the security.

For example, given a 256-bit prime $q$, an 80-bit security level will require dimension $n = 6756$. This large estimate is actually an upper bound and assumes that the LTV scheme can be reduced to the RLWE problem. It is not clear whether the reverse is true, i.e. whether attacks against the RLWE problem apply to the LTV scheme. For instance, the standard attack on the LWE problem requires many samples generated with the same secret $s$. However, in the LTV scheme, the corresponding samples are ciphertexts of the form $c = h's + 2e + m$, where the $s$ polynomials are randomly generated and independent. This difference alone suggests that standard attacks against LWE problems cannot be directly applied to the LTV scheme.

**NTRU Lattice Attacks.** As a variant of NTRU, the LTV scheme suffers from the same attack as the original NTRU. We can follow a similar approach as in the original NTRU paper [19] (see also [33]) to find the secret $f$: Consider the following $2n$ by $2n$ NTRU lattice where the $h_i$ are the coefficients of $h = 2gf^{-1}$. Let $A_L$ be
the lattice generated by the matrix.

\[ \mathcal{L} = \begin{pmatrix} \mathbf{I} & \mathbf{H} \\ 0 & q \mathbf{I} \end{pmatrix} \]

where \( \mathbf{H} \) is derived from the public key polynomial \( h \) as defined above. Clearly, \( \Lambda_{\mathcal{L}} \) contains the vector \( a = (f, 2g) \) which is short, i.e. \( \|a\|_\infty \leq 4B + 1 \). Now the problem is transformed to searching for short lattice vectors. Quite naturally, to be able to select concrete parameters with a reasonable safety margin we need to have a clear estimate on the work factor of finding a short vector in \( \Lambda_{\mathcal{L}} \).

In what follows, we present a Hermite (work) factor estimate, and experimental results that will allow us to choose safe parameters.

### 3.1.4.1 Hermite Factor Estimates

Gama and Nguyen [34] proposed a useful approach to estimate the hardness of the SVP in an \( n \)-dimensional lattice \( \Lambda_{\mathcal{L}} \) using the Hermite factor \( \delta \) defined as

\[
\left( \prod_{i=1}^{d} \lambda_i(\Lambda_{\mathcal{L}}) \right)^{1/d} \leq \sqrt{\delta} \text{VOL}(\Lambda_{\mathcal{L}})^{1/n},
\]

where in the equation \( \lambda_i(\Lambda_{\mathcal{L}}) \) denotes the \( i \)-th minimum of the lattice \( \Lambda_{\mathcal{L}} \) and \( d \) is any number between the range 1 \( \leq d \leq n \). More practically we can compute \( \delta \) as

\[
\delta^n = \|b_1\|/\det(\Lambda_{\mathcal{L}})^{1/n}
\]

where \( \|b_1\| \) is the length of the shortest vector or the length of the vector for which we are searching. The authors also estimate that, for larger dimensional lattices, a factor \( \delta^n \leq 1.01^n \) would be the feasibility limit for current lattice re-
duction algorithms. In [32], Lindner and Peikert gave further experimental results regarding the relation between the Hermite factor and the break time as $t(\delta) := \log(T(\delta)) = 1.8/\log(\delta) - 110$. For instance, for $\delta^n = 1.0066^n$, we need about $2^{80}$ seconds on the platform in [32].

For the LTV scheme, we can estimate the $\delta$ of the NTRU lattice and thus the time required to find the shortest vector. Clearly, the NTRU lattice has dimension $2n$ and volume $q^n$. However, the desired level of approximation, i.e. the desired $||b_1||$ is unclear. In [34], Gama and Nguyen use $q$ as the desired level for the original NTRU. However, for the much larger $q$ used in the LTV scheme, this estimate will not apply. In particular, Minkowski tells us that $\Lambda_L$ has a nonzero vector of length at most $\det(L)^{1/t}\sqrt{t}$ where $t$ is the dimension. There will be exponentially many (in $t$) vectors of length $\text{poly}(t)\det(L)^{1/t}$.

To overcome this impasse we make use of an observation by Coppersmith and Shamir [35]: we do not need to find the precise secret key since most of the vectors of similar norm will correctly decrypt NTRU ciphertexts. Setting $||b_1||$ as the norm of the short vector we are searching for and volume as $q^n$, we can simplify the Hermite factor to

$$\delta^{2n} = \frac{||b_1||}{(q^n)^{1/2n}}.$$ 

Following the recommendation of [35], we set the norm of $b_1$ as $q/4$. Coppersmith and Shamir observed that $q/10$ can ensure a successful attack in the majority of cases for NTRU with dimension $n = 167$ while $q/4$ is enough to extract some information. With a much larger dimension used than in NTRU, we may need a $||b||$ even smaller than $q/10$ to fully recover a usable key. However, we choose $q/4$ here to provide a conservative estimate of the security parameters. Thus $\delta^{2n} = \frac{q/4}{q^{1/2}} = \sqrt{q}/4$. In Table 3.2 we compiled the Hermite factor for various choices of $q$ and $n$ values.
Table 3.2: Hermite Factor estimates for various dimensions $n$ and sizes of $q$. According to [32] for $\delta^n = 1.0066^n$, we need about $2^{80}$ seconds computation on a current PC. Therefore, we need $\delta < 1.0066$ and the smaller $\delta$ the higher the security margin will be.

### 3.1.4.2 Experimental Approach

As a secondary countermeasure we ran a large number of experiments to determine the time required to compromise the LTV scheme following the lattice formulation of Hoffstein, Pipher, and Silverman with the relaxation introduced by Coppersmith and Shamir [35]. We generated various LTV keys with coefficient size $\log(q) = 1024$ and various dimensions. To search the short vectors required in the attacks described as above we used the Block-Korkin-Zolotarev (BKZ) [36] lattice reduction functions in Shoup’s NTL Library 6.0 [37] linked with the GNU Multiprecision (GMP) 5.1.3 package. We set the LLL constant to 0.99 and ran the program with the block size 2. The block size has exponential impact on the resulting vector size and the running time of the algorithm. For the dimensions covered by our experiments, even the lowest block size was enough to successfully carry out attacks. Experiment results show that with the same block size, the size of the recovered keys grows exponentially with the dimension and the time for the algorithm grows polynomially with the dimension. As discussed above, the recovered vectors are only useful if they are shorter than $q/4$. When the dimension is sufficiently large we end up with vectors longer than this limit, and we will need larger block sizes causing an exponential rise in the time required to recover a useful vector [36]. From the collected data, we
estimated that the block size of 2 can be used until about dimension $n = 26777$.

Clearly, we cannot run test on such large dimensions to examine the exponential effects and estimate the cost for higher dimensions. To investigate the detailed impact of larger block sizes, we ran the experiment on low dimensions with higher block sizes and checked the changes on the recovered key sizes and the running time. The result of the experiment follows the prediction of [36], i.e. the result vector size decreases exponentially while the running time grows exponentially with the block size. Assuming that the higher dimensions follow similar rates, we estimate the security level for higher dimensions in Table 3.3. The estimation assumes the relation between parameters follows a similar pattern for low dimension and high dimensions and ignores all sub-exponential terms\(^1\). Therefore the estimated security level is not very precise. However, the results are not far off from what the Hermite factor estimate predicts. For instance, our experiments predict a 80-bit security for dimension $n = 28940$ with $\log(q) = 1024$. The Hermite work factor estimate for the same parameters yields $\delta = 1.0061$. This is slightly more conservative than [32] whose experiments found that $\delta = 1.0066$ for the same security level.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
Dimension & 28340 & 28940 & 30140 & 31300 & 32768 \\
Security & 70 & 80 & 100 & 120 & 144 \\
\hline
\end{tabular}
\caption{Estimated security level with BKZ. Running times were collected on an Intel Xeon 2.9 GHz machine and converted to bits by taking the logarithm.}
\end{table}

### 3.1.4.3 BKZ 2.0

In a recent work, van de Pol and Smart [38] demonstrated that it is possible to work with lattices of smaller dimensions while maintaining the same security level by utilizing the BKZ-2.0 simulator of Chen and Nguyen. They argue that general

\[^1\text{Ignoring those terms will result in a more conservative estimation.}\]
assumption of a secure Hermite factor $\delta_B$, for a lattice basis $B$, that works for every dimensional lattice is not true. Therefore, one should take into account the hardness of lattice basis reduction in higher dimensions during the parameter selection process. The authors use the following approach to determine the Hermite factor $\delta_B$ and the dimensions $(n, q)$ of the lattice. First, the security parameter $\text{sec}$, e.g. 80, 128 or 256, is selected which corresponds to visiting $2^{\text{sec}}$ nodes in the BKZ algorithm. Then the lattice dimension $d$ and $\delta_B$ are chosen such that the reduction works by visiting $2^{\text{sec}}$ nodes. The evaluation is carried out with the simulator of Chen and Nguyen [39] for various block sizes and number of rounds. This results in a Hermite factor $\delta_B$ as a function of lattice dimension $d$ and security parameter $\text{sec}$. Lastly, they use the Hermite factor $\delta_B$ to obtain $(n, q)$ using the distinguishing attack analysis of Micciancio and Regev [31].

This work was later revisited by Lepoint and Naehrig [1]. Pol and Smart [38] only computed the Hermite factor for powers of two and used linear interpolation to match the enumeration costs to compute the Hermite factors. On the other hand Lepoint and Naehrig performed the experiments for all dimensions from 1000 to 65000 and further they used quadratic function interpolation to set the missing values of enumeration costs. This results in a more precise Hermite factor computation. Also, the authors rely on the more recent work of Chen and Nguyen [40] for enumeration costs to determine the Hermite factors. Their Hermite factor computation is given in Table 3.4.

<table>
<thead>
<tr>
<th>Dimension</th>
<th>1,000</th>
<th>5,000</th>
<th>10,000</th>
<th>15,000</th>
<th>20,000</th>
<th>25,000</th>
<th>30,000</th>
<th>40,000</th>
<th>50,000</th>
<th>60,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sec} = 64$</td>
<td>1.00851</td>
<td>1.00896</td>
<td>1.00931</td>
<td>1.00940</td>
<td>1.00948</td>
<td>1.00954</td>
<td>1.00964</td>
<td>1.00972</td>
<td>1.00979</td>
<td></td>
</tr>
<tr>
<td>$\text{sec} = 80$</td>
<td>1.00763</td>
<td>1.00799</td>
<td>1.00811</td>
<td>1.00826</td>
<td>1.00833</td>
<td>1.00839</td>
<td>1.00846</td>
<td>1.00851</td>
<td>1.00857</td>
<td>1.00862</td>
</tr>
<tr>
<td>$\text{sec} = 128$</td>
<td>1.00592</td>
<td>1.00609</td>
<td>1.00619</td>
<td>1.00624</td>
<td>1.00628</td>
<td>1.00629</td>
<td>1.00634</td>
<td>1.00638</td>
<td>1.00641</td>
<td>1.00644</td>
</tr>
</tbody>
</table>

Table 3.4: Hermite factor $\delta$ estimates for security level $\text{sec}$ reported in [1].
3.2 F-NTRU

After the instantiation of LTV scheme, an attack is introduced by Albrecht, Bai and Ducas [22] specifically targeting NTRU based encryption schemes. Here in this section, we first discuss the effects of the attack to the NTRU based schemes. Later, we introduce a new FHE scheme called F-NTRU that is resilient to the subfield attack. We provide security and noise analysis and also share performance results for our implementation.

3.2.1 Impact of the Subfield Lattice Attack on LTV and YASHE’

On the downside many of these assumptions are still open to debate from a security point of view. A very recent work by Albrecht, Bai and Ducas [22] painfully demonstrated this fact. The authors exploit the presence of a subfield to solve the NTRU problem for large moduli $q$ and show that when the NTRU parameters are chosen poorly it becomes possible to norm-down the NTRU public key $h$ to a subfield yielding an easier lattice problem. Consequently, any sufficiently good solution may be lifted to a short vector in the full NTRU lattice. The attack works when the secret key $f$ is chosen from a narrow distribution, e.g. $\|f\| \leq \sqrt{q}$ and when the polynomial modulus is chosen such that a subfield of reasonable size exists. In this setting, Albrecht et al. show that the DSPR problem is not as hard as believed thereby invalidating the basic assumption in the LTV [16] and YASHE’ [2] schemes. Thus, the subfield lattice attack significantly diminishes the asymptotic security of both schemes.

Both LTV and YASHE’ rely on the secret key from being sampled from a narrow distribution to support even a single homomorphic multiplication. This eliminates
the possibility of sampling the key from a wider distribution. The adverse effect of
the attack could be mitigated, by maximizing size of the subfield as recommended
in [22]. Even then, the lattice dimension and parameters need to be increased to
restore the projected security level of LTV and YASHE’. Another important side
effect of the Subfield Lattice Attack is that it makes the selection of parameters that
support batching rather difficult.

### 3.2.2 Our proposal: F-NTRU Scheme

Here we propose a new scheme called F-NTRU that shares the goals of the GSW
construction [21], i.e. no evaluation keys, no expensive relinearization operations,
no modulus switching and simple homomorphic additions and multiplications. To
this end we adopt the flattening approach of [21] and apply it to the NTRU variant,
i.e. NTRU’, by Stehle and Steinfeld [20].

**Preliminaries.** We work in $R_q = \mathbb{Z}_q[x]/(x^n + 1)$. We adapt two functions from [21]
to work in the polynomial setting as follows:

- **Bit-Decomposition:** The BitDecomp function takes a ciphertext polynomial
  $c(x)$ and splits them into binary polynomials and forms a vector as:

  $$\tilde{c}(x) = \text{BitDecomp}(c(x)) = [c_{\ell-1}(x)c_{\ell-2}(x)\ldots c_0(x)].$$

  Here a polynomial $c_i(x)$ with index $i$ represents the binary polynomial that
  is formed using the $i^{th}$ bit index of the coefficients of $c(x)$. One may easily
  reconstruct the ciphertext $c(x)$ by simply computing:

  $$c(x) = \sum_{i=0}^{\ell-1} 2^i \cdot c_i(x) \in R_q.$$
Note that when we are computing $\text{BitDecomp}^{-1}$, the elements in the vector do not necessarily have to be binary polynomials. It is possible that polynomials can contain coefficients that are not bits.

- **Flatten**: When we are performing arithmetic operations, i.e. addition and multiplication, in our scheme, the elements of a flattened ciphertext vector lose their binary form. These extra bits in the coefficients of the polynomials cause additional noise in subsequent arithmetic operations. To prevent this, we use Flatten. Flatten restructures all the elements of the ciphertext vector into binary polynomials. We evaluate the Flatten operation as follows:

$$\text{Flatten} (\vec{c}(x)) = \text{BitDecomp} \left( \text{BitDecomp}^{-1} (\vec{c}(x)) \right).$$

Here in the equation $\text{BitDecomp}^{-1}$ converts the ciphertext vector into a full ciphertext polynomial in $R_q$. Later, using $\text{BitDecomp}$ we convert the ciphertext into a binary polynomial vector again. Basically, this method carries over the extra bits of an element in the vector from least significant to most significant and performs a modular reduction using $q$ to prevent overflow in the overall scheme.

**The F-NTRU Scheme.** The primitive operations of F-NTRU are defined as follows:

- **KeyGen**: We use the key generation method of NTRU' to select our parameters. For a security parameter $\lambda$, we choose our message modulus as 2, modulus $q = q(\lambda)$, polynomial degree $n = n(\lambda)$ where $n$ is power of 2. Also we set Gaussian distributions $\chi_{\text{err}} = \chi_{\text{err}}(\lambda)$ and $\chi_{\text{key}} = \chi_{\text{key}}(\lambda)$. Sample $g, f' \in \chi_{\text{key}}$ and set public key $h = 2gf^{-1}$ and secret key $f = 2f' + 1$.

- **Encrypt**: In order to encrypt a message $\mu$, we create a vector with length
\( \ell = \log q \). Later, we fill the elements with encryptions of zeros using the NTRU’ scheme:

\[
\vec{c} = \{\text{Enc}_{\ell-1}(0), \text{Enc}_{\ell-2}(0), \ldots, \text{Enc}_0(0)\}
\]

\[
= \{c_{\ell-1}, c_{\ell-2}, \ldots, c_0\},
\]

where \( \text{Enc}_i(0) = hs_i + 2e_i + 0 \). We call this the ciphertext vector. We take the transpose of the vector to list the elements in row and apply \text{BitDecomp} to the ciphertexts to turn the vector into a \( \ell \times \ell \) matrix:

\[
c = \text{BitDecomp}((\vec{c}\top))
\]

Then, we use the matrix to encrypt a message \( \mu \) by evaluating:

\[
C = \text{Flatten}(I_{\ell} \cdot \mu + c).
\]

Here, in the equation \( I_{\ell} \) is identity matrix with size \( \ell \).

- **Decrypt**: To decrypt the message we take the first row of the matrix and apply Inverse-Bit-Decomposition to form a NTRU’ ciphertext:

\[
\text{BitDecomp}^{-1}\{c_{(0,\ell-1)}, c_{(0,\ell-2)}, \ldots, c_{(0,1)}, c_{(0,0)}\} = c_0.
\]

Once the NTRU’ ciphertext is constructed, we are able to decrypt the message using the secret key \( f \) for the NTRU’ scheme as \( \lfloor c_0 f \rfloor \mod 2 = \mu \).

- **Eval.** The homomorphic XOR and AND operations are simply computed as matrix addition and multiplication operations, respectively followed by a
\[ C' = \text{Flatten}(C + \tilde{C}) = \]

\[
\begin{bmatrix}
  c(3,3) + \tilde{c}(3,3) + \mu + \tilde{\mu} & c(3,2) + \tilde{c}(3,2) & c(3,1) + \tilde{c}(3,1) & c(3,0) + \tilde{c}(3,0) \\
  c(2,3) + \tilde{c}(2,3) & c(2,2) + \tilde{c}(2,2) + \mu + \tilde{\mu} & c(2,1) + \tilde{c}(2,1) & c(2,0) + \tilde{c}(2,0) \\
  c(1,3) + \tilde{c}(1,3) & c(1,2) + \tilde{c}(1,2) & c(1,1) + \tilde{c}(1,1) + \mu + \tilde{\mu} & c(1,0) + \tilde{c}(1,0) \\
  c(0,3) + \tilde{c}(0,3) & c(0,2) + \tilde{c}(0,2) & c(0,1) + \tilde{c}(0,1) & c(0,0) + \tilde{c}(0,0) + \mu + \tilde{\mu}
\end{bmatrix}
\]

Figure 3.1: Homomorphic XOR operation

\[ C' = \text{Flatten}(C + \tilde{C}) , \quad C' = \text{Flatten}(C \cdot \tilde{C}) . \]

### 3.2.2.1 Correctness of Homomorphic Circuit Evaluation

The correctness of encryption/decryption trivially follows from the correctness of the NTRU' scheme. In this section we briefly demonstrate how the NTRU' ciphertext and associated F-NTRU ciphertext matrix forms are preserved thus allowing homomorphic evaluation. For clarity we use ciphertexts of sizes \( \ell = 4 \).

First note that \( \text{BitDecomp}^{-1}(C) \) is actually a vector that contains the encryptions of message bits scaled by powers of 2:

\[
\text{BitDecomp}^{-1}(\text{Flatten}(I_\ell \cdot \mu + c)) = \\
\left[ c_{\ell-1} + 2^{\ell-1} \cdot \mu , \ldots , c_1 + 2^1 \cdot \mu , c_0 + 2^0 \cdot \mu \right]^\top .
\]

This is the form we need to preserve throughout homomorphic evaluations for correctness.

**Homomorphic XOR.** A homomorphic XOR operation between two ciphertext matrices is computed as shown in Figure 3.1. When we apply \( \text{BitDecomp}^{-1} \) to the
rows of the addition matrix we obtain:

\[
[(c_3 + \tilde{c}_3) + 8(\mu + \tilde{\mu}), (c_2 + \tilde{c}_2) + 4(\mu + \tilde{\mu}),
(c_1 + \tilde{c}_1) + 2(\mu + \tilde{\mu}), (c_0 + \tilde{c}_0) + 1(\mu + \tilde{\mu})]
\]

The ciphertext vector is still valid for the following two reasons:

1. The first part of the addition \(c_i + \tilde{c}_i\) still holds an encryption of zero. The only difference is that it is noisier compared to a fresh encryption.

2. The second term in each entry is the message scaled by powers of two, i.e. \(2^i \cdot (\mu + \tilde{\mu})\) as in a fresh ciphertext.

**Homomorphic AND.** A homomorphic AND operation between two ciphertext matrices is computed as shown in Figure 3.2. We summarize the derivation of the

\[
C' = Flatten(C \cdot \tilde{C}) = [\vec{c}'_3, \vec{c}'_2, \vec{c}'_1, \vec{c}'_0]^	op = \begin{bmatrix}
\begin{array}{cccc}
  c_{(3,3)} + \mu & c_{(3,2)} & c_{(3,1)} & c_{(3,0)} \\
  c_{(2,3)} & c_{(2,2)} + \mu & c_{(2,1)} & c_{(2,0)} \\
  c_{(1,3)} & c_{(1,2)} & c_{(1,1)} + \mu & c_{(1,0)} \\
  c_{(0,3)} & c_{(0,2)} & c_{(0,1)} & c_{(0,0)} + \mu
\end{array}
\end{bmatrix}
\begin{bmatrix}
\begin{array}{cccc}
  \tilde{c}_{(3,3)} + \tilde{\mu} & \tilde{c}_{(3,2)} & \tilde{c}_{(3,1)} & \tilde{c}_{(3,0)} \\
  \tilde{c}_{(2,3)} & \tilde{c}_{(2,2)} + \tilde{\mu} & \tilde{c}_{(2,1)} & \tilde{c}_{(2,0)} \\
  \tilde{c}_{(1,3)} & \tilde{c}_{(1,2)} & \tilde{c}_{(1,1)} + \tilde{\mu} & \tilde{c}_{(1,0)} \\
  \tilde{c}_{(0,3)} & \tilde{c}_{(0,2)} & \tilde{c}_{(0,1)} & \tilde{c}_{(0,0)} + \tilde{\mu}
\end{array}
\end{bmatrix}
\]

Figure 3.2: Homomorphic AND operation

rows of the product matrix as follows:

**Row 0:** In Table 3.5 we show columns of the vector \(\vec{c}'_0 = \text{BitDecomp}(c'_0(x))\). In the last row of the table we evaluate \(\text{BitDecomp}^{-1}(\vec{c}'_0)\). The last column contains the respective powers of 2 used in \(\text{BitDecomp}^{-1}\) for easy reference.

The final ciphertext in Table 3.5 has the proper ciphertext form. The first part of the ciphertext \(c_{(0,3)} \cdot \tilde{c}_3 + c_{(0,2)} \cdot \tilde{c}_2 + c_{(0,1)} \cdot \tilde{c}_1 + c_{(0,0)} \cdot \tilde{c}_0 + c_0 \cdot \tilde{\mu} + \tilde{c}_0 \cdot \mu\) is still
an encryption of zero since ciphertexts $c_i$ and $\tilde{c}_i$ are encryptions of zero and their multiplication with a binary polynomial will result in zero encryptions as long as noise is contained.

**Row 1:** We apply the same arithmetic for the row number 1 and form a similar construction in Table 3.6. We achieve a similar derivation for second row in Table 3.6.

<table>
<thead>
<tr>
<th>$C^{(0,0)}$</th>
<th>$C^{(0,3)}$</th>
<th>$C^{(0,2)}$</th>
<th>$C^{(0,1)}$</th>
<th>$C^{(0,0)}$</th>
<th>$\tilde{C}$^{(1,3)}</th>
<th>$C^{(1,3)}$</th>
<th>$C^{(0,3)}$</th>
<th>$C^{(1,3)}$</th>
<th>$C^{(1,0)}$</th>
<th>$C^{(0,0)}$</th>
<th>$\tilde{C}$^{(1,0)}</th>
<th>$C^{(1,0)}$</th>
<th>$C^{(0,0)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
<td>$\tilde{c}_0$</td>
</tr>
</tbody>
</table>

Table 3.5: Derivation of Row 0 of product ciphertext

The only difference is that now the message is scaled by 2.

We can now generalize the arithmetic for each row $i$ as follows:

$$c_i' = \sum_{j=0}^{\ell-1} c_{(i,j)} \cdot \tilde{c}_j + c_i \cdot \tilde{\mu} + \tilde{c}_i \cdot \mu + 2^j (\mu \cdot \tilde{\mu}).$$

As shown in Equation 3.5, after a matrix multiplication the ciphertext vector elements contains a (noisier) encryption of zero along with the message scaled by powers of 2, i.e. $\tilde{C} = \{\tilde{C}_{\ell-1} + 2^{\ell-1} \tilde{\mu}, \ldots, \tilde{c}_1 + 2^1 \tilde{\mu}, \tilde{c}_0 + 2^0 \tilde{\mu}\}$ in which $\tilde{\mu} = \mu \cdot \tilde{\mu}$. Hence correctness holds throughout circuit evaluation as long as noise is contained.
3.2.3 Optimizations

In our scheme we are able to perform certain optimizations on the arithmetic operations to reduce computational complexity and the memory footprint. The two main optimizations we perform are:

**Using High Radix Representations.** The flattened ciphertext matrix takes up a large space, $\ell^2 = O(\log(q)^2)$. To mitigate we may use a higher radix representation, i.e. a larger radix $2^\omega$ and group $\omega$ bits for each element of the ciphertext matrix. When we apply $\text{BitDecomp}^{-1}$, we use the powers of the chosen radix $(2^\omega)^i$ instead of powers of $2^i$ to reconstruct the ciphertext. With this approach we drastically reduce the matrix size: $\ell' = \ell / \omega$. Note that the number of zero encryptions of NTRU’ should be equal to $\ell'$ as well. Using the high radix encoding the ciphertext size is reduced by $\omega$ times. In addition, the ciphertext matrix size is reduced by $\omega^2$ which decreases the complexity of matrix multiplication significantly.

**Matrix Multiplication.** A straight schoolbook matrix multiplication takes $O(\ell^3)$ time and it may reduced to $O(\ell^{2.374})$ time using Coppersmith-Winograd algorithm. However, to evaluate deep circuits we will need a large modulus and even with the Coppersmith-Winograd algorithm multiplication will be slow. Therefore, we change matrix multiplication into a matrix-vector multiplication as follows:

$$\text{BitDecomp}^{-1} \left( \text{BitDecomp}(\vec{c}) \cdot \text{BitDecomp}(\vec{\tilde{c}}) \right) = \text{BitDecomp}(\vec{c}) \cdot \vec{\tilde{c}}.$$  

A schoolbook matrix-vector multiplication has $O(\ell^2)$ runtime which is faster than a matrix multiplication. The only downside of the algorithm is that a binary by a high radix polynomial multiplication should be implemented instead of a binary by binary polynomial multiplication. Although binary by high radix polynomial...
multiplication is slower, the time gap is closed in the higher level while computing the matrix-vector product for large $\ell$ values.

### 3.2.4 Security Analysis

Our scheme uses the NTRU' encryption scheme adopted from [20]. In F-NTRU, an attacker has access only to a ciphertext vector which holds NTRU' ciphertexts built with fresh encryptions of zero. Therefore as long as NTRU' ciphertexts are secure, our scheme is also secure. According to the Stehlé and Steinfeld [20], their scheme is IND-CPA secure under the hardness of R-LWE assumption as long as the error has a wide distribution $\mathbb{D}_{\mathbb{Z}^n, \sigma}$, i.e. $\sigma > \sqrt{q} \cdot \text{poly}(n)$. However, in the DHS and YASHE' schemes such a high noise instantiations are not possible due to the noise growth in homomorphic multiplications. Therefore, both schemes use narrow error distributions and additionally assume hardness of the Decisional Small Polynomial Ratio (DSPR) Problem along with the R-LWE assumption.

Fortunately, our scheme has a better control on noise management. The (size of the) noise increases linearly with the multiplicative depth. Therefore, we are able to use a wide error distribution and achieve IND-CPA security as in the original scheme by Stehlé and Steinfeld in [20]. Thus the standard deviation $\sigma_{\text{key}}$ of the discrete Gaussian distribution $\mathbb{D}_{\mathbb{Z}^n, \sigma_{\text{key}}}$ needs to be set as:

$$
\sigma_{\text{key}} > 2n \sqrt{\log (8nq)} \cdot q^{1/2+\epsilon}
$$

for $\epsilon > 0$, i.e. $\epsilon = 2^{-128}$. In this setting we are able to generate a public key $h = g/f$, i.e. $g \in \chi_{\text{key}}$ and $f' \in \chi_{\text{key}}$ ($f = 2f + 1$) in which $\chi_{\text{key}}$ is truncated $\mathbb{D}_{\mathbb{Z}^n, \sigma_{\text{key}}}$, that is indistinguishable from a uniformly random distribution.

For R-LWE security we follow the settings in [16]. The noise parameters $s$
and $e$ are sampled from the distribution $\chi_{err}$, a truncated Gaussian distribution $D_{\mathbb{Z}_n, \sigma_{err}}$. The standard deviation $\sigma_{err}$ of the error distribution has the following bound $\sigma_{err} > \sqrt{n \log(n)}$. With this bound, we are able to add noise to our public key and keep it computationally indistinguishable from random uniform distribution.

As above mentioned, the distributions $\chi_{key}$ and $\chi_{err}$ are truncated Gaussian distributions. They are also defined as $B$-bounded distributions which means the samples are selected from $[-B, B]$. The bound value $B$ is selected as a function of the advantage $\epsilon$ in a distinguishing attack, e.g. $\epsilon = 2^{-128}$. The function, e.g. see [41], gives for a sample $x \in \chi$ the probability of $x$ being larger than $k \cdot \sigma$ for a factor $k$ is:

$$\text{Prob}_{x \leftarrow D_{\mathbb{Z}_n, \sigma}}[|x| > k \cdot \sigma] = \text{erf}(k/\sqrt{2}).$$

Using the equation above, we select the factor $k$ as $k(\epsilon) = \min\{k \mid \text{erf}(k/\sqrt{2}) < \epsilon\}$, and select the $B$-bound as $B = k(\epsilon) \cdot \sigma$.

**Hermite Factor.** The existing attacks rely on the lattice reduction algorithms and the best attack known in practice is BKZ 2.0 [39] by Chen and Nguyen. The quality of the security is measured by Hermite factor $\delta$. A recent work by van de Pol and Smart [38] shows that a fixed Hermite factor for all the lattice dimensions is not true. They show that it is possible to reduce the lattice dimension by selecting a larger Hermite factor for the same security level by utilizing the work of Chen and Nguyen in [40]. Later, a similar approach is followed by Lepoint and Naehrig [1]. They used a quadratic function to interpolate the enumeration costs in [40] and compute the Hermite factors for a wide range of lattice dimensions. This results in Hermite factors that are more precise for the required security levels. Furthermore,
they also derive the following condition on $q$

$$\log(q) \leq \min_{n \leq m} \frac{m^2 \log(\delta(m)) + m \log(\sigma/\alpha)}{m - n}$$

where $\alpha = \sqrt{-\log(\epsilon)/\pi}$. Using this equation and the Hermite factor values in [1] we estimate the required $n$ and $q$ pairs for the security levels $\lambda = \{80, 128\}$ in Section 3.2.9.

### 3.2.5 Noise Analysis

In this section we analyze the noise performance of our scheme with homomorphic evaluations. In case of a homomorphic addition, the noise increases only a small percent compared to homomorphic multiplication. In our analysis we want to determine the depth of the circuit we can evaluate and still decrypt correctly with growing noise. This analysis includes average case and worst case scenarios for homomorphic multiplication that estimates the number of possible multiplicative levels. Since additions contribute minimally to noise growth we focus only on homomorphic multiplications.

For an element $a \in \mathbb{R}$ we define the Euclidean norm $||a|| = \sqrt{\sum a_i^2}$ and the infinity norm $||a||_{\infty} = \max|a_i|$ for all possible values of $i$. In multiplication, we can bound the noise growth with the aid of the following Lemma.

**Lemma 3.2.1 ([42, 16]).** In a ring $R = \mathbb{Z}[x]/(x^n + 1)$, for any two polynomials $a, b \in R$ we have the following norms $||ab|| \leq \sqrt{n}||a|| \cdot ||b||$ and $||ab||_{\infty} \leq n||a||_{\infty} \cdot ||b||_{\infty}$.

**Recursive Evaluation.** We assume the evaluation circuit is arranged into a tree with levels of parallel multiplication gates that accept as input the output ciphertexts from the previous level. Lets denote a ciphertext matrix that is at a certain
multiplicative level $i$ as $C^{(i)}$. Then, a ciphertext matrix at a multiplicative level $C^{(i)} = C^{(i-1)} \cdot \tilde{C}^{(i-1)}$. Lets recall that these ciphertext matrices are actually NTRU' ciphertext vectors with BitDecomp applied. We also denote the ciphertexts in the vector for multiplicative level $i$ and row index $j$ as $c_j^{(i)}$.

In the first multiplicative level, i.e. $i = 0$, ciphertext vectors are fresh encryptions with security parameters explained in Section 3.2.4. Basically, we have samples $g, f' \in \chi_{\text{key}}$ and $s, e \in \chi_{\text{err}}$ and ciphertexts $c_j^{(0)} = hs_j + 2e_j + 2^j\mu$ where $\mu$ is the message, $f = 2f' + 1$ and $h = 2gf^{-1}$ is public key. Lets recall that for ciphertext matrix multiplications we have NTRU' ciphertexts that hold the form given in Equation 3.5. The equation can be rewritten with level index to show the result of a multiplicative level as:

$$c_j^{(i)} = \sum_{k=0}^{\ell-1} c_{j,k} \cdot \tilde{C}_k^{(i-1)} + c_j^{(i-1)} \cdot \tilde{\mu} + \tilde{c}_j^{(i-1)} \cdot \mu + 2^j(\mu \cdot \tilde{\mu}).$$

We can simplify the equation, for noise evaluation, by replacing radix size polynomials $c_{j,k}$ with $y_\tau$, choosing ciphertext vector index $j = 0$, and substituting $y(i)$ in place of all ciphertexts since they have the same noise level for the same multiplicative level $i$:

$$y(i) = y(i-1) f y_\ell + y(i-1) \tilde{\mu} + y(i-1) \mu + \mu \tilde{\mu}.$$ 

To be able to decrypt $y(i)$ correctly, we need $\|y(i)f\|_\infty < q/2$. Thus, we need

$$\|y(i)f\|_\infty \leq \|y(i-1)f y_\ell\|_\infty + \|y(i-1)f \tilde{\mu}\|_\infty + \|y(i-1)f \mu\|_\infty + \|\mu \tilde{\mu}f\|_\infty.$$ 

Later, we expand $\|y(i-1)f\|_\infty$ in terms of $\|y(i-2)f\|_\infty$ and continue the process re-
cursively. At the lowest level we have:

\[ ||y(0)f||_\infty = ||c^{(0)}f||_\infty \leq ||2gf^{(-1)}sf||_\infty + ||2ef||_\infty. \]

Since \( ||g||_\infty = ||f'||_\infty = B_{\text{key}}, ||s||_\infty = ||e||_\infty = B_{\text{err}}, \) the worst-case noise is equal to

\[ ||y(0)f||_\infty \leq 2nB_{\text{key}}B_{\text{err}} + 2nB_{\text{err}}(2B_{\text{key}} + 1) \]
\[ \leq 2nB_{\text{err}}(3B_{\text{key}} + 1). \]

For an arbitrary level \( i \), the noise can be evaluated recursively by setting \( B_i = ||y(i)f||_\infty, ||y_r||_\infty = 2^\omega - 1 \) and \( ||\mu||_\infty \leq 1. \) The message at level \( i \) is bounded as \( ||\mu_i||_\infty \leq n^{2^i-1}. \) Then, the noise evaluation is evaluated as

\[
\begin{align*}
  ||y(i)f||_\infty &\leq \underbrace{||y(i-1)f y_\ell||_\infty + ||y(i-1)f \tilde{\mu}_i||_\infty}_{B_i} \\
  &\leq \ell n(2^\omega - 1)B_{(i-1)} + n^{2^i} B_{(i-1)} \\
  &\quad + \underbrace{||y(i-1)f \tilde{\mu}_i||_\infty + ||\mu_i \tilde{\mu}_i f||_\infty}_{2^{i+1}(2B_{\text{key}} + 1)} \\
  &\leq \ell n(2^\omega - 1)B_{(i-1)} + 2n^{2^i} B_{(i-1)} + n^{2^{i+1}} (2B_{\text{key}} + 1)
\end{align*}
\]

which is also summarized as

\[ B_i \leq \ell n(2^\omega - 1)B_{(i-1)} + 2n^{2^i} B_{(i-1)} + n^{2^{i+1}} (2B_{\text{key}} + 1) \]

In the average case the noise accumulation will be much slower than what the bound given above predicts due to the Gaussian distribution. Using the worst case noise bound, we can simply obtain the average case noise by substituting \( \sqrt{n} \) and \( \sqrt{\ell} \) for \( n \) and \( \ell \), respectively,
Taking Advantage of Noise Asymmetry. As derived earlier the ciphertext $c'$ obtained from the homomorphic product of ciphertexts $c$ and $\tilde{c}$ can be expressed as

$$c'_j = \sum_{k=0}^{\ell-1} c(j,k) \cdot \tilde{c}_k + c_j \cdot \tilde{\mu} + \tilde{c}_j \cdot \mu + 2^j (\mu \cdot \tilde{\mu}).$$

It is important to note that roles of the input ciphertexts in the summation are not symmetric. The (left) input $c$ is processed as binary polynomials breaking the structure of the ciphertext, while the (right) input $\tilde{c}$ is kept intact. Therefore the noise content in $c$ becomes irrelevant in the summation. Since we have the freedom to switch the inputs, we may take advantage of this fact by by placing the noisier ciphertext to the left during our evaluations. In the extreme case we can always feed fresh ciphertexts from the right input. If we iterate $i$ multiplications with fresh ciphertexts $\tilde{c}_i$ we obtain

$$y_1 = \tilde{y}_0 y_{\ell} + \tilde{y}_0 \mu_0 + y_0 \tilde{\mu}_0 + \mu_0 \tilde{\mu}_0$$
$$y_2 = \tilde{y}_1 y_{\ell} + \tilde{y}_1 \mu_1 + y_1 \tilde{\mu}_1 + \mu_1 \tilde{\mu}_1$$
$$\vdots$$
$$y_i = \tilde{y}_{i-1} y_{\ell} + \tilde{y}_{i-1} \mu_i + y_{i-1} \tilde{\mu}_{i-1} + \mu_i \tilde{\mu}_{i-1}$$

The noise experienced in decrypting after iteration $i$ will be

$$\|f y_i\|_\infty \leq \|f \tilde{y}_{i-1} y_{\ell}\|_\infty + \|f y_{i-1}\|_\infty + \|f y_{i-1} \tilde{\mu}_{i-1}\|_\infty + \|f \mu_i \tilde{\mu}_{i-1}\|_\infty$$
$$\leq \|(2\tilde{g} \tilde{s} + 2\tilde{e} f) y_{\ell}\|_\infty + \|(2\tilde{g} \tilde{s} + 2\tilde{e} f)\mu_i\|_\infty + \|f y_{i-1} \tilde{\mu}_{i-1}\|_\infty + \|f \mu_i \tilde{\mu}_{i-1}\|_\infty$$

64
Basically, we have samples $\tilde{g}, f' \in \chi_{\text{key}}$ and $\tilde{s}, \tilde{e} \in \chi_{\text{err}}$. For an arbitrary iteration $i$, the noise can be evaluated by setting $B_i = ||y_if||_\infty$, $||y_\tau||_\infty = 2^w - 1$, $||\mu_0||_\infty \leq 1$ and for the messages $||\tilde{\mu}_i||_\infty \leq 1$, and $\mu_i = \mu_{i-1}\tilde{\mu}_{i-1}$, $||\mu_i||_\infty \leq n||\mu_{i-1}||_\infty$ and with $\tilde{\mu}_0 \leq 1$ we have $||\mu_i||_\infty \leq n^i$. We explicitly derive the noise bound as

$$B_i = ||fy_i||_\infty \leq [2n^2B_{\text{key}}B_{\text{err}}(2^w - 1)\ell$$

$$+ 2n^2B_{\text{err}}(2B_{\text{key}} + 1)(2^w - 1)\ell]$$

$$+ [2n^{i+2}B_{\text{err}}B_{\text{key}} + 2n^{i+2}B_{\text{err}}(2B_{\text{key}} + 1)]$$

$$+ [nB_{i-1}] + [n^{i+2}(2B_{\text{key}} + 1)]$$

Here for $B_0 = ||fy_0||_\infty = ||2gs + 2ef||_\infty \leq 5nB_{\text{key}}B_{\text{err}} + 2nB_{\text{err}}$. In the average case the noise growth can be captured by

$$B_{i,\text{avg}} \leq [2nB_{\text{key}}B_{\text{err}}(2^w - 1)\sqrt{\ell}$$

$$+ 2nB_{\text{err}}(2B_{\text{key}} + 1)(2^w - 1)\sqrt{\ell}]$$

$$+ [2\sqrt{n^{i+2}}B_{\text{err}}B_{\text{key}} + 2\sqrt{n^{i+2}}B_{\text{err}}(2B_{\text{key}} + 1)]$$

$$+ [\sqrt{n}B_{i-1}] + [\sqrt{n^{i+2}}(2B_{\text{key}} + 1)]$$

**Improving Scalability with Single Bit Encryption.** The analysis above shows that the noise growth scales with $O(n)$ over the levels of encryption. In practice, during evaluations the noise term increases with the first term setting the noise floor, i.e. $O(n^2B_{\text{key}}B_{\text{err}}\ell)$ and then with each additional level contributing by a factor of $n$ (in the worst case). Therefore the number of levels supported is heavily determined by the dimension $n$.

The factor $n$ is due to message polynomials we are encrypting. Encrypting polynomials enables batching and yields better amortized times. However, if we are
willing to trade off batching, and instead encrypt bits we may improve scalability, significantly. We may make up for the loss of batching by employing much smaller parameter sizes. If we assume all messages \( \mu, \tilde{\mu} \in \{0, 1\} \), then the noise bound \( B_i = ||fy_i||_\infty \) simplifies to

\[
B_i \leq [2n^2B_{\text{key}}B_{\text{err}}(2^w - 1)\ell + 2n^2B_{\text{err}}(2B_{\text{key}} + 1)(2^w - 1)\ell] + [2nB_{\text{err}}B_{\text{key}} + 2nB_{\text{err}}(2B_{\text{key}} + 1)] + [B_{i-1}] + [(2B_{\text{key}} + 1)]
\]

Supporting Large Integer Messages. Our scheme is capable of supporting a large message space using the approach in [43]. The authors facilitate an integer to polynomial (and reverse) mapping by changing the message modulus from \( p = 2 \) into a polynomial \( p(x) = x - 2 \) in the encryption, decryption and key generation stages. Then we may simply encode the message bits into the message polynomial by distributing each message bit into the coefficients of a polynomial. For instance, a bit-decomposed message \( m = \sum_i m_i2^i \) is encoded as \( m(x) = \sum_i m_ix^i \). With such an encoding we are able to support a very large integer message space, i.e. \( m \in [-2^n, 2^n] \).

After decryption, a message can be evaluated by simply applying \( x = 2 \) into the message polynomial. Since the input message is binary polynomial, the noise analysis is similar to the case that takes advantage of the noise asymmetry. In that case our binary polynomials have degree \( n \), whereas now our message polynomials start with a small degree \( n'\) and it grows with each multiplication. In the one-sided multiplication case, each multiplication increases the degree by \( n' \). This gives us a

\[3\text{This also means that the input message is } n'-\text{bits.}\]
more controllable noise compared to the message polynomials with degree $n$. By modifying the equation of noise asymmetry with $n'$, we obtain

$$B_i = ||fy_i||_\infty \leq [4n^2B_{\text{key}}B_{\text{err}}(2^w - 1)\ell
+ 4n^2B_{\text{err}}(4B_{\text{key}} + 1)(2^w - 1)\ell]
+ [4nB_{\text{err}}(i + 1)n^{i+1}(5B_{\text{key}} + 1)]
+ [n'B_{i-1}] + [(i + 1)n^{i+2}(4B_{\text{key}} + 1)]$$

### 3.2.6 The F-NTRU' Variant

The F-NTRU construction manages to eliminate evaluation keys and relinearizations and the DSPR assumption. However, the ciphertext grows due to the matrix based construction compared to a standard NTRU based SHE constructions. The growth in the ciphertext size not only increases the memory footprint, but also increases the latency of homomorphic evaluations. With this motive we introduce a variant we call F-NTRU' that uses less noise, by sampling the secret key $f'$ and $g$ from a slightly narrower distribution, i.e. $O(poly(n)\cdot q^{1/3})$ instead of $O(poly(n)\cdot q^{1/2})$. From Section 3.2.4 remember that for F-NTRU we require $\sigma_{\text{key}} > 2n\sqrt{\log (8nq) \cdot q^{1/2+\epsilon}}$. For a discussion on the effect of noise distribution on the security of NTRU based schemes, and possible attacks enabled by the use of very narrow key distributions see [22].

Under this modification we can select smaller parameters. The security and noise analysis presented above still applies with the exception of the re-introduced DSPR assumption. Therefore we can still use the noise bounds and parameter derivations by simply setting the noise to the appropriate values, i.e. for $f'$ and $g$ we use $\sigma_{\text{key}} > 2n\sqrt{\log (8nq) \cdot q^{1/3+\epsilon}}$ and for $e$ and $s$ we use $\sigma_{\text{err}} = \sqrt{n\log(n)}$. 

67
3.2.7 Circuit Evaluation

To take advantage of the scalability gained with single bit encryption, we need to maintain the message values in the ciphertexts always as 0 or 1 not only in fresh ciphertexts, but also in ciphertexts obtained through homomorphic evaluation. In [21] this is achieved by restricting the circuit to only (universal) NAND gates. For input ciphertexts $A$ and $B$, we can explicitly express the gate operations as follows

- **NOT:** $C = I_N - A$
- **AND:** $C = A \cdot B$
- **NAND:** $C = I_N - A \cdot B$
- **XOR:** $C = (I_N - A) \cdot B + A \cdot (I_N - B) = A + B - 2A \cdot B$
- **OR:** $C = I_N - ((I_N - A) \cdot (I_N - B)) = A + B - A \cdot B$

The evaluation process requires costly matrix multiplications. However note that for decryption we only need the first cipherext vector element to recover the result. Thus, during circuit evaluation we multiply all the elements of the ciphertext vector with the first element of the left operand. The remaining elements in the ciphertext vector are simply discarded. With this approach we achieve $\ell$ times speedup in circuit evaluations. This is achieved by simply evaluating the boolean circuit from the left-hand side by only using the first element of the ciphertext vector. The fresh ciphertext is always given as input from the right and the accumulated ciphertext is kept on the left. In multiplications, bit decomposition is only applied to the first element and its sum of product is computed with the ciphertext vector on the right-hand side. For instance, we may compute $C' = C \cdot \tilde{C}$ as follows

- $C = [c_{\ell-1}, c_{\ell-2}, \ldots, c_0]$ and $\tilde{C} = [\tilde{c}_{\ell-1}, \tilde{c}_{\ell-2}, \ldots, \tilde{c}_0]$
• Discard the unused ciphertexts: $C = [0,0,\ldots,0,c_0]$

• Take the bit decomposition of $C$: $\{c_{(0,\ell-1)}c_{(0,\ell-2)},\ldots,c_{(0,0)}\}$.

• Compute the sum of product: $C' = \sum_{i=0}^{\ell-1} c_{(0,i)} \cdot \tilde{c}_i$

Using this method, we are able to complete the circuit evaluation in $\ell$ operations rather than $\ell^2$.

3.2.8 Complexity

In Table 3.7 we summarize the asymptotic complexities of F-NTRU and YASHE schemes. We use $\ell = (\log q)/\omega$ for radix size $2^\omega$. Note that our scheme does not require evaluation keys, but the ciphertexts consist of $\ell$ polynomials. After circuit evaluations the $\ell$ polynomials are reduced to a single polynomial ciphertext. On the other hand YASHE requires $\ell^3$ polynomials for evaluation keys. For homomorphic AND evaluation our scheme requires $\ell^2$ polynomial multiplications or using the one sided AND evaluation it uses $\ell$ polynomial multiplications. In case of YASHE, the scheme uses $\ell^2$ polynomial multiplications followed by a costly key switching operation computed via $\ell^3$ polynomial multiplications.

<table>
<thead>
<tr>
<th></th>
<th>F-NTRU/F-NTRU'</th>
<th>YASHE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eval. Key Size</td>
<td>-</td>
<td>$O(\ell^3 n \log q)$</td>
</tr>
<tr>
<td>Ciphertext Size</td>
<td>$O(\ell n \log q)$</td>
<td>$O(n \log q)$</td>
</tr>
<tr>
<td>Final Ciphertext Size</td>
<td>$O(n \log q)$</td>
<td>$O(n \log q)$</td>
</tr>
<tr>
<td>AND Eval.</td>
<td>$O(\ell^2)$</td>
<td>$O(\ell^2)$</td>
</tr>
<tr>
<td>One Sided AND Eval.</td>
<td>$O(\ell^2)$</td>
<td>$O(\ell)$</td>
</tr>
<tr>
<td>Key-Switching</td>
<td>-</td>
<td>$O(\ell^3)$</td>
</tr>
</tbody>
</table>

Table 3.7: Comparison of F-NTRU and YASHE: homomorphic AND evaluation and Key Switching complexities are in terms of polynomial multiplications with $\ell = (\log q)/\omega$. 

69
3.2.9 Parameter Selection

In our choice of parameters we aimed for 80-bit and 128-bit security levels. For comparison, we also tabulated parameter choices for selected depths for F-NTRU, F-NTRU’ and YASHE as shown in Table 3.8. These parameters are for the single bit encryption case and for an implementation that takes advantage of the noise asymmetry.

<table>
<thead>
<tr>
<th>$L$</th>
<th>F-NTRU</th>
<th>F-NTRU</th>
<th>F-NTRU’</th>
<th>YASHE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\lambda \geq 80$</td>
<td>$\lambda \geq 128$</td>
<td>$\lambda \geq 128$</td>
<td>$\lambda \geq 128$</td>
</tr>
<tr>
<td>5</td>
<td>(12,142)</td>
<td>(12,142)</td>
<td>(12,105)</td>
<td>(11,359)</td>
</tr>
<tr>
<td>10</td>
<td>(12,152)</td>
<td>(13,157)</td>
<td>(12,113)</td>
<td>(13,840)</td>
</tr>
<tr>
<td>20</td>
<td>(12,172)</td>
<td>(13,177)</td>
<td>(12,128)</td>
<td>(14,1705)</td>
</tr>
<tr>
<td>30</td>
<td>(12,192)</td>
<td>(13,198)</td>
<td>(12,144)</td>
<td>(14,2538)</td>
</tr>
</tbody>
</table>

Table 3.8: Parameters ($\log(n)$, $\log(q)$) to support depth $L$ evaluations with $\omega = 16$. The distinguishing advantage is set to $2^{-128}$.

In Table 3.9 and Table 3.10, we summarize the maximum possible number of multiplications, i.e. $2^L$, for 80-bit and 128-bit security level. Note that it is advantageous to select large radix sizes $\omega$ for faster evaluation. This decreases the run-time significantly, however, the depth is reduced somewhat since the noise is increased.

As for ciphertext sizes, we are able to decrease it by a factor of $\omega$.

<table>
<thead>
<tr>
<th>$(N, \log q)$</th>
<th>$\omega = 16$</th>
<th>$\omega = 8$</th>
<th>$\omega = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Worst</td>
<td>Average</td>
<td>Worst</td>
</tr>
<tr>
<td>(4096, 193)</td>
<td>14</td>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td>(8192, 388)</td>
<td>106</td>
<td>124</td>
<td>114</td>
</tr>
<tr>
<td>(16384, 787)</td>
<td>301</td>
<td>319</td>
<td>309</td>
</tr>
</tbody>
</table>

Table 3.9: Entries give the equivalent multiplicative depth $L$, i.e. parameters support $2^L$ multiplications. The security parameter $\lambda$ is chosen to support 80-bit security.
\( \omega = 16 \) \( \omega = 8 \) \( \omega = 4 \)

\[ \begin{array}{c|cc|cc|cc|cc|cc|cc} 
(N, \log q) & \text{Worst} & \text{Average} & \text{Worst} & \text{Average} & \text{Worst} & \text{Average} \\
\hline
(4096, 150) & 0 & 8 & 1 & 16 & 5 & 21 \\
(8192, 290) & 58 & 75 & 66 & 83 & 70 & 87 \\
(16384, 597) & 206 & 225 & 214 & 233 & 218 & 237 \\
\end{array} \]

Table 3.10: Entries give the equivalent multiplicative depth \( L \), i.e. parameters support \( 2^L \) multiplications. The security parameter \( \lambda \) is chosen to support 128-bit security.

### 3.2.10 Implementation Results

We implemented the F-NTRU and F-NTRU' schemes using Shoup’s NTL library version 9.6.4 [37] compiled with the GMP 6.1 package. We ran our experiments on a 125 GBs of RAM and Intel Xeon E5-2637v2 64-bit CPU server clocked at 3.5 Ghz. The timing results for homomorphic multiplication are summarized in Table 5.3 for the parameters choices in Table 3.8. Although the algorithm is suitable for parallelization, NTL’s threading capabilities are limited. Therefore, when we use 4 threads, i.e. \( C=4 \), we only achieve \( \sim 2 \) times speedup. Furthermore, it is clear from the table that using 8 threads does not change the timings significantly.

\[ \begin{array}{cccccc}
L & \text{F-NTRU} & \lambda \geq 80 & \text{F-NTRU} & \lambda \geq 128 & \text{F-NTRU}' & \lambda \geq 128 \\
& C=1 & C=4 & C=8 & C=1 & C=4 & C=8 & C=1 & C=4 & C=8 \\
5 & 43.5 & 25.1 & 24.4 & 43.5 & 25.1 & 24.4 & 33.0 & 18.9 & 18.1 \\
10 & 53.3 & 29.8 & 30.8 & 110.7 & 74.2 & 60.7 & 37.2 & 26.0 & 19.7 \\
20 & 60.0 & 32.0 & 31.2 & 133.4 & 68.1 & 72.5 & 37.9 & 25.4 & 19.9 \\
30 & 65.5 & 34.3 & 36.4 & 145.9 & 92.5 & 76.0 & 42.6 & 25.0 & 24.4 \\
\end{array} \]

Table 3.11: Homomorphic multiplication times (msec) for radix selection \( \omega = 16 \). \( C \) denotes the number of threads.

The main advantage of the proposed F-NTRU and F-NTRU' schemes is that it eliminates costly evaluation keys and the slow relinearization operations. In addition the homomorphic evaluation is immensely simplified as we no longer care about
keeping track of the evaluation levels per ciphertext. Finally, we summarize the evaluation key and ciphertext sizes in Table 3.12.

<table>
<thead>
<tr>
<th>Evaluation Key</th>
<th>Ciphertext</th>
</tr>
</thead>
<tbody>
<tr>
<td>YASHE</td>
<td>F-NTRU</td>
</tr>
<tr>
<td>( \lambda \geq 80 )</td>
<td>( \lambda \geq 80 )</td>
</tr>
<tr>
<td>( \omega = 2 )</td>
<td>( \omega = 16 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( L )</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.86 TB</td>
<td>478 TB</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>639 KB</td>
<td>760 KB</td>
<td>946 KB</td>
<td>1152 KB</td>
</tr>
<tr>
<td></td>
<td>639 KB</td>
<td>1570 KB</td>
<td>2124 KB</td>
<td>2574 KB</td>
</tr>
<tr>
<td></td>
<td>368 KB</td>
<td>452 KB</td>
<td>512 KB</td>
<td>648 KB</td>
</tr>
<tr>
<td></td>
<td>87 KB</td>
<td>820 KB</td>
<td>3.3 MB</td>
<td>4.9 MB</td>
</tr>
</tbody>
</table>

Table 3.12: Evaluation key and ciphertext sizes to support depth \( L \) evaluation, i.e. \( 2^L \) multiplications. YASHE key sizes are computed by using the formulation in [2] which is for the worst case scenario.
Chapter 4

Software Implementations of FHE Schemes

In this chapter, we give three software applications using our FHE constructions. First, in Section 4.1 we implemented AES homomorphically and compare it with other existing homomorphic AES implementations. Later, in Section 4.2 we introduce a more practical block cipher named Prince and compute it homomorphically. Last, in Section 4.3 we introduce a private information retrieval (PIR) implementation using our FHE constructions.

4.1 Homomorphic AES Implementation

Here we briefly summarize the AES circuit we use during evaluation. The homomorphic evaluation function takes as input the encrypted AES evaluation keys, and the description of the AES circuit as input. All input bits are individually encrypted into separate ciphertexts. We do not use byte-slicing in our implementation. Our description follows the standard definition of AES with 128-bit keys where each of
the 10 rounds are divided into four steps: \texttt{AddRoundKey}, \texttt{ShiftRows}, \texttt{MixColumns} and \texttt{SubBytes}:

4.1.0.1 \texttt{AddRoundKey}

The round keys are derived from the key through an expansion algorithm and encrypted to be given alongside the message beforehand. The first round key is added right after the computation starts and the remaining round keys are added at the end of each of their respective rounds during evaluation. Therefore, each round key is prepared for the level during which it will be used. As we will shortly show each AES level requires 4 multiplication levels. Therefore the round key for level $i$ is computed in $R_{q_{4i}}$ for $0 \leq i \leq 10$. Adding a round key is a simple XOR operation performed by addition of the ciphertexts. Since round keys are fresh ciphertexts, the added noise is limited to a single bit.

4.1.0.2 \texttt{ShiftRows}

The shifting of rows is a simple operation that only requires swapping of indices trivially handled in the code. This operation has no effect on the noise.

4.1.0.3 \texttt{MixColumns}

The Mix Column operation is a $4 \times 4$ matrix multiplication with constant terms in $GF(2^8)$. The multiplication is between a byte and one of the constant terms of \{ $x + 1$, $x$, 1 \} with modulo ($x^8 + x^4 + x^3 + x + 1$). These products are evaluated by
simple additions and shifts as follows.

\[
\begin{align*}
(b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) & \times 1 \to (b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) \\
(b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) & \times x \to (b_6 b_5 b_4 b_3 b_2 b_1 b_0 b_7) \oplus (000 b_7 0 b_7 0) \\
(b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) & \times (x+1) \to (b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) \oplus (b_6 b_5 b_4 b_3 b_2 b_1 b_0 b_7) \oplus (000 b_7 0 b_7 0)
\end{align*}
\]

Once the multiplication of the rows are finished, 4 values are added to each other. The addition operations add a few bits of noise.

4.1.0.4 SubBytes

The SubBytes step or the S-Box is the only place where we require homomorphic multiplications and Relinearization operations. An S-Box lookup in AES corresponds to a finite field inverse computation followed by the application of an affine transformation; i.e., \( s = M b^{-1} \oplus B \). \( M \) is a \( \{0, 1\} \) matrix and \( B \) is constant vector for the affine transformation which may are simply realized using addition operations between ciphertexts. The time consuming part of the S-Box is the evaluation of inversion operation. In [44], the authors introduced a compact design for computing the inverse. The input byte in \( GF(2^8) \) is converted using an isomorphism into a tower field representation, i.e. \( GF(((2^2)^2)^2) \), which allows much more efficient inversion. This conversion to/from tower field representation is achieved by simply multiplying with a conversion matrix with \( \{0, 1\} \) coefficients. The inversion operation can be written as: \( b^{-1} = X(X^{-1}b)^{-1} \). With this modificaition the operations in the SubBytes step can be expressed as \( s = M(X(X^{-1}b)^{-1}) \oplus B \). The conversion
matrices $X^{-1}$ and the matrix product $MX$ are given as follows.

\[
X^{-1} = \begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 1 & 1
\end{pmatrix}
\]

\[
MX = \begin{pmatrix}
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 0
\end{pmatrix}
\] (4.1)

With tower field representation, the 8-bit S–Box substitution requires 4 (multiplication) levels of circuit evaluation. The full 10 round 128 bit-AES block homomorphic evaluation requires the evaluation of a depth 40 circuit.

### 4.1.1 Implementation Results

In order to implement AES homomorphically, first we implemented DHS-FHE scheme in Section 3.1 with the optimizations summarized in Section 3.1.1 using Shoup’s NTL library version 6.0 [37] compiled with the GMP 5.1.3 package. The implementation batches bits into ciphertexts using CRT applied using a cyclotomic modulus polynomial $\Phi_m(x)$ where $\deg(\Phi) = n$. Also note that we fix $\chi$ with $B = 2$ and $r' = 1.55$. The evaluation functions supports homomorphic additions and multiplication operations. Each multiplication is followed by a Relinearization operation and modulus switching in our implementation. After homomorphic evaluation the results may be recovered from the message slots using remainder computations as usual.
4.1.1.1 Homomorphic AES evaluation

Using the DHS primitives we implemented 40 level AES circuit described above. The AES S-Box evaluation, is completed using 18 Relinearization operations and thus 2,880 Relinearizations are needed for the full AES.

We ran the AES evaluation for two choices of parameters:

- Polynomial degree of \( n = 27000 \) with a modulus of size \( \log(q) = 1230 \) and Hermite factor \( \delta = 1.0078 \) (low security setting). For a error margin of \( \alpha \approx 8 \) and number of additions per AES level of \( \nu \approx 100 \) if we cut \( \log(p) = \log(1/K) = 30 \) bits at each level Equation 3.3 tells us that the noise will stabilize around 12.8 bits. For \( \alpha \approx 8 \) we obtain an error probability of \( 2^{-41} \) per ciphertext. Under these parameters the total running time of AES is 25 hours. Since we batched with 1800 message slots we obtain 50 seconds evaluation time per block.

- Polynomial degree set as \( n = 32768 \) with modulus size \( \log(q) = 1271 \) and Hermite factor \( \delta = 1.0067 \). For a error margin of \( \alpha \approx 8 \) and number of additions of \( \nu \approx 100 \) if we cut \( \log(1/K) = 31 \) bits at each level the noise will stabilize around 12.6 bits. The total running time is 29 hours resulting in 51 seconds per block encryption with 2048 message slots.

We summarize the the parameters for the two settings and the timing results in Table 4.1.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log(q_0) )</th>
<th>( \delta )</th>
<th>( \log(1/K) )</th>
<th>Mess. Slots</th>
<th>Total Time</th>
<th>Time/Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>27000</td>
<td>1230</td>
<td>1.0078</td>
<td>30</td>
<td>1800</td>
<td>25 hours</td>
<td>50 sec</td>
</tr>
<tr>
<td>32768</td>
<td>1271</td>
<td>1.0067</td>
<td>31</td>
<td>2048</td>
<td>29 hours</td>
<td>51 sec</td>
</tr>
</tbody>
</table>

Table 4.1: The two settings under which we evaluated AES and timing results on Intel Xeon @ 2.9 GHz.
### Table 4.2: Sizes of public-key in various representations with and without optimization for the two selected parameter settings.

<table>
<thead>
<tr>
<th>REPRESENTATION</th>
<th>ORIGINAL (GBytes)</th>
<th>OPTIMIZED (GBytes)</th>
<th>AES SPEEDUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polynomial</td>
<td>67</td>
<td>4.75</td>
<td>1</td>
</tr>
<tr>
<td>NTT</td>
<td>172</td>
<td>12.2</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>87</td>
<td>6.13</td>
<td></td>
</tr>
<tr>
<td></td>
<td>184</td>
<td>13.1</td>
<td></td>
</tr>
</tbody>
</table>

When we utilize the two Hermite factor for the security analysis using the formula in [32], i.e. \(1.8 / \log \delta - 110\), we compute 77 and 50 - bits security for \(\delta = 1.0067\) and \(\delta = 1.0078\) respectively. If we use the Hermite factor parameters from the Table 3.4 we have a security level of larger than 128- bits for \(\delta = 1.0067\) and we have security level of between 80 and 128 - bits for \(\delta = 1.0078\).

#### 4.1.1.2 Memory requirements

In the implementation we are taking advantage of the reduced public key size as described in Section 3.1.1. To support a 40 level AES circuit evaluation with the original scheme in [16] for the two settings outlined above we would need to store public keys of size 67 GBytes and 87 GBytes, respectively. The optimized scheme reduces the public keys to 4.75 Gbytes and 6.15 GBytes. This demonstrates the effectiveness of the optimization. We can perform the evaluation on common machines with less than 16 Gbytes memory. Table 4.2 summarizes the public key sizes for the two chosen parameter settings with and without the public key optimization.

Since our server has more memory, to speed up the relinearization operations we keep the public keys in the NTT domain requiring 12.2 Gbytes and 13.1 Gbytes, respectively. Also we keep all the keys for a round (4 levels at a time) in memory. Keeping the public keys in the NTT domain improved the speed of relinearizations by about 3 times. Since relinearizations amount to about 70% of the time we gained an overall speedup of 2.5 times in AES evaluation.
4.1.2 Comparison

In the following, we will briefly compare our implementation with other homomorphic encryption libraries and implementations that have appeared in the literature.

- **GHS-AES**: When compared to the BGV style leveled AES implementation by Gentry, Smart, Halevi (GHS) [23]; our implementation runs 47 times faster than the bit-sliced and 5.8 times faster than the byte-sliced implementation. Our implementation is more comparable to the bit-sliced version since we did not customize our software library to more efficiently evaluate AES in order to keep it generic. While we also use optimizations such as modulus switching, and batching the two implementations differ in the way they handle noise. In the GHS FHE implementation take a more fine grain approach to modulus switching, by cutting the noise even after constant multiplications, additions and shifting operations. Depending on the implementation is bit-sliced or byte-sliced, the number of levels ranges between 50 to 100 where in each level 18-20 bits are cut. In the presented work we only cut the modulus after multiplications and therefore we have a fixed 40 levels with 30-31 bits cut per level.

- **GHS-AES (Updated Implementation)**: In the final revision of this manuscript, Gentry, Smart, Halevi (GHS) [45] published significantly improved runtime results. Compared to the earlier implementation, the authors used the latest version of the HElib library. They managed to decrease the number of levels in the AES circuit but had to increase the number of bits cut in each level to manage the additional noise growth. The new result only reports a SIMD version. Two variations of the implementation are reported: one with bootstrapping and one without bootstrapping. In the bootstrapping version, 180
blocks are processed at a time which achieves a 6 seconds amortized runtime using only 23 circuit levels. In the non-bootstrapping implementation, they process 120 blocks at a time and achieve 2 seconds per block runtime for 40 circuit levels. Also the design only requires around 3.5GB of memory.

- **MS-AES:** Very recently Mella and Susella (MS) revisited the homomorphic AES computation of GHS AES with some optimizations in [46]. They used the homomorphic encryption library HElib [47] that is based on BGV style homomorphic encryption. The authors managed to reduce the number of levels to 4 per AES round to a total of 40 levels for the entire AES circuit evaluation. They implemented two versions of AES; byte-sliced and packed. In byte-sliced version they were able to pack 12 AES evaluations with 16 ciphertexts, whereas in the packed version they use 1 ciphertext and were not able to pack multiple AES evaluations. The byte-sliced implementation has an execution time of 2 hours 47 minutes with an amortized time of 14 minutes. For the packed implementation the total execution time is only 22 minutes. In terms of execution time the MS implementation is the fastest with 22 minutes, and 2 hour 47 minute runtime compared to ours with 29 hours, and to GHS with 36 and 65 hours runtime. However in the amortized case, we are the fastest with 51 second runtime compared to MS with 22 minutes and 14 minutes, and to GHS with 40 minutes and 5 minutes runtimes.

- **YASHE:** In another recent work, Bos et al. introduced a scale-invariant implementation of LTV called YASHE [2]. The authors select a word size $w$ for the homomorphic computations, and create $\ell = \lceil \log_w q \rceil + 2$ vectors to support the radix $w$ operations. The construction has evaluation key that is element of $R^{\ell^3}$ for a polynomial ring $R$. This means that, any increase in the
evaluation depth and size of $q$ will cause cubic growth in the evaluation key size. In order to overcome large growth the authors modified the scheme called YASHE’ to reduce the key size and achieved it to be an element of $R^\ell$. Also they are able to reduce the complexity of the key-switching operation from $\ell^3$ multiplications to $\ell$.

Our variant of LTV is closer to YASHE’ in terms of complexity. The computations and memory requirements grow linearly with the evaluation depth, i.e. $\log q$. Furthermore, as we progress through the levels with modulus switching technique, the homomorphic evaluation accelerates since the operands shrink which is not the case in scale-invariant with fixed run time. The YASHE scheme overcomes the negative effect by selecting a larger word size $w$, e.g. they can set $w = 32$ and achieve a high performance boost. In our case, our scheme is constructed with bit operations. Thus it is harder to eliminate the negative effects of bit size growth. The number of multiplications and evaluation key sizes are given in Table 4.3.

The YASHE authors give implementation results only for the small case. They set the ring $R = \mathbb{Z}[x]/(X^{4096} + 1)$, the prime $q$ as a 127-bit number and the word size $w$ as $2^{32}$. In the implementation key-switching takes 31 ms. Using the same parameters our relinearization takes about 139 ms when $q$ equals to 125-bit number. At the smallest bit size, $q$ is equal to 25-bit number and it takes 20 ms to complete relinearization. For the given case, our scheme seem to be slower with an average of 80 ms run time. Of course this is a single setting and more experiments with various settings should be studied to see how the timings are affected.

- **GPU Implementation of LTV:** Dai et al. recently reported a GPU im-
<table>
<thead>
<tr>
<th># of Multiplications</th>
<th>YASHE</th>
<th>YASHE’</th>
<th>OURS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eval. Key Size</td>
<td>$\ell n \log q$</td>
<td>$\ell^3 n \log q$</td>
<td>$n(\log q)^2$</td>
</tr>
</tbody>
</table>

Table 4.3: Number of multiplications and evaluation key sizes for constructions in [2] and ours.

implememtation of our modified LTV scheme on an NVIDIA GeForce GTX 690 graphics cards in [48]. The authors developed a custom GPU CUDA library to support fast discrete Fourier transform based arithmetic for large degree polynomials. This fast arithmetic library later is used to accelerate the homomorphic multiplication and relinearization operations. The authors achieved a 4.15 hour batched AES evaluation resulting in a 7.3 second per block amortized running time. This implementation achieves a 7.6 times speedup over our CPU implementation.

### 4.2 Homomorphic PRINCE Implementation

In this section we present the homomorphic evaluation of the Prince block cipher. We are motivated by the drastic bandwidth savings that may be achieved by scheme conversion. To unlock this advantage we turn to lightweight ciphers such as Prince. These ciphers were designed from scratch to yield fast and compact implementations on resource-constrained embedded platforms. We show that some of these ciphers have the potential to enable near practical homomorphic evaluation of block ciphers. Our analysis shows that Prince can be implemented using only a 24 level deep circuit. Using an NTRU based implementation we achieve an evaluation time of 3.3 seconds per Prince block one and two orders of magnitude improvement over homomorphic AES implementation.
Several lightweight block ciphers have been proposed with the goal of permitting a compact hardware implementation or good performance at small memory footprint in software. Examples include ciphers like Present, KATAN, TEA, HIGHT, etc. An overview of implementation properties can be found in [49]. Among these, Prince is a lightweight block cipher that has been optimized for low latency and a small hardware footprint [50]. It features a 64-bit block size, 128-bit key size. Prince implements a substitution-permutation network which iterates for 12 rounds. The round function is AES-like and operates on a 4 by 4 array of nibbles, with 4-bit S–boxes, shift rows and mix columns operations. The round key remains constant, but is augmented with a 64-bit round constant to ensure variation between rounds. An interesting feature of Prince is the inflective property: encryption and decryption only differ in the round key, i.e. decryption can use the same implementation as encryption, only the round key needs to be modified. Figure 4.1 shows the structure of the Prince cipher. To implement Prince, the following operations have to be realized:

**Key Schedule** The 128-bit key is split into two parts $k_0$ and $k_1$. $k_0$ is used to generate another key $k'_0 = (k_0 >>> 1) \oplus (k_0 >> 63)$. The keys $k_0$ and $k'_0$ are used as pre- and post-whitening keys, i.e. are XOR-added to the state before
and after all round functions are performed. The round key $k_1$ is the same for all rounds and is also XOR-added during the key addition phase.

**Round Constant Addition** Prince defines different round constants $RC_i$ for each round. A noteworthy property of the round constants is that $RC_i \oplus RC_{11-i} = \alpha$ for $0 \leq i \leq 11$, with $\alpha = \text{c0ac29b7c97c50dd}$. The round constant addition is a binary addition, just as the round key addition. Both operations can be merged.

**S–box** The S–box layer uses a mapping of 4-bit to 4-bit, as defined in the following table. The S–box is the only operation of Prince that is not linear in the bits, and hence needs costly AND operations (or binary multiplication) for its implementation. While other S–boxes are possible for Prince, we chose to use the original S–box, since the maximum depth of multiplication is already optimal for the standard S–box. More details on how we implemented the S–box is given in Section 4.2.1.2.

**Linear Layer** The linear layer consists of two parts: a shift rows which is similar to the shift rows used in AES and simply changes the order of the nibbles. Hence, it is a free operation in a bit-oriented implementation. The mix columns equivalent XOR-adds three input bits to compute one output bit in such a way that the operation is invertible. Again, this operation is linear and easily implementable.

All operations also need an implementation of their inverse, as the last six rounds use the inverse operations.
4.2.1 NTRU based Homomorphic Evaluation

In this section we describe our implementation in detail. Specifically, we first present a study of the depth characteristics of popular lightweight block ciphers among which we identify the Prince cipher as the most promising for homomorphic evaluation. Later we present in detail a shallow circuit implementation of Prince. In what follows, we select optimal parameters for our implementation that is based on DHS FHE scheme.

4.2.1.1 Picking a Lightweight Block Cipher

We are looking for any cipher that provides efficient encryption while permitting a shallow circuit implementation, i.e. the number of consecutive multiplication levels should be minimized. Therefore we turn our attention to lightweight block ciphers [51]. There are two main factors that increase the number of consecutive multiplications: The size and complexity of the S-boxes, as higher non-linearity usually results in higher-degree terms, i.e. an increased number of consecutive binary multiplications. PRESENT [52], for example, has very simple S-boxes, resulting in a shallow circuit for each individual S-box. Another important factor is the number of rounds, where PRESENT is less optimal due to the rather high number of rounds. Prince, a recently proposed block cipher [50], has roughly the same complexity for the S-boxes, but has only 12 rounds which make it a much more efficient choice for our purposes. The more complex linear layer is not a problem, since it does not introduce new binary multiplications. We present an overview of the complexity of different lightweight ciphers in Table 4.4.

Note that the cipher depth is almost fully determined by the consecutive levels of binary AND-statements. The two software-oriented ciphers, namely SEA and HIGHT, feature Feistel-structure and a high number of rounds. The number of
Table 4.4: Comparison of the complexity of common lightweight block ciphers in number of rounds, algebraic degree of the S–box function, algebraic degree of a round excluding the S–box, per round and total number of multiplicative levels.

<table>
<thead>
<tr>
<th>Cipher</th>
<th># Rounds</th>
<th>Algebraic Degree</th>
<th>Total Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>S–box Rem. Round</td>
<td>Per Round</td>
</tr>
<tr>
<td>AES-128 [53]</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Present [52]</td>
<td>31</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Prince [50]</td>
<td>12</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>HIGHT [54]</td>
<td>32</td>
<td>N/A</td>
<td>3</td>
</tr>
<tr>
<td>SEA [55]</td>
<td>93</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>KATAN-64 [56]</td>
<td>254</td>
<td>N/A</td>
<td>1</td>
</tr>
<tr>
<td>Simon-64/96 (64/128) [57]</td>
<td>42 (44)</td>
<td>N/A</td>
<td>1</td>
</tr>
</tbody>
</table>

rounds, together with the Feistel structure, results in a high depth circuit, making them a bad choice for our purposes. Furthermore, additions mod $2^n$ add significant depth due to high nonlinearity for the most significant output bits. While there are [2, 58] FHE implementations capable of evaluating integer operations they do not support mixing of integer and bit-oriented operations as required by most block ciphers. Hence, the hardware-oriented ciphers such as Present and Prince seem more appropriate. Certain possible cipher-specific optimizations are likely missed in the table. Katan, for example, allows the evaluation of a few rounds in parallel, since independent bits are processed in consecutive rounds. We did not explore this further due to the big starting disadvantage in the number of rounds. It can be seen that AES already offers quite a low depth, due to the low number of rounds. In practice, the depth 30 implementation of AES is not attainable since the number of multiplications grows significantly. Instead at best a depth 40 implementation is used in practice 4.1. Either way, the Prince cipher offers a significant improvement over AES.
As described in Section 2.3, Prince can be implemented in a way that every operation is done on a single bit. Consecutive AND operations are costly in the ATV FHE scheme so it is a necessity to prevent them as much as possible. The only part of Prince that is nonlinear is the S-box layer. To determine an optimal representation of the S-box, we use Mathematica to obtain the Algebraic Normal Form (ANF), which represents all equations only in terms of XOR or AND statements. The following table gives the resulting ANF representation of the Prince S-box \( S(A, B, C, D) = (S_0, S_1, S_2, S_3) \). According to the table the S-box requires 28 AND-operations. Further optimization, making use of efficient reuse of intermediate terms, enables a significant reduction of two-input AND operations. The values for \( AB, AC, AD, BC, BD, CD \) can simply be stored and used whenever it is necessary instead of recalculating them every time. There exist four more terms in the formula that can be saved and used again; these values are \( ABD, ABC, ACD, BCD \). To be more efficient, for calculating the first two terms and the next two terms we will use the saved value \( AB \) and \( CD \), respectively. The resulting depth of the multiplication is 2 i.e. one for calculating terms such as \( AB \) and one for calculating terms such as \( ABD \). Hence the total number of ANDs for S-box would be 10—much less than by straight implementation of the ANF. The same procedure is applied to optimize the implementation of the inverse S-box.

<table>
<thead>
<tr>
<th>( S )</th>
<th>( A \oplus C \oplus AB \oplus BC \oplus ABD \oplus ACD \oplus BCD \oplus 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>( A \oplus D \oplus AC \oplus AD \oplus CD \oplus ABC \oplus ACD )</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>( AC \oplus BC \oplus BD \oplus ABC \oplus BCD \oplus 1 )</td>
</tr>
<tr>
<td>( S_2 )</td>
<td>( A \oplus B \oplus AB \oplus AD \oplus BC \oplus CD \oplus BCD \oplus 1 )</td>
</tr>
</tbody>
</table>

### 4.2.1.2 Prince as a Shallow Circuit
4.2.1.3 Parameter Selection

We follow the parameter selection process as in Section 3.1.4 for our homomorphic Prince implementation. In Table 4.5 we summarize the chosen parameters for Prince and AES. Clearly, the 24 levels of Prince give us an advantage over the 40 level AES in selecting smaller parameters: The polynomial degree of Prince is half the size of AES with $n = 16384$. The per level cutting rate is $\log(p) = 20$ bits, better than expected than the noise analysis homomorphic AES. The reason is simple; the Prince S–box has AND operations with three gates, e.g. $A \cdot B \cdot C$, and therefore in the second level two polynomials with different noise levels are multiplied, whereas homomorphic AES assumes the product inputs bear the same level of noise. With

\[ \log(p) = 20, \]

the modulus may be chosen as $\log(q_0) = 500$ which is less than half as long as the AES modulus, i.e. 1271-bits used in homomorphic AES implementation in Section 4.1. With $n = 16384$ and $\log(q_0)$, our Hermite factor is $\delta = 1.0052$. This gives us a 130-bit security level, which actually exceeds the security claims of Prince. The only disadvantage of our Prince evaluation is that we have fewer message slots, exactly half of those of the AES evaluation.

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$\log(q_0)$</th>
<th>$\delta$</th>
<th>Levels</th>
<th>$\log(p)$</th>
<th>Message Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES (Section 4.1)</td>
<td>32768</td>
<td>1271</td>
<td>1.0067</td>
<td>40</td>
<td>31</td>
<td>2048</td>
</tr>
<tr>
<td>Prince</td>
<td>16384</td>
<td>500</td>
<td>1.0052</td>
<td>24</td>
<td>20</td>
<td>1024</td>
</tr>
</tbody>
</table>

4.2.2 Implementation Results

We ran our implementation on a single thread on Intel Core i7 3770K running 3.5 Ghz with 32 GBytes of memory. The most expensive Prince operation is the
evaluation of the S-box circuit, since it is the only operation that contains multiplications and therefore requires **Relinearization**. The S-box is evaluated using 6 **Relinearizations**, resulting in 1,152 **Relinearizations** for the entire evaluation. The execution completes in 57 minutes compared to 31 hours Section 4.1 and 36 hours [23] for AES. This shows about $\times 30$ speedup. A block of Prince encryption takes 3.3 seconds compared to 55 seconds for AES blocks. Another significant advantage of Prince is that at 1 Gbytes the public key is much smaller. Therefore we can run our implementations on standard machines.

Table 4.6: Performance comparison of Prince against AES implementations.

<table>
<thead>
<tr>
<th></th>
<th>Total Time</th>
<th>#Blocks</th>
<th>Per Block</th>
<th>PK Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES (Section 4.1)</td>
<td>31 hours</td>
<td>2048</td>
<td>55</td>
<td>13.1</td>
</tr>
<tr>
<td>AES-Byte Sliced</td>
<td>65 hours</td>
<td>720</td>
<td>300</td>
<td>n/a</td>
</tr>
<tr>
<td>AES-SIMD Sliced</td>
<td>36 hours</td>
<td>54</td>
<td>2400</td>
<td>n/a</td>
</tr>
<tr>
<td>Prince (Ours)</td>
<td>57 minutes</td>
<td>1024</td>
<td>3.3</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### 4.3 PIR Implementation

In this section we present a private information retrieval (PIR) scheme based on a customize NTRU-based SWHE scheme in order to evaluate a specific class of fixed depth circuits relevant for PIR implementation, thus achieving a more practical implementation. Practically, our construction can evaluate a depth 5 circuit is sufficient to construct a PIR capable of retrieving data from a database containing 4 billion rows. We leverage this property in order to produce a more practical PIR scheme. Compared to previous results, our implementation achieves a significantly lower bandwidth cost (more than 1000 times smaller). The computational cost of
our implementation is higher than previous proposals for databases containing a small number of bits in each row.

4.3.1 Homomorphic Encryption Based PIR Schemes

In this section we briefly survey 3 representative cPIR schemes constructed out of homomorphic encryption schemes most relevant to our proposal. We note that this survey is only intended to provide a basis for later comparison.

4.3.1.1 Kushilevitz-Ostrovsky PIR

At the essence of the K-O scheme [59] is the use of a secure homomorphic operations and the idea of conceptually storing the database as a matrix. To elaborate, we can think of Bob as having a database $D$ of size $2^h$ with each location containing a single bit (this can easily be extended for longer strings). Bob then stores $D$ in a matrix $M$ of size $2^{h/2} \times 2^{h/2}$. For any location $i$ in the database $D$, this process can be done by using the first $h/2$ bits of $i$ to represent the number of the row in $M$ and the last $h/2$ bits of $i$ to represent the number of the column in $M$ where $i$ will be placed. Now for Alice to recover the entry $D(i)$, she will take the first $h/2$ bits of $i$ encode them into a one hot encoding $A$ and carry out the same process for the lower $h/2$ bits of $i$ to produce $B$. Finally, Alice uses a partially homomorphic encryption scheme $E$ to encrypt each bit in $A$ and $B$. Thus Alice sends to Bob $(E(A_0) \ldots E(A_{h/2-1}), E(B_0) \ldots E(B_{h/2-1}))$. With this information Bob can now carry out some homomorphic operations between the encrypted bits sent by Alice and the data stored within the matrix $M$ in order to produce an encrypted output which encodes the bit $D(i)$ and can then be sent to Alice for decryption. The matrix is of size $2^{h/2} \times 2^{h/2} = N$ and therefore the communication complexity becomes $O(\sqrt{N})$. 90
4.3.1.2 Boneh-Goh-Nissim (BGN) Scheme

The BGN cryptosystem is a partially homomorphic encryption scheme [60] capable of evaluating 2-DNF expressions in encrypted form. For example, given two encryptions of messages, we can obtain an encryption of the sum of the messages without decryption or compromising privacy. Indeed, the cryptosystem remains semantically secure. Being (in part) based on the Paillier cryptosystem, BGN inherits its additive homomorphic properties. Moreover, with the clever introduction of pairings, BGN is capable of homomorphically evaluating one level of multiplication operations.

The BGN algorithm constructs a homomorphic encryption scheme using finite groups or composite order that support bilinear maps. The construction outlined in [60] uses groups over elliptic curve where homomorphic additions translate into elliptic curve point addition and homomorphic multiplication translates into a pairing operation. Leveraging the single multiplication afforded by pairing operation Boneh, Goh and Nissim also manage to reduce the communication complexity in the basic step of the Kushilevitz-Ostrovsky PIR protocol from $O(\sqrt{N})$ to $O(\sqrt[3]{N})$. In contrast, the computational efficiency of BGN (for the server side PIR computation) scheme lies the pairing operation. Guillevic [61] developed optimized implementations to support BGN which reveals that parings over composite order elliptic curves are far less efficient than parings over prime order curves and also require significantly larger parameter sizes to reach the same security level.

4.3.1.3 Aguilar-Melchor-Gaborit’s Lattice Based PIR

Most of the single server cPIR schemes rely on costly algebraic operations with large operands such as modular multiplications [62, 59, 63], or pairing operations on elliptic curves [60] to realize the homomorphic evaluations. In contrast, the PIR scheme by Aguilar-Melchor and Gaborit [64, 65] makes use of a new lattice
based construction replacing costly modular operations, with much cheaper vector addition operations in lattices. The security is based on the differential hidden lattice problem, which they relate like in many lattice based construction to NP-complete problems in coding theory. Via this connection the scheme is also related to the NTRU scheme [19].

Very briefly, the PIR schemes works as follows. The scheme utilizes a secret random \([N, 2N]\) matrix \(M\) of rank \(N\) over a field \(\mathbb{Z}_p\) which is used to generate a set of different matrices obtained by multiplication by invertible random matrices. These matrices are disturbed by the user by the introduction of noise in half of the matrix columns to obtain softly disturbed matrices (SDMs) and hardly disturbed matrices (HDMs). To retrieve an element from the database the client sends a set of SDMs and one HDM to the PIR server. The PIR server inserts each of its elements in the corresponding matrix with a multiplicative operation \(\text{OP}\) and sums all the rows of the resulting matrices collapsing the PIR server reply to a single noisy vector over \(\mathbb{Z}_p\).

While they proposed full fledged protocol and implementation [64, 65], their analysis was limited to server-side computations on a small database consisting of twelve 3 MByte files. Later Olumofin and Goldberg [66] performed extensive experiments with a broad set databases sizes under realistic network bandwidth settings determining that the lattice based Aguilar-Melchor and Gaborit PIR scheme is one order of magnitude more efficient than a simple PIR.

### 4.3.2 From SWHE to PIR

Consider a database \(D\) with \(|D| = 2^\ell\) rows. Clearly \(\ell\) index bits are sufficient to address all rows. Assume the data bit contained in row \(i\) is denoted by \(D_i\). We may
retrieve an element of $D$ with given index $x \in \{0, 1\}^{\ell}$ which holds $D_x$ by computing:

$$f(x) = \sum_{y \in [2^{\ell}]} (x = y) D_y \pmod{2}, \quad (4.2)$$

where the bitwise comparison $(x = y)$ may be computed as $\prod_{i \in [\ell]} (x_i + y_i + 1)$. Here $[\ell] = \{0, 1, \ldots, \ell - 1\}$ for $\ell > 0$ and $[\ell] = \{\}$ otherwise. The function of the inner loop is to check if each bit of the given $x$ matches the corresponding bit of the $y$ value currently processed. The boolean result is multiplied with the current data value $D_y$ and added to the sum. All of the summed terms except the one where there was a match becomes zero and therefore does not affect the result. Therefore, $f(x) = D_x$.

This arithmetic retrieval formulation allows us to build PIRs and sPIRs. In this case the index value $x$ is in encrypted form. Therefore, the database curator does not know which row is read from the database. We wish the curator to still be able to retrieve and serve the requested row. The data in the row itself can also be in encrypted form in which case the protocol is referred to as a symmetric PIR or sPIR in short. In this setting, if the index $x$ is encrypted using a homomorphic encryption scheme $E$ we may evaluate $f(x)$ homomorphically. From the formulation of $f(x)$ we need $E$ to be able to compute a large number of homomorphic additions (XORs) $O(2^\ell)$ and a small number of multiplications (ANDs) $\ell$ and $\ell + 1$ if the rows are encrypted\(^1\).

### 4.3.3 Picking the SWHE Scheme

To build a PIR for a database of size $2^\ell$ as described in Section 4.3.2 we need an efficient SWHE instantiation that can evaluate a circuit of depth $\lceil \log_2(\ell) \rceil$. In

\(^1\)Note that we restricted the database entries $D_i$ to be bits but a $w$-bit entry can also easily be handled by considering $w$ parallel and independent function evaluations.
practice a depth 5 or 6 circuit will suffice since that will give us an ability to construct a PIR for a database of size $2^{32}$ and $2^{64}$, respectively.

For this we make use of the modified NTRU scheme [19] introduced by Stehlé and Steinfeld [20] with a number of optimizations introduced to this construction by Lopez-Alt, Tromer and Vaikuntanathan [16] to turn Stehlé and Steinfeld’s scheme into a full fledged fully homomorphic encryption scheme. Here we only need to support a few levels and therefore the full Lopez-Alt, Tromer and Vaikuntanathan scheme is not needed. Stehlé and Steinfeld [20] formalized the security setting and reduced the security of their NTRU variant to the ring learning with error (RLWE) problem. More specifically, they show that if the secret polynomials are selected from the discrete Gaussian distribution with rejection then the public key is indistinguishable from a uniform sample. Unfortunately, the reduction does not carry over to the fully homomorphic setting since relaxation of parameters e.g. a larger modulus is needed to accommodate the noise growth during homomorphic evaluation as noted in [16]. We next summarize our instantiation of the scheme in [20] in a way that supports restricted homomorphic evaluations but does not require all the machinery of [16].

We require the ability to sample from a probability distribution $\chi$ on $B$-bounded polynomials in $R_q := \mathbb{Z}_q[x]/(x^n + 1)$ where a polynomial is “$B$-bounded” if all of its coefficients lie in $[-B, B]$. For example, we can sample each coefficient from a discrete Gaussian with mean 0 and discard samples outside the desired range. Each AND gate evaluation incurs significant noise growth and therefore we use the modulus reduction technique introduced by Brakerski, Gentry and Vaikuntanathan [26]. We assume we are computing a leveled circuit with gates alternating between XOR and AND and modulus reduction taking place after each AND level. We use a decreasing sequence of odd prime moduli $q_0 > q_1 > \cdots > q_d$ where $d$ is the depth
of the PIR circuit. In this way, the key (public and evaluation keys) can become quite large and it remains a practical challenge to manage the size of this data and handle it efficiently. For this implementation we specialize the prime moduli $q_i$ by requiring $q_i | q_{i+1}$. This allows us to eliminate the need for key switching and to reduce the public key size significantly. Also in this implementation we do not use relinearizations since we are in a single user setting and we have a shallow well structured circuit (a perfect binary tree) to evaluate. This will significantly improve the efficiency of implementation since relinearization is an expensive operation. The primitives are as follows:

- **KeyGen**: We choose a decreasing sequence of primes $q_0 > q_1 > \cdots > q_d$ and a polynomial $\Phi_m(x)$, the $m$-th cyclotomic polynomial of degree $n = \varphi(m)$. For each $i$, we sample $u^{(i)}$ and $g^{(i)}$ from distribution $\chi$, set $f^{(i)} = 2u^{(i)} + 1$ and $h^{(i)} = 2g^{(i)}(f^{(i)})^{-1}$ in ring $R_{q_i} = \mathbb{Z}_{q_i}[x]/(\phi(x))$. (If $f^{(i)}$ is not invertible in this ring, re-sample.)

- **Encrypt**: To encrypt a bit $b \in \{0, 1\}$ with a public key $(h^{(0)}, q_0)$, random samples $s$ and $e$ are chosen from $\chi$ and compute the ciphertext as $c^{(0)} = h^{(0)}s + 2e + b$, a polynomial in $R_{q_0}$.

- **Decrypt**: To decrypt the ciphertext $c$ with the corresponding private key $f^{(i)} = f^{2^i} \in R_{q_i}$, multiply the ciphertext and the private key in $R_{q_i}$ then retrieve the message via modulo two reduction: $m = c^{(i)}f^{(i)} \pmod{2}$.

- **XOR**: For two ciphertexts $c_1^{(0)} = \text{Encrypt}(b_1)$ and $c_2^{(0)} = \text{Encrypt}(b_2)$ then their homomorphic XOR is evaluated by simply adding the ciphertexts $\text{Encrypt}(b_1 + b_2) = c_1^{(0)} + c_2^{(0)}$.

- **AND**: Polynomial multiplication is realized in two steps. We first compute
\( \tilde{c}^{(i-1)}(x) = c_1^{(i-1)} \cdot c_2^{(i-1)} \pmod{\phi(x)} \) and then perform a modulus reduction operation as \( \tilde{c}^{(i)}(x) = \left\lfloor \frac{q_0 \tilde{c}^{(i-1)}(x)}{q_0} \right\rfloor_2 \) where the subscript 2 on the rounding operator indicates that we round up or down in order to make all coefficients equal modulo 2.

### 4.3.3.1 Concrete Setting

To instantiate the Stehlé Steinfeld variant of NTRU for depth \( d \) we need to pick a large enough \( q_0 \) value to reduce the modulus \( d \) times. For instance, for a selection of \( B = 2 \) and if we cut by 24 bits in each iteration we need at least 200 bits. For such a \( q \) parameter we can then select \( n \) based on the Hermite factor. The Hermite factor was introduced by Gama and Nguyen [34] to estimate the hardness of the shortest vector problem (SVP) in an \( n \)-dimensional lattice \( L \) and is defined as

\[
\gamma^{2n} = \frac{||b||}{\text{vol}(L)^{\frac{1}{2n}}}
\]

where \( ||b|| \) is the length of the shortest vector or the length of the vector for which we are searching. The authors also estimate that, for larger dimensional lattices, a factor \( \delta^n \leq 1.01^n \) would be the feasibility limit for current lattice reduction algorithms. In [32], Lindner and Peikert gave further experimental results regarding the relation between the Hermite factor and the recovery time as \( t(\gamma) := \log(T(\gamma)) = 1.8/\log(\gamma) - 110 \). For instance, for \( \gamma^n = 1.0066^n \), we need about \( 2^{80} \) seconds on an AMD Opteron running at 2.5 Ghz [32]. Since we are using a construction based on NTRU we need to determine the desired Hermite factor for the NTRU lattice. Coppersmith and Shamir in [35] show that an attacker would gain useful information with a lattice vector as close as norm \( q/4 \) to the original secret key vector. Therefore we take \( ||b|| = q/4 \) and \( \text{vol}(L) = q^n \) and compute the
Hermite factor for the NTRU lattice as $\gamma = (\sqrt{q}/4)^{1/(2n)}$.

To select parameters we also need to consider the noise growth. Since we no longer use relinearization, the powers of the secret key will grow exponentially through the levels of evaluation. To cope with the growth we use the modulus reduction as described in Section 4.3.3. Following the noise analysis in Section 3.1.3 we can express the correctness condition as $||c^d f^2d||_\infty < q_d/2||$ assuming we are evaluating a depth $d$ circuit. Also note that instantiation we fix $\chi$ to choose from $\{-1, 0, 1\}$ with probabilities $\{0.25, 0.5, 0.25\}$, respectively. With modulus reduction rate of $\kappa \approx q_{i+1}/q_i$ the following equation holds $c^d f^{2d} = (\ldots ((c^2 \kappa + p_1)^2 \kappa + p_2)^2 \ldots \kappa + p_{2i}) f^{2d}$.

In Table 4.7 we computed the Hermite factor and supported depth for various sizes of $q_0$ and $n$ for our scheme.

Table 4.7: Hermite factor and supported circuit depth ($\gamma, d$) for various $q$ and $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\log_2(q)$</th>
<th>512</th>
<th>640</th>
<th>768</th>
<th>1024</th>
<th>1280</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{13}$</td>
<td>(1.01083, 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{14}$</td>
<td>(1.00538, 5)</td>
<td>(1.0135, 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$2^{15}$</td>
<td>(1.00269, 5)</td>
<td>(1.0067, 5)</td>
<td>(1.0162, 6)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3.4 The NTRU based PIR Protocol

In our encryption scheme we are able to batch additional information to the ciphertext polynomials. This allows us to perform retrieval using two different query mechanisms:

**Bundled Query** one query is used to retrieve data stored at different rows of the database (different indicies are queried).

**Single Query** the query retrieves data from a single row (a single index) but processes more indices at a time during the PIR server computation.
Next we explain an FHE optimization technique named *batching* and show how it gives us the two query methods.

**Batching.** Batching was introduced by Smart and Vercauteren [67, 28]. It allows us to evaluate a circuit on multiple independent data inputs simultaneously by embedding them into the same ciphertext. The independent data inputs are encoded to form special binary polynomials that are used as message polynomials. Addition and multiplication of these message polynomials has the effect of evaluating XOR and AND operations on the packed message bits. The encoding is achieved using the Chinese Remainder Theorem. First we set $R_{q_0} = \mathbb{Z}_{q_0}/(\Phi_m(x))$, where $\Phi_m(x)$ is defined as the $m^{th}$ cyclotomic polynomial. The cyclotomic polynomial $\Phi_m(x)$ is factored into equal degree irreducible polynomials over $\mathbb{F}_2$ $\Phi_m(x) = \prod_{i \in [\ell]} F_i(x)$, where $\lambda = \text{deg}(F_i)$ is the smallest integer that satisfies $m|(2^\lambda - 1)$. A message polynomial $m(x)$ in the residue space is represented as $m_i = m(x) \pmod{F_i(x)}$. Therefore, given a message bit vector $\mathbf{m} = \{m_0, m_1, m_2, m_3, \ldots, m_\varepsilon\}$ we may compute the corresponding message polynomial using inverse CRT $m(x) = \text{CRT}^{-1}(\mathbf{m})$. Using these special formed messages, we can perform bit level AND and XOR operations: $m_i \cdot m_i' = m(x) \cdot m'(x) \pmod{F_i(x)}$ and $m_i \oplus m_i' = m(x) + m'(x) \pmod{F_i(x)}$.

**Bundled Query.** The batching technique allows us to embed multiple indices into a query ciphertext and thereby facilitate retrieval of multiple database entries. First recall our PIR function $\sum_{y \in [2^\ell]} \prod_{i \in [\ell]} (x_i + y_i + 1)|D_y$, which we will now evaluate on encrypted $x$ and $y$ values. Using the batching technique we may evaluate $\varepsilon$ retrievals with indices $\beta[1], \ldots, \beta[\varepsilon]$ simultaneously. First we form their bit representation as:

$\beta[1] = (\beta_{\ell-1}[1] \quad \beta_{\ell-2}[1] \quad \ldots \quad \beta_0[1])$

$\beta[2] = (\beta_{\ell-1}[2] \quad \beta_{\ell-2}[2] \quad \ldots \quad \beta_0[2])$

$\vdots \quad \vdots \quad \vdots \quad \vdots$

$\beta[\varepsilon] = (\beta_{\ell-1}[\varepsilon] \quad \beta_{\ell-2}[\varepsilon] \quad \ldots \quad \beta_0[\varepsilon])$
Using the columns of the bit matrix on the RHS, we can compute the batched polynomial versions of the index bits $\tilde{\beta}_i(x)$ as:

$$\tilde{\beta}_i(x) = \text{CRT}^{-1}(\beta_i[1], \beta_i[2], \ldots, \beta_i[\ell])$$

Later, these polynomials are encrypted as: $\xi_i(x) = h(x)s_i(x) + 2e_i(x) + \tilde{\beta}_i(x)$ for $i \in [\ell]$. The query $Q = [\xi_i(x), \ldots, \xi_{\ell-1}(x)]$ is then sent to the PIR server. In order to perform parallel comparisons vector row index bit $\{y_i, y_i, \ldots, y_i\}$ should also be converted into a polynomial representation using inverse-CRT. Since we are dealing with bits $y_i = \{0, 1\}$, the inverse-CRT will result in $\{0, 1\}$ polynomials, and thus $y_i(x) = y_i$. This is true for data $D_y$ as well. Then, we can rewrite the PIR equation as: $r(x) = \sum_{y \in [2]} \left( \prod_{i \in [\ell]} (\xi_i(x) + y_i(x) + 1) \right) D_y(x)$. Given that $y_i(x)$ has small coefficient, i.e. 1 or 0, the additions are done over the least coefficient term in the polynomial. Furthermore, having $D_y(x) = \{0, 1\}$ we may skip the product evaluations unless $D_y(x) = 1$. Once $r(x)$ is homomorphically evaluated simultaneously over the $\ell$ ciphertexts, the response (a single ciphertext) $R = r([\xi_0(x), \ldots, \xi_{\ell-1}(x)])$ is sent back to the PIR client. The ciphertext response is first decrypted and the individual data entries are recovered using modular reductions: $D_i = \text{dec}(r(x)) \pmod{F_i(x)}$.

**Single Query.** In the single query mode we will also perform batching as in the Bundled Query mode. However, here we will place the same index into all index slots. The resulting polynomials are encrypted as before giving us a query $Q = [\xi_i(x), \ldots, \xi_{\ell-1}(x)]$. Though this is similar to the Bundled Query, the PIR server side computation is handled quite differently. For parallel comparisons we batch the
row bits of $y_i$ and $D_y$ as well:

$$y_i(x) = \text{CRT}^{-1}\{y_i[1], \ldots , y_i[\varepsilon]\}, \quad D_y(x) = \text{CRT}^{-1}\{D_y[1], \ldots , D_y[\varepsilon]\}.$$ 

These conversions are done on-the-fly and are not precomputed. Working in modulo 2 arithmetic makes the evaluations sufficiently fast and easy such that it only adds a small overhead. Although precomputation is an option, storing converted message polynomials would take extra space. The comparison equation will stay the same with the Bundled Query, but $y_i(x)$ and $D_y(x)$ will now binary polynomials. Therefore, we require polynomial addition inside the product and a polynomial multiplication with $D_y(x)$. Since in each iteration we are comparing $\varepsilon$ indecies simultaneously we can process the database $\varepsilon$ times faster. This speedup comes at a price where each iteration need to carryout a multiplication by the polynomial representation of the batched $D_y$.

The response ciphertext is first decrypted and then reduced to recover the evaluation bits as before: $z_i = \text{dec}(r([\xi_0(x), \ldots , \xi_{\ell-1}(x)])) \pmod{F_i(x)}$. In a Single Query each $z_i$ refers to a subsection of the summation therefore to compute the overall result we perform a final bit summation $D_y = \sum_{i \in \varepsilon} z_i \pmod{2}$.

### 4.3.5 Performance

We implemented the proposed PIR protocol with both the Single and Bundled Querying modes in C++ where we relied on Shoup’s NTL library version 6.0 [37] for the lattice operations. Table 4.8 shows minimal parameters to support various evaluation depths. Each depth can support up to $2^{d_\varepsilon}$ entries, e.g. $d = 5$ can support 4 Billion entries. The parameter $\varepsilon$ denotes the number of message slots that we can bundle. The query and response sizes are given in Table 4.8 without normalization.
Table 4.8: Polynomial parameters and Query/Response sizes necessary to support various database sizes $N$.

<table>
<thead>
<tr>
<th>$\max \ N$</th>
<th>$(\log q, n)$</th>
<th>$\varepsilon$</th>
<th>Query Size (MB)</th>
<th>Response Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Billion</td>
<td>(512, 16384)</td>
<td>1024</td>
<td>32</td>
<td>784</td>
</tr>
<tr>
<td>65536</td>
<td>(250, 8190)</td>
<td>630</td>
<td>3.9</td>
<td>154</td>
</tr>
<tr>
<td>256</td>
<td>(160, 4096)</td>
<td>256</td>
<td>0.625</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 4.9: Index comparison and data aggregation times per entry in the database for $(d, \varepsilon)$ choices of $(5, 1024)$, $(4, 630)$ and $(3, 256)$ on Intel Pentium @ 3.5 Ghz.

<table>
<thead>
<tr>
<th>Depth $(d)$</th>
<th>Bundled Query (msec)</th>
<th>Single Query (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Index comparison</td>
<td>4.45 0.71 0.31</td>
<td>4.56 2.03 1.29</td>
</tr>
<tr>
<td>Data aggregation</td>
<td>0.22 0.09 0.04</td>
<td>37 7.45 3.40</td>
</tr>
</tbody>
</table>

by $\varepsilon$. In the Bundled Query mode sizes may be normalized with $\varepsilon$ to determine the bandwidth per query. In Table 4.9, we present the time performance for query processing. The reported times are normalized per row of the database and per query. The time is split into two components: the time required to compare the encrypted index to the index of the currently processed row, and the time required to add the data in the current row to the summation. While the computation cost in comparison is quite high we should note that we are paying primarily for the index comparison. In the Bundled Query case, once the index comparison is completed we may simply reuse the comparison result and only compute an addition operation for each additional bit in the same database entry. In this sense, our results are similar to the other lattice based PIR construction by Melchor and Gaborit [64]. The index comparison may be considered as a one time overhead to be paid for each row that would be amortized as database rows get wider. Still due to the large vector sizes data aggregation will be rather slow. For instance; in a Bundled Query with $d = 4$ and 1 GBytes of data in a row, the processing time will be about 8 times slower than a Kushilevitz and Ostrovsky implementation as given in [66].
Table 4.10: Comparison of query sizes for databases up to $2^{32}$, $2^{16}$ and $2^8$ entries. Bandwidth complexity is given in the number of ciphertexts; $\alpha$ denotes the ciphertext size.

<table>
<thead>
<tr>
<th>BW Compl.</th>
<th>BGN $\alpha \sqrt{N}$</th>
<th>KO $\alpha \sqrt{N}$</th>
<th>Ours (Single) $\alpha \log^2 N$</th>
<th>Ours (Bundled) $\alpha \log N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d = 5$</td>
<td>6144</td>
<td>2048</td>
<td>8388608</td>
<td>8192</td>
</tr>
<tr>
<td>$d = 4$</td>
<td>6144</td>
<td>2048</td>
<td>2047500</td>
<td>3250</td>
</tr>
<tr>
<td>$d = 3$</td>
<td>6144</td>
<td>2048</td>
<td>655360</td>
<td>2560</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query Size</th>
<th>$d = 5$</th>
<th>$d = 4$</th>
<th>$d = 3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>96 MB</td>
<td>32 MB</td>
<td>32 MB</td>
</tr>
<tr>
<td></td>
<td>384 KB</td>
<td>128 KB</td>
<td>3.9 MB</td>
</tr>
<tr>
<td></td>
<td>24 KB</td>
<td>8 KB</td>
<td>640 KB</td>
</tr>
</tbody>
</table>

What we lose in computational efficiency, we make up for in terms of bandwidth. In Table 4.10, we give Complexity and Query size comparisons. As before, $N$ is the size of the database and $\alpha$ is the ciphertext size that differs in each scheme. In the Bundled Query case, for instance, the query is formed by $\ell = 2^d = 32$ ciphertexts each made of Mbytes. By normalizing with $\varepsilon$ index retrievals in a single query, per retrieval we are paying about 32 Kbytes. The query size of our scheme is smaller by a factor of 1024, 1200, and 3072 when compared to BGN, Melchor-Gaborit and Kushilevitz-Ostrovsky, respectively.

Finally, we would like to point out that for all practical purposes the size $\alpha$ of the ciphertexts in the query and response can be considered *almost* independent of the database size. Therefore, the size of the ciphertext, i.e. $\alpha$ is very mildly effected when the database size is increased. Indeed, as seen in Table 4.10 when the table size is grown from 256 entries to $2^{16}$ entries, the ciphertext size grows only about by 2.52 times in the bundled case.

\[\text{For [64], we used the given size of 37.5 MByte for 20,000 entries since it does not provide a complexity. The size will grow significantly when } N \text{ goes to } 2^{32}.\]
Chapter 5

FHE Hardware Designs

In this chapter, we introduce two hardware designs that is used to accelerate FHE schemes. In Section 5.1 we designed the first hardware for fully homomorphic encryption schemes, specifically Gentry’s FHE scheme. Later, in Section 5.2 we introduce a hardware multiplier for polynomials to accelerate LTV based FHE schemes.

5.1 Implementation of Gentry’s FHE in Hardware

In this section, we explain the first FHE hardware which is constructed for Gentry’s FHE scheme [25]. We implemented the primitive functions that is summarized in the background 2.1.2. The outline of the section is as follows; first we give some background information on the arithmetic operations underlying the hardware design, later we share the overview of our hardware architecture and give details on the large integer multiplier and FHE primitive hardware blocks. Later, we give the implementation details and finish the section by sharing a comparison of our hardware with the current software implementations.
5.1.1 Background

5.1.1.1 Number Theoretic Transform Based Arithmetic

The Number Theoretic Transformation (NTT) is a special form of Fourier Transform over rings. This special form eliminates the error prone structure of Fourier Transform because of the floating point arithmetic. We use NTT as the backbone of the million–bit arithmetic operations. Especially, it is effective for large integer multiplication (in million–bit range) which lies at the heart of all the primitives. Common multiplication schemes (Karatsuba Algorithm [68], schoolbook multiplication method) become infeasible for very–large integer multiplications. Schönhage–Strassen algorithm [30] is currently asymptotically the fastest algorithm for very–large numbers. It has been shown that it outperforms classic schemes for operand sizes larger than $2^{17}$ bits [69]. FFT–based large integer multiplier architectures were presented in [70, 71, 72].

**Schönhage Strassen Algorithm.** The Schönhage–Strassen Algorithm is an NTT–based large integer multiplication algorithm, with a runtime of $O(N \log N \log \log N)$ [30]. For an $N$–digit number, NTT is computed using the ring $R_N = \mathbb{Z}/(2^N + 1)\mathbb{Z}$, where $N$ is a power of 2. A summary of the algorithm is as follows. For an in–depth review of the Schönhage Strassen algorithm see [73]. We sample the numbers $A$ and $B$ that fits into $N$ digits with a sampling size $\epsilon$. The selected $p$ is a prime number with a primitive root $w$, i.e. $w^p = 1 \pmod p$. Then, we can represent the NTT forms of the numbers as $A_k = \sum_{k=0}^{N-1} w^k a_k$ and $B_k = \sum_{k=0}^{N-1} w^k b_k$. Later, the components are multiplied to form $c_k = A_k \cdot B_k \mod p$. Using the inverse–NTT (INTT) we compute $C_k = \sum_{k=0}^{N-1} w^{-k} c_k$. In the last step, we accumulate the carry additions to finalize the evaluation of $C$. To realize the Schönhage–Strassen Algorithm efficiently, it is crucial to employ fast NTT and INTT computation techniques. We adopted
the most common method for computing Fast Fourier Transform (FFT), i.e. the Cooley–Tukey FFT Algorithm [74]. The algorithm computes the Fourier Transform of a sequence $X$ as $X_k = \sum_{j=0}^{N-1} x_j e^{-i2\pi k \frac{j}{N}}$, by turning the length $N$ transform computation into two $\frac{N}{2}$ size Fourier Transform computations as follows

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} \theta^m + e^{-\frac{2\pi i k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} \theta^m,$$

where $\theta = e^{-2\pi i \frac{k}{N}}$. We change $e^{-\frac{2\pi i k}{N}}$ with powers of $w$ and perform the divisions into two halves with odd and even indices, recursively. With the use of fast transform technique, we can evaluate the Schönhage–Strassen Multiplication Algorithm in $O(N \log N \log \log N)$ time.

**Modular Reduction.** We may use Barrett Modular Reduction (BMR) algorithm [75] to compute $r \equiv x \pmod{M}$ as following:

$$r \equiv x \pmod{b^{k+1}} - \left\{ \left\lfloor \frac{x/b^{k-1}}{b^{k+1}} \right\rfloor \mu \right\} M \pmod{b^{k+1}}.$$

In the equation $b$ is the radix and other parameters are $k = \log_b M + 1$ and $\mu = \lfloor b^{2k}/M \rfloor$. According to [75] $r$ has the following equality: $r < 3M$. Therefore; after evaluating $r$, first we check if it is negative and perform $r = r + b^{k+1}$ and later we subtract $M$ from $r$ while $M < r$.

**Block–wise Arithmetic.** In the further sections of the paper, we refer to block–wise (or block) computations. The term defines separation of a large integer in the NTT form into computational blocks where each block contains $l$ digits. Each integer that is in NTT form will be formed of $N/l$ number of blocks. Since the NTT structure of these large integers is suitable for performing parallel arithmetic, these
blocks are distributed among arithmetic units.

### 5.1.2 Overview of Our Architecture

The overall architecture presented in Figure 5.1 contains five components: **Large Integer Multiplier**, **Barrett Reduction Unit**, **Decryption Completion Unit**, **Encryption Unit** and **Recryption Unit**. These are controlled by the **Master Control Unit (MCU)**. Each of the Encryption, Decryption and Recryption primitives require large integer multiplications. However, providing a dedicated **Large Integer Multiplier** for each primitive is too costly. In our design we incorporate one **Large Integer Multiplier** that will be shared between these
primitives.

To realize each primitive, the MCU controls the units, handling the order of operations and I/O between the units and the external memory. Since the operands are in the range of million-bits, data transactions between units are impractical. We assume an external memory (RAM) in the design for storage. We utilize a 64-bit bus for I/O transactions between the units and the external RAM. Holding the public keys, RAM acts as a shared memory between the units when the primitive operations are realized. The total operation time of a unit is computed as the sum of the time needed to read data from the RAM, the latency of the arithmetic operations and the time needed to write the result back to RAM. With effective addressing and utilizing prefetching from RAM, the initial address decoding overhead can be eliminated.

5.1.2.1 Parameter Selection

In the following we explain the details of the parameter selection for Large Integer Arithmetic to support million-bit multiplication operation. Next, we give parameters for FHE primitives and show some potential trade-offs.

Large Integer Arithmetic Parameters. The parameters for the NTT-based implementation is based on [76, 77]. We choose a 64-bit word size, and a sampling size of $\epsilon = 2^{24}$ with modulus $p = 2^{64} - 2^{32} + 1$, a Solinas prime [78]. This allows us to realize a modular reduction using a few primitive arithmetic operations. A 128-bit number is denoted as $z = 2^{96}a + 2^{64}b + 2^{32}c + d$. Using the selected $p$, we perform $z \pmod{p}$ operation as $2^{32}(b + c) - a - b + d$. The large integer size parameter $N$ is chosen to satisfy $\frac{N}{2}(\epsilon - 1)^2 < p$ to prevent overflow. Also, $N$ should be large enough to enable million-bit multiplication with smallest possible value, i.e. $2$ million bits $< N \cdot \epsilon$. The best candidate for $N$ is determined as $3 \cdot 2^{15}$.\footnote{We choose the digit size as a power of two which ease arithmetic computations.} Given
the parameters and equation $w^N \equiv 1 \mod p$, $w = 3511764839390700819$.

In Cooley–Tukey FFT, each recursive halving operation is referred as a stage and it is denoted as $S_i$, where $i$ is the stage index. The size of the smallest NTT block is selected as 12 digits and it is referred as the $0^{th}$ stage, i.e. $S_0$. The remaining 13 stages are reconstruction stages and require different arithmetic operations from the ones in $S_0$. In terms of INTT operations, every stage and operation is identical to NTT. Only difference is selection of $w'$, which is computed as: $w' = w^{-1} \mod p$.

**FHE Primitive Parameters.** We instantiate the scheme for the smallest parameters in [25] as; $n = 2048$, $l = 46$, $s = 15$, $p' = 5$, $S = 512$, $\rho = \frac{2032}{2048}$ and $\log d = 785000$.

For FHE primitives, we utilize addition operations in NTT form, i.e. $\sum_{i=1}^{j} u_i(\epsilon - 1)^2 < p$. The $j$ value is equal to $S$ and $(1 - \rho) \cdot n$ in Recryption and Encryption respectively. Therefore, we choose a different sample rate $\epsilon$ to prevent overflow. Selecting $\epsilon = 2^{16}$, we support primitive operations up to 786432 bits, which is larger than $\log d$.

### 5.1.3 Large Integer Architecture

The FHE primitives are based on efficient large integer arithmetic. In the following, we give the design details of a large integer multiplier and a modular reduction architecture.

#### 5.1.3.1 Large Integer Multiplier

**Architecture Overview.** Our architecture is composed of a data cache, a multiplier control unit, two routing units and a function unit, which is illustrated in Figure 5.2. The architecture is designed to perform a restricted set of special functions. There are four functions for handling the input/output transactions and three
functions for arithmetic operations:

- **Sequential Load**: Stores a million-bit number in the cache.

- **Sequential Unload**: The cache releases its contents starting from the least significant to most significant.

- **Butterfly Load**: Distributes the digits into the right indices using the butterfly operation.

- **Scale & Unload**: Scales and outputs the result, i.e. \( N^{-1} \mod p \).

- **12x12 NTT/INTT**: The smallest NTT/INTT computation is for 12 digits. NTT Unit\(^2\) takes the digits sequentially and computes the 12-digit NTT/INTT by using simple shift and add operations.

- **Stage-Reconstruction**: This function is used for reconstruction of a stage. In order to complete a full reconstruction, it is called for 13 stages.

- **Inner-Multiplication**: Computes the digit-wise modular multiplications. For this we utilize the multipliers used in Stage-Reconstruction Units.

\[^2\text{We refer to 12x12 NTT/INTT Unit as NTT Unit}\]
Using the functions outlined above, we can compute the product of million-bit numbers \(A\) and \(B\) using the following sequence of operations:

1. \(A\) is loaded into cache by using \textsc{Butterfly Load}.

2. The \textsc{NTT} of number \(A\), i.e. \(\text{NTT}(A)\), is computed by calling; first 12x12 \textsc{NTT} function, and afterwards \textsc{Stage-Reconstruction} function for all stages.

3. \(\text{NTT}(A)\) is stored to RAM using \textsc{Sequential Unload}.

4. Using steps 1-2-3 above, we also compute \(\text{NTT}(B)\).

5. The cache can only hold the half of the digits of \(\text{NTT}(A)\) and \(\text{NTT}(B)\) together. Therefore, the numbers are divided into lower and upper halves: \(\text{NTT}(A) = \{\text{NTT}(A)_h, \text{NTT}(A)_l\}\) and \(\text{NTT}(B) = \{\text{NTT}(B)_h, \text{NTT}(B)_l\}\).

6. \textsc{Sequential Load} stores \(\text{NTT}(A)_h\) and \(\text{NTT}(B)_h\).

7. \textsc{Inner Multiplication} computes modular multiplication of the upper halve:
   \[
   C_h[i] = \text{NTT}(A)_h[i] \ast \text{NTT}(B)_h[i].
   \]

8. The result is stored to the RAM by \textsc{Sequential Unload}.

9. We repeat above three steps to compute the lower part: \(C_l[i] = \text{NTT}(A)_l[i] \ast \text{NTT}(B)_l[i]\).

10. The result digits, i.e. \(C[i]\), are loaded into the cache by \textsc{Sequential Load}. At this point the cache will contain the multiplication result, but still in the \textsc{NTT} form.

11. The result is converted to integer form by using, 12x12 \textsc{INTT} function which is followed by a complete \textsc{Stage-Reconstruction} function \(C' = \text{INTT}(C[i])\).
12. In the last step, the result is scaled and the carries are accumulated by SCALE & UNLOAD function to finalize computation of $C$: $C[i+1] = C'[i+1] + \lfloor C'[i]/p \rfloor$ and $C[i] = C'[i] \pmod{R}$.

**Multiplier Cache System.** The size of the cache is important for the latency of multiplications. In each STAGE–RECONSTRUCTION process of the NTT algorithm, we need to match the indices of *odd* and *even* digits. The index difference of the *odd* and *even* digits in a reconstruction stage is: $S_{i,diff} = 12 \cdot 2^{i-1}$, where $i$ is the index of reconstruction stages, i.e. $1 \leq i \leq 13$. Since, in later stages we require digits from distant indices, an adequate sized cache is chosen to reduce the number of input/output transactions between the cache and RAM.

Let's call $N'$ as the chosen cache size. Then, we can divide the $N$ digits into $2^t = N/N'$ blocks, i.e. $N = \{N_{2^t-1}, N_{2^t-2}, \ldots, N_0\}$. Once a block is given as input, we can compute the reconstruction stages until $N' < S_{i,diff}$ for the $i^{th}$ stage. Then, starting from the $i^{th}$ stage, $N_j$ requires digits from $N_{j+1}$ where $j$ is block index. So, we need to divide $N_j$ and $N_{j+1}$ into halves and match the upper halves of $N_j$ with $N_{j+1}$, and lower halves of $N_j$ with $N_{j+1}$. This matching process adds $2N'$ clock cycles for each block. Then, the total input/output overhead is evaluated as $2N \cdot \log_2(N/N')$, where $\log_2(N/N')$ is the number of the stages that requires digit matching from different blocks. In our implementation, we aim to optimize the speed by selecting $N'$ as $N$.

Although a huge sized cache is important for our design, a straight cache implementation is not sufficient to support parallelism. The main arithmetic functions utilized in the multiplication process, such as $12 \times 12$ NTT/INTT, STAGE–RECONSTRUCTION and INNER MULTIPLICATION, are highly suitable for paralleliza-

---

3We used one $12 \times 12$ NTT/INTT unit for this function. For few number of arithmetic units for multiplier, i.e. $m = 4$, the performance gain is %3. However, for larger $m$ such as 64, performance gain will go up to %20.
Table 5.1: Assignment Table

<table>
<thead>
<tr>
<th>S_{11}</th>
<th>S_{12}</th>
<th>S_{13}</th>
</tr>
</thead>
<tbody>
<tr>
<td>arith_0</td>
<td>arith_1</td>
<td>arith_2</td>
</tr>
<tr>
<td>sc_0 - sc_7</td>
<td>sc_0 - sc_1</td>
<td>sc_0 - sc_2</td>
</tr>
<tr>
<td>sc_2 - sc_3</td>
<td>sc_4 - sc_5</td>
<td>sc_4 - sc_6</td>
</tr>
<tr>
<td>sc_5 - sc_7</td>
<td>sc_6 - sc_7</td>
<td>sc_5 - sc_7</td>
</tr>
</tbody>
</table>

To achieve parallelization, the cache should be able to sustain required bandwidth for multiple units. In order to sustain the bandwidth, we build up the cache by combining small, equal size caches or as we refer them sub-caches. Combining these sub-caches, we can select the cache to be used as a single–cache or a multi–cache system. In case of linear functions, such as Sequential Load, Butterfly Load, etc., the cache works as a single–cache with one I/O port, where as for parallel functions, it works as a multi–cache system with multiple I/O ports. The number of sub–caches should be equal to $2 \times m$ (double the size of Stage-Reconstruction Unit number) to eliminate access read/write to the same sub–cache in the reconstruction process. Each sub–cache has a size of $N/(2 \times m)$ and we denote them as; \{sc_0, sc_1, \ldots, sc_{2m-1}\}.

**Routing Unit.** The Routing Unit matches the odd and even digits to the arithmetic units. As stated previously, the indice difference of the digits is $(12 \cdot 2^{i-1})$. Therefore, in last $\log 2m$ reconstruction stage, odd and even digits fall into different sub–caches. The assignment of sub–caches to proper arithmetic units$^4$ for each Stage–Reconstruction is shown in Table 5.1. In the Table, arithmetic units are referred as arith_i, which i is the index number.

**Function Unit.** The Function Unit is divided into three parts, i.e. the Scaler Unit, the NTT Unit and multiple Stage–Reconstruction Unit.

$^4$Arithmetic units are the Stage–Reconstruction Units
Scaler Unit: Denoting the digits as $d_i$ and including the carries as $c_i$, digits of the result is $d_i \times N^{-1} + c_i \pmod{p} = \{c_{i+1}, r_i\}$. As $N^{-1} \pmod{p} = 0xFFF55555560001$ – a constant number with a special form, we implemented the product using simple shift and add circuit.

NTT Unit: The unit computes 12–digit NTT and INTT using the formula: $x_i = \sum_{i=0}^{11} (w')^i \times d_i \pmod{p}$, where $d_i$ is the given 12 digit input. The parameter $w'$ is set as; $w' = w^{2^{13}} \pmod{p}$ and $w' = (w^{-1})^{2^{13}} \pmod{p}$ for NTT and INTT operations respectively. Note that in NTT $w' = 0x10000$ and in INTT $w' = 0xFFFFF00010001$. These constant multiplications are implemented using simple add and shift circuits.

These simple operations can be squeezed into few clock cycles and pipelined to optimize throughput. Due to pipelining, 12–digit NTT of the large integer is completed in $N$ clock cycles.

Stage Reconstruction Unit: The unit in Figure 5.3 is responsible for two functions; Stage–Reconstruction and Inner Multiplication. The Arithmetic Logic Unit (ALU) in Figure 5.4 consists of 32–bit multipliers, adders and a reduction circuit to complete 64–bit modular multiplications.

In Inner Multiplication 64–bit numbers are fed into Odd and Coeff bus. Even is fed with zero, so that ALU only performs modular multiplication. The ALU can output a modular multiplication product in every two clock cycles after the initial startup cost of the pipeline. The whole function takes $\frac{N}{2}$ multiplications and with $m$ multipliers it will cost $\frac{N}{m}$ clock cycles.

In Stage–Reconstruction we compute: $O_{i,j} = E_{i-1,j} - O_{i-1,j} \times w_i^{j \pmod{n_{i-1}}} \pmod{p}$ and $E_{i,j} = E_{i-1,j} + O_{i-1,j} \times w_i^{j \pmod{n_{i-1}}} \pmod{p}$

where, $i$ denotes the stage index from 1 to 13, $j$ denotes the index of the digits, $w_i$ is the coefficient of stage $i-1$ and finally $n_{i-1}$ is the modular reduction to select the appropriate power of the $w_i$. The following equation is true for $n_i$ parameters;
Figure 5.3: Stage Reconstruction Unit

\[ n_{i+1} = 2 \times n_i \] with initial setting of \( n_0 = 12 \). Ideally we need to store all the coefficients along with the odd digits. However this will require another large cache of size \((12 + 24 + 48 + \cdots + 49152) \approx N \) digits. Although we save half the memory size by reuse of the powers for different stages, necessity of storing \( w^{-1} \) doubles the size requirement. We reduce the memory requirement by using memory–time tradeoff.

The coefficients are computed efficiently as follows:

1. The coefficients required in two consecutive stages are as follows: \( S_{i+1} : w_0^i, w_1^i, \ldots, w_n^i \) and \( S_{i+2} : w_0^{i+1}, w_1^{i+1}, \ldots, w_n^{i+1} \).

2. Then \( S_{i+2} : w_0^{i+1}, w_1^{i+1}, \ldots, w_2^{n_i} \) since it holds that \( n_{i+1} = 2 \times n_i \).

3. Further, \( S_{i+2} : w_0^i, w_1^i, \ldots, w_2^{n_i} \), since \( w_i = w_1^{2} \).

4. This shows that half of the coefficients of \( S_{i+2} \) are same as \( S_{i+1} \) and the other half are the square roots of the coefficients of \( S_{i+1} \).

5. We compute square roots by multiplying each \( w_1^i \) with \( w_1^{i+1} \).

Thus, we construct the Coefficient Table by storing two columns of coefficients. In the first column, since our smallest computation block is 12, we compute and store all \( w_0^j \) coefficients for \( 11 \geq j \geq 0 \). We denote these coefficients as \( w_{\text{first},i} \), where \( i \) denotes the index of the coefficient. In the second column, for each of the remaining stages we compute and store \( w_1^j \). The second column coefficients are denoted by \( w_{\text{second},i} \). This makes a total of 24 coefficients which we can use to compute
any of the $w_i^j$ values. When we include also the coefficients for the INTT operations, our table contains 48 coefficients. The computation of an arbitrary coefficient using the table can be achieved as $w_i^j = w_{\text{first},l} \times \prod_{t=0}^{i} w_{\text{second},t}^e$.

The values of $l$ and $e$ are functions of $i$ and $j$. Also $e$ is a value equal to 1 or 0. Therefore we can omit the multiplications whenever $e = 0$. The total number of multiplications, for evaluating $w_i^j \times O_i$, is computed as:

1. In every reconstruction stage we start by multiplying odd digits with $w_{\text{first},l}$’s. This step makes a total of $\frac{N}{2}$ multiplications.

2. Apart from the first reconstruction stage, in each stage we also require coefficients from $w_{\text{second},0}$ to $w_{\text{second},i-1}$. Since we cannot store the coefficients, in each stage we need to rebuild the previous stage coefficients to build up the coefficients. We are using half of the previous stage values so for each stage we need $\frac{N}{4}$ additional multiplications.

3. The total number of multiplications becomes $\sum_{i=0}^{N-1} (\frac{N}{2} + i \times \frac{N}{4}) = 26 \times N$.

**Figure 5.4: ALU of The Stage Reconstruction Unit**

**Multiplier Control Unit.** The Multiplier Control Unit includes a state machine for a large integer multiplication operation. The main job is to send correct
indices to Function Units to complete arithmetic functions, such as Inner Multiplication and Stage–Reconstruction. NTT Unit and Scale Unit consist only of datapath.

The Multiplier Control Unit also handles I/O addressing of the sub–caches. Sequential operation requires incremental addressing for each sub–cache. In Stage–Reconstruction operation, addressing is computed according to the stage level, which is basically updated with the index range of dependent odd and even digits.

**Performance Analysis.** The latency of each functional block is given in Table 5.2. In order to perform a complete multiplication we require two complete NTT, two Inner Multiplication and one INTT operations.

![Table 5.2: Clock Cycle Counts of Functional Blocks](image)

<table>
<thead>
<tr>
<th></th>
<th>2 Butterfly Load</th>
<th>2 12 x 12 NTT</th>
<th>2 Stage-Recon</th>
<th>2 Sequential Unload</th>
</tr>
</thead>
<tbody>
<tr>
<td>NTT(A), NTT(B)</td>
<td>2N</td>
<td>2N</td>
<td>26N</td>
<td>2N</td>
</tr>
<tr>
<td>AxB</td>
<td>2 Sequential Load</td>
<td>2 Inner Multiplication</td>
<td>2 Sequential Unload</td>
<td>2N</td>
</tr>
<tr>
<td>INTT(AxB)</td>
<td>Butterfly Load</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>12 x 12 INTT</td>
<td>13N</td>
<td>Scale Unload</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stage-Recon</td>
<td>N</td>
<td></td>
<td>TOTAL 52.5N</td>
</tr>
</tbody>
</table>

### 5.1.3.2 Modular Reduction

In Barrett Reduction, the result is evaluated using two large integer multiplications and a few subtractions. The values μ and M are stored in NTT form to avoid conversion costs. Selecting \( b = 2^{64} \) simplifies the arithmetic. Division and reduction
operations, such as \( \lfloor x/b^{k-1} \rfloor \) and \( x \pmod{b^{k+1}} \), are accomplished by loading from different memory address. For division, bits are read from \( b^{k-1} \) to most significant bit and for modular arithmetic, bits are read from least significant to \( b^{k+1} \).

Once Barrett Reduction is requested, the state machine performs the multiplications with \( M \) and \( \mu \), and computes \( r_1 \) and \( r_2 \) values. \( r_1 \) and \( r_2 \) are loaded for subtraction that is evaluated in digits by a simple subtracter with word length of 64 bits. Since reading both \( r_1 \) and \( r_2 \) occupies a huge portion of the bandwidth, a \( \kappa \)-digit local cache is added to store partial results to prevent the I/O collusions from/to the RAM. The local cache is based on two parallel \( \kappa/2 \)-digit FIFOs so that we can output 2 digits per clock cycle. Later, \( r \) is checked if it is negative and it is corrected by setting zero after \( b^{k+1} \). For comparison \( r \) and \( M \) are loaded into comparator unit to decide \( r \geq M \). The decision is done by implementing a 512–bit comparator. If the first 8 digits are equal then it loads the next 8 digits for decision until they are not equal. If comparison is true, \( r \) is updated as \( r = r - M \).

The time for a Barrett Reduction heavily depends on the multiplications and the subtractions have a small effect. The multiplications are completed in \( 2 \cdot 36.5N \) clock cycles. Subtractions take \( \approx 3 \times 23500 \) clock cycles, which is omitted. The total time required for a Barrett reduction is \( \approx 73N \) clock cycles.

### 5.1.4 FHE Primitives

#### 5.1.4.1 Decryption

The decryption operation is a rather simple operation that requires a modular multiplication operation followed by a modulo 2 reduction, i.e. \( \text{Decrypt}(c) = [cw_i]_d \pmod{2} \). During decryption, the Master Control Unit uses the Large Integer Multiplier and the Barrett Reduction units to realize \( [cw_i]_d \). This is
followed by the application of the Decryption Completion Unit, which contains a simple arithmetic circuit that takes the least significant digit of the modular multiplication result and pads it with zeroes to match the operand length.

We can reduce the large integer multiplication time by storing the $w_i$ in NTT form. The conversion operation is only applied to the ciphertext $c$. Therefore the multiplication operation takes $36.5N$ clock cycles. The modulo 2 reduction is realized by reading the last digit and by forming the large integer result with padding takes less than 8,000 clock cycles. Since $8,000 \ll 36.5N$ we neglect this quantity. Including Barrett Reduction, the overall decryption operation takes $109.5N$ clock cycles.

5.1.4.2 Encryption

The most time consuming part of encryption is evaluating powers of $r$. In [8], these are computed using a recursive algorithm. While asymptotically faster, such a recursive approach is not suitable for hardware implementations. Instead we utilize a window–based serial evaluation scheme. Algorithm $A'$ proposed in [79] is suitable for efficient polynomial evaluation in hardware.

The algorithm divides the evaluation into three steps. First, the polynomial terms are grouped into windows of $k$ digits where each grouping is multiplied by
Algorithm 1: Algorithm $A'$

1. Define $u(r) = u_0 r^0 + u_1 r^1 + \cdots + u_{n-1} r^{n-1}$ and a window size $k$ as $k < n$ and $k \mid n$.

2. Group coefficients of $u(r)$ using powers of $r^k$ as:
   
   \[
   (u_0 r^0 + \cdots + u_{k-1} r^{k-1}) r^0 + (u_k r^0 + \cdots + u_{2k-1} r^{k-1}) r^k + \cdots + (u_{n-k} r^0 + \cdots + u_{n-1} r^{k-1}) r^{n-k}.
   \]

3. Define Inner Polynomials as:
   
   \[
   P(j) = r^{j \cdot k} \left( \sum_{i=0}^{k-1} u_{(j+k+i)} r^i \right)
   \]

4. Then, the $u(r)$ polynomial can be rewritten as:
   
   \[
   u(r) = \sum_{j=0}^{\frac{n}{k}-1} P(j) = \sum_{j=0}^{\frac{n}{k}-1} r^{j \cdot k} \left( \sum_{i=0}^{k-1} u_{(j+k+i)} r^i \right)
   \]

increasing powers of $r^k$. After the summation operations, the window sums are scaled by the proper powers of $r^k$. The last step is to aggregate the scaled window sums. The algorithm reduces the number of multiplications to $k + 2 \frac{n}{k}$. A further speed-up is achieved by storing two tables: \{ $r^0, r^1, \ldots, r^{k-1}$ \} and \{ $r^k, r^{2k}, \ldots, r^{n-k}$ \}. Since $r$ is set during the KeyGen step the lookup tables can be precomputed. With the introduction of the lookup tables, the only multiplication operations needed are the ones computed when the window sums are multiplied by the power of $r^k$. Using the lookup tables, the number of multiplications are reduced further to $\frac{n}{k} - 1$ with a storage requirement of $\frac{n}{k} + k - 2$.

The algorithm can be further improved by realizing the operations entirely in the NTT domain. By storing the table elements in NTT form, an encryption operation may be realized as

\[
\text{Encrypt}(m) = \text{INTT} \left[ M + 2 \sum_{j=0}^{\frac{n}{k}-1} R^{j \cdot k} \left( \sum_{i=0}^{k-1} u_{(j+k+i)} R^i \right) \right]
\]

where we use uppercase symbols to denote the NTT form of the variables, e.g. $R = \text{NTT}(r)$ and $R^j = \text{NTT}(r^j)$. Since the message $m$ is a single bit, we simplify $M = (0, \ldots, 0)$ if $m = 0$, else $M = (1, \ldots, 1)$ if $m = 1$. The equation eliminates
NTT conversions and requires only one INTT and a single modular reduction at the end.

NTT–based arithmetic operations in aggregate, referred to as NTT Encryption, are evaluated with what we call the Encryption Unit. Remainder of the operations are completed by utilizing the Large Integer Multiplier Unit and the Barrett Reduction Unit. To realize the encryption primitive, Master Control Unit runs Encryption Unit, Large Integer Multiplier and Barrett Reduction units in order.

Encryption Unit. Encryption Unit is designed as a semi–systolic architecture as illustrated in Figure 5.5. Its architecture contains a Control Unit, a Storage Unit for $u$ and Encryption Processing Elements (EPEs). Since NTT–based arithmetic can be efficiently parallelized, RAM access latency becomes the bottleneck in our design. We can achieve maximum throughput by incorporating $\#\text{EPEs}=\text{bandwidth}/\text{frequency}$ processing elements into the design.

Encryption Processing Element (EPE). EPE as shown in Figure 5.6 is designed to evaluate NTT–Encryption of a block size $\kappa$. Parameter $\kappa$ also represents the size of the local cache. The local cache acts as a temporary variable $t$ and it is used to reduce the number of I/O transactions. It is also important to note that the unit is fully pipelined with 10 stages. Therefore each block operation will have an extra 10 clock cycles for time evaluations, i.e. we multiply the total timing with $(1 + 10/\kappa)$. The unit evaluates encryption with the following two steps:

- The first step evaluates the window summations and the scaling operation is shown in the Algorithm 2. With built–in local cache, a window summation can be evaluated in at most $k$ input and 1 output transactions. If a $u_i$ value is zero, then $R^{i}_{(l)}$ is not loaded into the system. Therefore, the probability of
the coefficients of the $u$ polynomial being 0 directly gives us the cost of the operations. The total number of I/O transactions is $k \cdot (1 - \rho) + 2$, where $1 - \rho$ is the probability of non-zero terms and plus 2 is for input of the scaler and output terms.

**Algorithm 2: Window Sum & Scale Operation**

```
\textbf{Input:} \ r = \{(R^0_0, R^1_0, \ldots, R^{k-1}_0),
\{R^k_0, R^2_0, \ldots, R^{n-k}_0\}\}, \ u = \{u_0, u_1, \ldots, u_{n-1}\}

\textbf{Output:} \ \text{Inner Polynomial Block} \ P^{(j)}_l = R^{j\cdot k}_l \sum_{0}^{k-1} u_i R^i_0
```

1. for $j = 0 \rightarrow \frac{n}{k} - 1$
2. \hspace{1em} $t \leftarrow 0$
3. \hspace{2em} for $i = 0 \rightarrow k - 1$
4. \hspace{3em} if $u_i \neq 0$ then $t \leftarrow t + u_i R^i_0$
5. \hspace{3em} $t \leftarrow t \cdot R^{j\cdot k}_l$
6. \hspace{2em} $P^{(j)}_l \leftarrow t$

- The second step computes the window summations along with the doubling and addition of the message bit to finalize the NTT-Encryption. The algorithm is shown in Algorithm 3. The total number of I/O transactions for the window summations also depends on the probability $\rho$. If $\rho$ is large enough, the probability of $P^{(j)} = 0$, i.e. $\rho^k$, will be sufficiently large that they can be ignored during the additions. Then, the total number of I/O transactions is $\frac{n}{k} \cdot (1 - \rho^k) + 1$, where $\frac{n}{k}$ represents the number of windows and plus 1 is for the output term.

The EPE is controlled by signals \texttt{double}, \texttt{operation}, \texttt{add\_bit}, \texttt{read}, \texttt{write} and \texttt{clear}. Each EPE is connected with a 64–bit bus, which is utilized to load the powers of $r$ into the system. During the computation of the first algorithm, \texttt{double} and \texttt{add\_bit} signals are inactive and the input is directly fed to the Modular Arithmetic Unit. The unit consist of a 64–bit modular subtracter, an adder and a multiplier.
Algorithm 3: NTT Encryption

Input: \( P = \{P_l^{(0)}, P_l^{(1)}, \ldots, P_l^{(n_k-1)}\} \)

Output: \( R_l = 2 \sum_{0}^{n_k-1} P(i)_l + M_l \)

1. \( t \leftarrow 0; \)
2. for \( j = 0 \rightarrow \frac{n_k}{k} - 2 \) do
3. \( \text{if } P^j \neq 0 \text{ then } t \leftarrow t + P_l^{(j)} + P_l^{(j)}; \)
4. \( \text{if } P^{(j)} \neq 0 \text{ then } t \leftarrow t + P_l^{(\frac{n}{k}-1)} + P_l^{(\frac{n}{k}-1)} + M_l; \)
5. \( ; \)
6. \( R_l \leftarrow t \)

The operation signal enables the required modular arithmetic operation. In case of \( u_i = \pm 1 \) modular adder/subtractor is used to compute \( t = t \pm R_i \). For the scaling operation, modular multiplier is enabled by the operation signal to compute \( t = t \cdot R^j \cdot k \). The second algorithm is realized using two 64-bit modular adders that are controlled by the double and add bit signals. If the double signal is active, the input is added to its own to double the window summations: \( 2P = P + P \). If the add bit signal is active, the message bit \( m \) is added to the summation in NTT form. Using these two signals, the final equation is evaluated as \( 2u(R) + M = 2 \sum_{0}^{n-1} P(i) + M \).

Additionally, it is important to note that the clear signal is used in case of setting \( t = 0 \) and read/write signals are used to read and update \( t \) values.

Control Unit. The Control Unit is a state machine for the encryption operation. The inputs are the message bit \( m \), random polynomial \( u \) and its outputs are
the operation, double, add_bit and clear signals. Once the \( u \) polynomial is loaded, the 
Control Unit performs an encryption operation as follows:

1. Take message bit \( m \) as input.

2. Request for the \( u \) polynomial for the first window, \( \{u_0, u_1, \ldots, u_{k-1}\} \).

3. Using clear signal, reset the cache units \( t \leftarrow 0 \).

4. Check the value of \( u_i \) iteratively and skip if it is zero. In case of \( \pm 1 \), \( \beta \) blocks of the powers \( r_i^j \) is loaded into the bus and operation signal is selected. Each arithmetic core is assigned with different blocks to evaluate \( t = t + r_i^j \).

5. Iterate index \( i \). Computation of the window sum is completed. To scale the sum, the necessary \( \beta \) blocks (powers of \( r_i^k \)) are loaded into the cache and the operation signal is set to enable multiplication. The term \( t \) is updated as: \( t' = t \cdot R_i^k \). Now \( t \) holds the result of a scaled window summation: \( P_{i,j}^{(j)} = R_i^{j-k} \sum_{0}^{k-1} u_i R_i^j \). Since there are \( \beta \) blocks, the window sum is evaluated for the \( \beta \) blocks as \( P_{\text{sub}}^{(j)} = \{P_{\beta-1}^{(j)}, \ldots, P_{1}^{(j)}, P_{0}^{(j)}\} \).

6. Sequentially write the results \( P_{\text{sub}}^{(j)} \) back to the main memory.

7. Using the steps 3–5, process each block to finish the computation of a window: \( P^{(j)} = \{P_{bs-1}^{(j)}, \ldots, P_{1}^{(j)}, P_{0}^{(j)}\} \), where \( bs \) is the block size.

8. Repeating steps 3–7, Compute all of the windows: \( \{P^{(0)}, P^{(1)}, \ldots, P^{(n-1)}\} \).

9. Using the clear signal, clear all caches to 0.

10. Assert the double signal starting from \( j = 0 \), \( \beta \) blocks of \( P^{(j)} \) is loaded if \( P^{(j)} \neq 0 \). This evaluates \( t = t + 2 \cdot P_{i,j}^{(j)} \) up to \( j = \frac{n}{k} - 1 \). The add_bit signal is activated for the case \( j = \frac{n}{k} - 1 \). This will add the message bit \( m \): \( t = t + 2 \cdot P_{i,j}^{(\frac{n}{k}-1)} + m \).
11. Every arithmetic core unit writes the result sequentially to the main memory.

12. Using Steps 9–12 process each block to finish the computation of the equation

\[ R = 2 \cdot \sum_{i=0}^{2^n-1} P^{(j)} + m. \]

Parameter selection affects the efficiency of the architecture significantly. Since an \( r^i \) term is included if \( u_i \) is not 0, the probability distribution of \( u_i = \{0, 1, -1\} \) is important to evaluate the timing. We select the window size as 64, and the probability is selected \( \rho = 16/2048 \). Since we only have 16 non–zero values, we need to evaluate 16 of 32 windows in the worst case scenario\(^5\). If 16 of these windows are evaluated, it will take \( 16 \cdot 3N \) cycles. Addition of these 16 windows will take \( 17N \) clock cycles. Including the number of EPE’s, \( \text{INTT} \) and Barrett Reduction operations, total cost of the operations will be \( \frac{65N}{\beta} \cdot (1 + \frac{10}{\kappa}) + 89N \) cycles.

5.1.4.3 Recryption

Recryption operation \( \text{Decrypt}_{SK}(c) \) is evaluated as

\[
\left[ \sum_{j \in \mathcal{S}} \sum_{i \in [S]} \sigma_j(i) z_{j,i} \right] + \sum_{j \in \mathcal{S}, i \in [l]} \sigma_j(i) (y_{j,i}) \pmod{2}.
\]

The first summation has following form:

\[
q_j = \sum_{a \in [l]} \beta_{j,a} \left( \sum_{b \in [l]} \beta_{j,b} z_{j,i(a,b)} \right) \pmod{d}.
\]

We can take advantage of the fact that the public keys \( \beta_{j,i} \) are known ahead of time after \( \text{KEYGEN} \). By precomputing and storing the keys in \( \text{NTT} \) form, i.e. \( B = \text{NTT}(\beta) \), we can eliminate many costly large integer multiplications. The equation is rewritten as

\[
q_j = \text{INTT} \left( \sum_{a \in [l]} B_{(j,a)} \left( \sum_{b \in [l]} B_{(j,b)} z_{j,i(a,b)} \right) \right) \pmod{d}.
\]

The new equation eliminates most of the \( \text{NTT} \) and \( \text{INTT} \) conversions. Only one inversion and one modular reduction is required at the end. Furthermore, we benefit by precomputing the \( z_{j,i} \) terms and storing them in a table. This allows us to compute the

\(^5\)We divide the degree 2048 into 64 windows of equal degree polynomials.
NTT based arithmetic parts in blocks, since we are able to re-read the $z_{j,i}$ for each block. We divide the above equation into four steps:

- Evaluation of the precision bits $z_{j,i}$.
- Sum of PKs
  $$S_j = \sum_{a \in [l]} B_{(j,a)} \left( \sum_{b \in [l]} B_{(j,b)} z_{j,i(a,b)} \right).$$
- INTT conversion and Barrett Reduction of $S_j$.
- Grade–School Addition.

First two steps are computed by Recryption Unit as shown in Figure 5.9. Last two steps are computed by the Master Control Unit using the Large Integer Multiplier and the Barrett Reduction Unit.

### 5.1.4.3.1 Evaluating Precision Bits
The precision bits $z_{j,i}$ are $p'$-bit result of the quotient of $y_{j,i}/d$. We divide the computation of $z_{j,i}$ terms into two units. First the Binary Computation Unit evaluates the precision bits $z_{j,i} = \{b_{j,i}^{(0)}, \ldots, b_{j,i}^{(p'-1)}\} = \frac{y_{j,i}}{d}$. In the equation, $j$ is the public key index, $i$ is the hamming weight index and $p' = \lceil \log_2(s + 1) \rceil + 1$ is the number of precision bits. The second unit used in the computation is the Modular Computation Unit which evaluates $y_{j,i} = c \cdot x_j \cdot R^i \pmod{d}$ by computing $y_{j,i} = y_{j,i-1} \cdot R \pmod{d}$. The units are designed to make the evaluations for one public key. Therefore, for a public key of size $s$ we reuse the units for each $x_j$. In the following we give the design details.

**Binary Computation Unit.** The Binary Computation Unit is illustrated in Figure 5.7. It consists of $p'$ bit quotient evaluation units, a $p'$-bit buffer and a storage table that has size of $p' \cdot S$, denoted as Precision Bit Table. As shown in the figure, the quotient evaluation unit is an architecture that performs binary division.
by shift and subtraction operations. By using this design, we have a smaller area and timing overhead will still remain small compared to the overall timing. The evaluation of precision bits, for values $j, i$, is as follows:

1. The **Quotient Evaluation Unit** takes the first $k_1$ bits of the values $y_{j,i}$ and $d$ which are loaded into storage denoted as $y_{head}$ and $d_{head}$.

2. Using a comparator $y_{head}$ and $d_{head}$ is compared: if $y_{head} >= d_{head}$ then $b_{i,j}^{(l)} = 1$ else $b_{i,j}^{(l)} = 0$.

3. The precision bit $b_{i,j}^{(l)}$ is loaded into the buffer. The value $d_{head}$ is updated as $d_{head} = d_{head} \gg 1$ using a 1-bit shifter. Also, $y_{head}$ is updated according using the value of $b_{i,j}^{(l)}$ as: if $b_{i,j}^{(l)} == 1$ then $y_{head} = y_{head} - d_{head}$ else $y_{head} = y_{head}$.

4. We iterate Steps 2 and 3 until all the precision bits are calculated.

5. The **Precision Bit Table** has $S$ rows and each evaluated precision bits are loaded to the $i^{th}$ row of the table from the buffer.

With 64–bit word size arithmetic, load operation takes $k_1/64$ cycles, each update of the values takes $k_1/64$ cycles and each precision bit evaluation with comparison takes 1 cycle. The process of one precision bit evaluation takes $(\frac{(p' + 1)k_1}{64} + 1)$ cycles.
**Modular Computation Unit.** Modular Computation Unit is used to evaluate $y_{j,i}$ using the equation $y_{j,i} = y_{j,i-1} \cdot R \pmod{d}$ and setting $y_{j,0} = c \cdot x_j \pmod{d}$. $R$ is a special number equal to $2^{103}$. This simplifies the multiplication into a simple shift operation. Also, the modular reduction operation can be evaluated by a scaled subtraction operation, i.e. $y_{j,i} - t \cdot d = y_{j,i} \pmod{d}$, where $t$ is the largest coefficient that ensures $t \cdot d < y_{j,i}$. By combining these two, the computation can be expressed as $y_{i,j} = (y_{i-1,j} \ll 103) - t \cdot d$. In the equation coefficient $t$ is at most 103 bits, so we are able to design a fast modular reduction unit that computes and multiplies the small coefficients with million–bit numbers. The design in Figure 5.8 consist of a 103–bit quotient evaluation unit, a 64x128–bit multiplier unit, a carry accumulate unit, a 103–bit shifter, a 64–bit subtracter and a local storage. Evaluation of $(y_{i-1,j} \ll 103) - t \cdot d$ is performed with the following steps:

1. **103–bit Quotient Evaluation Unit** takes the first $k_2$ bits of the values $y_{j,i-1}$ and $d$.

2. **Quotient Evaluation Unit** evaluates the $t$ value as a 103–bit number as explained in Binary Computation Unit.

3. Since the evaluations output one bit at a time, bits are loaded into a 128–bit buffer. The buffer will hold a zero-padded 103–bit $t$.

4. After computing $t$, we can evaluate $y_{j,i} = y_{j,i} - t \cdot d = (y_{j,i-1} \ll 103) - t \cdot d$

The evaluation of $t$ by the **Quotient Evaluation Unit** takes $(\frac{104k_2}{64} + 1)$ cycles. In the rest of the computations, the design inputs two million–bit numbers and outputs a million–bit result. The design is fully pipelined and able to generate a result at each clock cycle. Also, the pipeline delay is small that we can neglect for timing computations. Therefore, transactions take 47000/ bandwidth cycles to finish an evaluation, where bandwidth is the rate of digits per clock cycle.
Filling the Precision Bits Table. In order to complete the operations and fill the Precision Bits Table, we use Modular Computation Unit and Binary Computation Unit in turns. Using a local Control Unit, we iterate the modular computation and binary computation for $S$ times to complete the table for a single public key. Each public key has the initial modular multiplication $(c \cdot w_i \mod d)$, so we eliminate extra NTT of ciphertext $c$ by only converting it once and using it in each public key multiplication. Furthermore, we store the public keys in NTT form and eliminate the conversion operations, which reduces the timing significantly. Therefore, we only perform digit multiplications, INTT conversions and Barrett Reduction. Then, completing the table for a single public key takes:

$$\tau = 93.5N + \left( \frac{104k_2}{64} + 2 + \frac{47000}{\text{bandwidth}} \right) \times S \text{ cycles.}$$

Using the same units for other public keys adds a factor of $s$ to the overall timing, i.e. $s \cdot \tau + 16N$. However, each public key has an independent operation which we can benefit by using multiple of these units. Still we need to increase the bandwidth by the number of units to achieve a speedup.

---

Figure 5.8: Modular Computation Unit

---

6Only adds an initial $16N$ clock cycles in overall operation
5.1.4.3.2 Evaluating the Sum of Public Keys

Recall the equation for the summation of the public keys:

\[ s_j = \sum_{a \in [l]} \beta_{j,a} \left( \sum_{b \in [l]} \beta_{j,b} z_{j,i(a,b)} \right) \pmod{d} \, . \]

As before we chose to store the \( \beta \)'s in NTT form to eliminate the conversions and rewrite the equation as:

\[ s_j = \sum_{a \in [l]} B_{(j,a)} \left( \sum_{b \in [l]} B_{(j,b)} z_{j,i(a,b)} \right) \]

Since \( z_{j,i(a,b)} \) is a \( p' \)-bit value, it is denoted as \( z_{j,i} = \{ b_{j,i}^{(p'-1)}, \ldots, b_{j,i}^{(1)}, b_{j,i}^{(0)} \} \). Then, \( s_j \) turns into a \( p' \) sized array that each bit computation is performed separately. By denoting \( s_j = \{ s_{j}'^{(p'-1)}, \ldots, s_{j}'^{(1)}, s_{j}'^{(0)} \} \), we can expand the equations as:

\[ s_{j}'^{(k)} = \sum_{a \in [l]} B_{(j,a)} \left( \sum_{b \in [l]} B_{(j,b)} b_{j,i}^{(k)} \right) \]

where \( k \) is the bit index. For the evaluation of the equation, we designed a RECRYPTION PROCESSING ELEMENT (RPE) which includes small local storage for rapid calculations. As clearly shown in the equation the same \( B \) inputs are used in all the evaluations. Therefore, we formed an array of \( p' \) RPE units and distribute each \( b_{j,i}^{(k)} \) to a unit. By doing that, we compute all \( s_{j}'^{(k)} \) evaluations with a single transactions rather than \( p' \). Furthermore, we can replicate the RPE array for processing multiple blocks since the evaluations are in NTT form. Likewise encryption, we can replicate RPE arrays to speed up the computations. The bandwidth limits the number of RPE arrays we can utilize. The design illustrated in Figure 5.9 consists of the RPE Arrays, the PRECISION BITS TABLE\(^8\) and a CONTROL UNIT. In the following we give design details of the units.

---

\(^7\)The values \( a, b \) are omitted for simplicity.

\(^8\)The table formed in Evaluating Precision Bits
**Recryption Processing Element.** The design of the RPE is illustrated in Figure 5.10. The unit consist of two 64-bit modular adders, one 64–bit modular multiplier, a multiplexer and two local storage units. The local storage units are referred as $\text{up}_c$ and $\text{low}_c$ and has size of $\kappa$-digits each. The unit is fully pipelined with a total of 11 clock–cycle depth. A complete block evaluation of the equation, performed by
RPE is shown in Algorithm 4.

**Algorithm 4: Recryption Algorithm**

**Input:** \( B_j = \{B_{j,l-1}, \ldots, B_{j,1}, B_{j,0}\}, b_{(j,i)}^{(k)} \subseteq \{b_{(j,i)}^{(p-1)}, \ldots, b_{(j,i)}^{(1)}, b_{(j,i)}^{(0)}\} \)

**Output:** \( s_j = \sum_{a} B_{j,a} \left( \sum_{b} B_{j,b} \cdot b_{(j,i)}^{(k)} \right) \)

1. \( \text{low}_c = 0 \)
2. for \( a = 0 \rightarrow l - 1 \) do
3. \( \text{up}_c = 0 \)
4. for \( b = a + 1 \rightarrow l - 1 \) do
5. \( \text{if } b_{(j,i)}^{(k)} == 1 \) then
6. \( \text{up}_c = \text{up}_c + B_{j,b} \)
7. \( \text{low}_c = \text{low}_c + \text{up}_c \cdot B_{j,a} \)

**Control Unit.** The Control Unit incorporates a state machine to handle the transactions and compute the Recryption operation. It controls the Precision Bits Table for requesting the required \( z_{j,i} \) bits with index \( i \) and they are directly fed to the RPEs. The unit controls the request_bits, read/write and clear signals. Including the output transactions the operation requires \( \frac{(S+x+p') \cdot N \cdot (1+11/\kappa)}{\phi} \cdot s \) clock cycles, in which factor of \( s \) comes from the public key number and \( \phi \) comes from the number of RPE arrays. Given the FHE primitive parameters and \( x \) being 14, the timing is \( \frac{531 \cdot N \cdot (1+11/\kappa)}{\phi} \cdot s \).
5.1.4.3.3 Conversion and Reduction of Sum of Public Keys  Once the precision bits for each public key are evaluated, they need to be converted back from the NTT domain. Using an INTT and Barret Reduction algorithms, the conversions will take $89N$ clock cycles. Having $s$ public keys and $p'$ precision bits, the total operation will take $s \cdot p' \cdot 89N$ cycles. The operations can be parallelized by using multiple large multiplier units. This will increase the area by the number of multiplier units, but will reduce the computation time by the same factor.

5.1.4.3.4 Grade School Addition  In this section we explain the method we used to add five 15-bit numbers, where each bit of every number is in encrypted form. All the bits are represented by a very large number and a conventional addition algorithm does not apply. Assume we are realizing the bitwise addition operation: \( \{c, s\} = x + y \) where $c$ is the resulting carry bit of the addition operation in 5.1.4.3.4 and $s$ is the sum. The logic realizing this operation is called a half adder. The result $c$ and $s$ can be represented as follows: $c = x \ AND \ y$ and $s = x \ XOR \ y$. Given that we realize this half-adder on the ciphertext, we can modify the equations as follows: For \( \{C, S\} = X + Y \) we write $C = X \times Y$ and $S = X + Y$ where A, B, C and S are ciphertext and multiplication and addition operations are large-number modular arithmetic.

Since 15 5-bit numbers in ciphertext form need to be added, we utilize Wallace tree [80] approach to minimize the number of large multiplications. For this operation, we utilize a total of 78 large multiplication operations and a total of $33 + 33 + 32 + 27 + 14 = 139$ large addition operations. The large multiplication operations require modular reductions, so that the bit growth is prevented. In the evaluation of $C$, we use two multiplications followed by additions, so the multiplications is reduced after the additions. This reduce the Barrett Reduction operation
by half. Then, the total timing is equal to \(78 \cdot 52.5N + 39 \cdot 73N + 25N\), in which \(25N\) is approximate cost of the additions.

### 5.1.5 Implementation Results

The design was synthesized with Synopsys Design Compiler using 90 nm TSMC Library. Timing analysis shows a maximum frequency of 666 MHz. A moderate memory speed of 1333 MTps (Megatransfers per second) is selected for the Main Memory (RAM). The ratio between the memory and the main circuit frequency, i.e. \(\beta = \phi = \frac{1333}{666} = 2\), results in I/O speed of 2 transactions (digits in our case) per clock cycle, which led us to set our EPE and RPE Array numbers as 2 each. Also, local cache sizes of the RPE, EPE and Barrett Reduction Unit was selected as 256 digits to have a smaller pipeline delay, i.e. \(\kappa = 256\). Finally, as mentioned before, we incorporated a single multiplier into our design. Under these settings, the timings are found to be as shown in Table 5.3. Large Integer Multiplication,

<table>
<thead>
<tr>
<th>Operation</th>
<th># of Clock Cycles</th>
<th>Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Integer Multiplication</td>
<td>52.5N</td>
<td>7.75 msec</td>
</tr>
<tr>
<td>Barrett Reduction</td>
<td>73N</td>
<td>10.70 msec</td>
</tr>
<tr>
<td>Decryption</td>
<td>109.5N</td>
<td>16.16 msec</td>
</tr>
<tr>
<td>Encryption</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPE</td>
<td>(\frac{68N}{2}) \cdot (1.039)</td>
<td>4.98 msec</td>
</tr>
<tr>
<td>INTT &amp; Reduction</td>
<td>89N</td>
<td>13.12 msec</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>18.10 msec</strong></td>
</tr>
<tr>
<td>Recryption</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation of (z_{j,i})</td>
<td>((93N + 840S) \cdot s + 16N)</td>
<td>0.488 sec</td>
</tr>
<tr>
<td>Sum of PK</td>
<td>(\frac{531N}{2} \cdot (1.042) \cdot s)</td>
<td>0.612 sec</td>
</tr>
<tr>
<td>INTT &amp; Reduction</td>
<td>(s \cdot p \cdot 89N)</td>
<td>0.985 msec</td>
</tr>
<tr>
<td>Grade School</td>
<td>6967N</td>
<td>1.027 sec</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>3.112 sec</strong></td>
</tr>
</tbody>
</table>

Barrett Reduction and Decryption operations share a single Large Integer Multiplier.
Therefore their latencies are directly related to the latency of the multiplier. By increasing the number of EPE units as well as the bandwidth, we can reduce the latency of the Encryption operation. However, the encryption latency is already small and a single EPE takes only about 26% of the entire Encryption operation. The recryption operation is the most significant yet slowest operation. Therefore, we aimed at optimizing this operation as much as possible. Using the recommended bandwidth settings, we reduced the total time of the recryption operation to 3.1 seconds. For the initial evaluation of $z_{j,i}$ and sum of PKs, we can utilize same elements. Since they are independent, the timing of the first two steps can be reduced to $1.075/\lambda$, in which $\lambda$ is the number of arithmetic units utilized for these steps. This will increase the area and bandwidth requirements for those operations by a factor of $\lambda$. For the last two steps of the operation, we can reduce the timing by adding more LARGE INTEGER MULTIPLIER units. Using $\kappa$ multipliers, the latency of the INTT & Reduction operation can be reduced by a factor of $\lambda$ and the delay of the Wallace–Tree can be reduced by close to a factor of $\lambda$. In multiplication, I/O transactions take 20% of the total operation. Therefore, with 5 multipliers we can perform multiple operations without increasing the bandwidth.

The design is synthesized with local cache sizes of 256, 128 and 64 digits and the area results are shown in Table 5.4. The local cache sizes affect the area of Encryption, Recryption and Barrett Reduction units only. Among these three units, the Recryption Unit covers the largest area with 1.17 million gates for a 256-digit cache size. The Decryption Unit, whose only purpose is to take the least significant bit and to augment it with zero bits, does not have any local cache and consumes only about 200 gates and additional rewiring. The LARGE INTEGER MULTIPLIER has a fixed size cache that makes it the largest hardware unit in the design with 26.5 million gates for cache and 0.2 million gates for $m = 4$. 134
arithmetic units. By increasing the number of multipliers $m$ we may further reduce the execution time. The time for the large integer multiplication ($\frac{158N}{m} + 13N$) is tabulated for various $m$ in Table 5.5. Note that the time–area product is optimal for $m = 64$ improving multiplication speed by 3.4 times while the area is increased by only 11%. We estimate a 2 times speedup in Recryption.

### 5.1.6 Comparison

We presented the first hardware implementation of a fully homomorphic encryption system with the goal of exploring the limits of the GH–FHE scheme in hardware. Hence a direct comparison with other FHE implementations is not possible. While a comparing to software implementations would not be fair, we find it useful to summarize these results alongside ours in Table 5.6.

In **LARGE INTEGER MULTIPLICATION**, the time performance of our design is close to that of the Xeon software implementation and $\sim$10 times slower than GPU

<table>
<thead>
<tr>
<th>$m$</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (in $N$)</td>
<td>52.5</td>
<td>32.7</td>
<td>22.8</td>
<td>17.9</td>
<td>15.4</td>
<td>14.2</td>
</tr>
<tr>
<td>Area</td>
<td>26.7</td>
<td>26.9</td>
<td>27.3</td>
<td>28.1</td>
<td>29.7</td>
<td>32.9</td>
</tr>
<tr>
<td>Time $\times$ Area</td>
<td>1401</td>
<td>879.6</td>
<td>622.4</td>
<td>502.9</td>
<td>457.3</td>
<td>1817</td>
</tr>
</tbody>
</table>
Table 5.6: Times in msec (top) and in million cycles (bottom)

<table>
<thead>
<tr>
<th></th>
<th>Multiplication</th>
<th>Decrypt</th>
<th>Encrypt</th>
<th>Recrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ours</td>
<td>7.750</td>
<td>16.1</td>
<td>18.1</td>
<td>3100</td>
</tr>
<tr>
<td>GPU [81]</td>
<td>0.765</td>
<td>2.5</td>
<td>220</td>
<td>4200</td>
</tr>
<tr>
<td>Xeon [25]</td>
<td>6.667</td>
<td>20.0</td>
<td>1800</td>
<td>32000</td>
</tr>
<tr>
<td>Ours</td>
<td>5.1</td>
<td>10.7</td>
<td>12</td>
<td>2000</td>
</tr>
<tr>
<td>GPU [81]</td>
<td>0.8</td>
<td>2.8</td>
<td>253</td>
<td>4800</td>
</tr>
<tr>
<td>Xeon [25]</td>
<td>20</td>
<td>60</td>
<td>5400</td>
<td>96000</td>
</tr>
</tbody>
</table>

implementation. Decryption is 20% faster compared to Xeon software but it is 6.5 times slower than the GPU implementation. Encryption is 101 times faster than the Xeon software and 12.3 times faster than the GPU implementation. However, the most critical operation is Recryption and we are 10 times faster than the Xeon software implementation. Moreover, our design is still 1.1 second faster than the GPU implementation. We should note however that our hardware runs at a slower frequency with a much lower gate count of less than 30 million equivalent gates. In contrast, the NVidia GPU in [81] contains approximately 900 million and the Xeon processor contains 205 million gates. This shows the benefit of our ASIC design compared to general purpose CPU and GPU implementation with much higher performance at lower area cost. In Table 5.6 (bottom) we normalize the timings with the clock rates of the chips. In the most critical primitive, i.e. Recrypt, our implementation requires fewer than half the clock cycles compared to GPU and 46.6 times fewer clock cycles compared to Xeon implementations at a much lower footprint. Our implementation can benefit at least a few times speedup, if synthesized with a smaller technology than 90 nm like Xeon and GPU processors (40–45 nm).
5.2 Implementation of DHS FHE in Hardware

In this section we introduce a hardware implementation for our DHS FHE implementation. First, we give background information on the arithmetic operations that is used in the implementation. Later, we give the architectural overview of our scheme, give details on the polynomial multiplier hardware and give the implementation results. We end the section by giving a comparison between our polynomial multiplier and other software/hardware polynomial multipliers. We also estimate homomorphic AES and Prince implementations and make a comparison between the software and GPU implementations of those schemes.

5.2.1 Background

In this section we briefly outline the primitives of the López-Alt, Tromer and Vaikuntanathan’s fully homomorphic encryption based schemes, and later discuss the arithmetic operations that will be necessary in its hardware realization.

5.2.1.1 Arithmetic Operations

To implement the costly large polynomial multiplication and relinearization operations we follow the strategy of Dai et al. [82]. For instance, in the case of polynomial multiplication we first convert the input polynomials using the Chinese Remainder Theorem (CRT) into a series of polynomials of the same degree, but with much smaller word-sized coefficients. Then, pairwise product of these polynomials is computed efficiently using Number Theoretical Transform (NTT)-based multiplier as explained in subsequent sections. Finally, the resulting polynomial is recovered from the partial products by the application of the inverse CRT (ICRT) operation.
5.2.1.2 CRT Conversion

As an initial optimization we convert all operand polynomials with large coefficients into many polynomials with small coefficients by a direct application of the Chinese Remainder Theorem (CRT) on the coefficients of the polynomials:

\[ \text{CRT : } A_j \rightarrow \{ A_j \mod p_0, A_j \mod p_1, \ldots, A_j \mod p_{l-1} \} , \]

where \( p_i \)'s are selected small primes, \( l \) is the number of these small primes, and \( A_j \) is a coefficient of the original polynomial. Through CRT conversion we obtain a set of polynomials \( \{ A^{(0)}(x), A^{(1)}(x), \ldots, A^{(l-1)}(x) \} \) where \( A^{(i)}(x) \in R_{p_i} = \mathbb{Z}_{p_i}[x]/\Phi(x) \). These small coefficient polynomials provide us the advantage of performing arithmetic operations on polynomials in a faster and efficient manner. Any arithmetic operation is performed between the reduced polynomials with the same superscripts, e.g. the product of \( A(x) \cdot B(x) \) is going to be \( \{ A^{(0)}(x) \cdot B^{(0)}(x), A^{(1)}(x) \cdot B^{(1)}(x), \ldots, A^{(l-1)}(x) \cdot B^{(l-1)} \} \).

A side benefit of using the CRT is that it allows us to accommodate the change in the coefficient size during the levels of evaluation, thereby yielding more flexibility. When the circuit evaluation level increases, since \( q_i \) gets smaller, we can simply decrease the number of primes \( l \). Therefore, both multiplication and relinearization become faster as we proceed through the levels of evaluation. After the operations are completed, a coefficient of the resulting polynomial, \( C(x) \) is computed by the Inverse CRT (ICRT):

\[ \text{ICRT}(C_j) = \sum_{i=0}^{l-1} \left( \frac{q}{p_i} \right) \cdot \left( \left( \frac{q}{p_i} \right)^{-1} \cdot C^{(i)}_j \mod p_i \right) \mod q, \]

where \( q = \prod_{i=0}^{l-1} p_i \). Note that we will drop the superscript notation used for the reduced polynomials by the CRT for clarity of writing since we will deal with mostly the reduced polynomials henceforth in this paper.
5.2.1.3 Polynomial Multiplication

The fundamental operation in the LTV scheme, during which the majority of execution time is spent, is the multiplication of two polynomials of very large degrees. More specifically, we need to multiply two polynomials, \( A(x) \) and \( B(x) \) over the ring of polynomials \( \mathbb{Z}_p[x]/(\Phi(x)) \), where \( p \) is an odd integer and degree of \( \Phi(x) \) is \( N = 2^n \). Namely, we have \( A(x) = \sum_{i=0}^{N-1} A_i x^i \) and \( B(x) = \sum_{i=0}^{N-1} B_i x^i \). The classical multiplication techniques such as the schoolbook algorithm have quadratic complexity in the asymptotic case, namely \( \mathcal{O}(N^2) \). In general, the polynomial multiplication requires about \( N^2 \) multiplications and additions and subtractions of similar numbers in \( \mathbb{Z}_p \). Other classical techniques such as Karatsuba algorithm \[68\] can be utilized to reduce the complexity of the polynomial multiplication to \( \mathcal{O}(N \log_2 3) \). Nevertheless, the classical techniques do not yield feasible solutions for large \( N \). The NTT-based multiplication achieves a quasi-linear complexity \( \mathcal{O}(N \log N \log \log N) \) for polynomial multiplication, which is especially beneficial for large values of \( N \).

The NTT can essentially be considered as a Discrete Fourier Transform defined over the ring of polynomials \( \mathbb{Z}_p[x]/(\Phi(x)) \). Simply speaking, the forward NTT takes a polynomial \( A(x) \) of degree \( N - 1 \) over \( \mathbb{Z}_p[x]/(\Phi(x)) \) and yields another polynomial of the form \( \mathcal{A}(x) = \sum_{j=0}^{N-1} A_j x^j \). The coefficients \( A_i \in \mathbb{Z}_p \) are defined as \( A_i = \sum_{j=0}^{N-1} A_j \cdot w^{ij} \mod p \), where \( w \in \mathbb{Z}_p \) is referred as the twiddle factor. For the twiddle factor we have \( w^N = \mod p \) and \( \forall i < N \) \( w^i \neq 1 \mod p \). The inverse transform can be computed in a similar manner \( A_i = N^{-1} \cdot \sum_{j=0}^{N-1} A_j \cdot w^{-ij} \mod p \).

Once the NTT is applied to two polynomials, \( A(x) \) and \( B(x) \), their multiplication can be performed using coefficient-wise multiplication over \( \mathcal{A}_i \) and \( \mathcal{B}_i \) in \( \mathbb{Z}_p \); namely we compute \( \mathcal{A}_i \times \mathcal{B}_i \mod p \) for \( i = 0, 1, \ldots N - 1 \). Then, the inverse NTT (INTT) is used to retrieve the resulting polynomial \( C(x) = INTT(NTT(A(x)) \odot NTT(B(x))) \), where the symbol \( \odot \) denotes the coefficient-wise multiplication of \( \mathcal{A}(x) \) and \( \mathcal{B}(x) \).
in \(\mathbb{Z}_p\). Note that the polynomial multiplication yields a polynomial \(C(x)\) of degree \(2N - 1\). Therefore, before applying the forward NTT, \(A(x)\) and \(B(x)\) should be padded with \(N\) zeros to have exactly \(2N\) coefficients. Consequently, for the twiddle factor we should have \(w^{2N} = 1 \mod p\) and \(\forall i < 2N \ w^i \neq 1 \mod p\).

Cooley–Tukey algorithm [74], described in Algorithm 5, is a very efficient method of computing forward and inverse NTT. The permutation in Step 2 of Algorithm 5 is implemented by simply reversing the indexes of the coefficients of \(A_i\). The new position of the coefficient \(A_i\) where \(i = (i_n, i_{n-1}, \ldots, i_1, i_0)\) is determined by reversing the bits of \(i\), namely \((i_0, i_1, \ldots, i_{n-1}, i_n)\). For example, the new position of \(A_{12}\) when \(N = 16\) is 3. The inverse NTT can also be computed with Algorithm 5, using the inverse of the twiddle factor, i.e. \(w^{-1} \mod p\). Therefore, we can use the same

\[\begin{aligned}
\textbf{Algorithm 5: Iterative Version of Number Theoretic Transformation} \\
\text{input :} & \quad A(x) = A_0 + A_1x + \ldots + A_{N-1}x^{N-1}, \ N = 2^n, \text{ and } w \\
\text{output:} & \quad A(x) = \mathcal{A}_0 + \mathcal{A}_1x + \ldots + \mathcal{A}_{N-1}x^{N-1} \\
\text{for } i = N \text{ to } 2N - 1 \text{ do} & \quad A_i = 0; \\
(A_0, A_1, \ldots, A_{2N-1}) \leftarrow \text{Permutation}(A_0, A_1, \ldots, A_{2N-1}); \\
\text{for } M = 2 \text{ to } 2N \text{ do} & \quad \text{for } j = 0 \text{ to } 2N - 1 \text{ do} \\
\text{for } i = 0 \text{ to } \frac{M}{2} - 1 \text{ do} & \quad x \leftarrow i \times \frac{2N}{M}; \\
\mathcal{I} \leftarrow j + i; & \quad \mathcal{I} \leftarrow j + i + \frac{M}{2}; \\
\mathcal{J} \leftarrow j + i + \frac{M}{2}; & \quad \mathcal{J} \leftarrow \mathcal{A}[\mathcal{I}] + w^x \mod 2^N \times \mathcal{A}[\mathcal{J}] \mod p; \\
\mathcal{A}[\mathcal{I}] \leftarrow \mathcal{A}[\mathcal{I}] + w^x \mod 2^N \times \mathcal{A}[\mathcal{J}] \mod p; & \quad \mathcal{A}[\mathcal{J}] \leftarrow \mathcal{A}[\mathcal{J}] - w^x \mod 2^N \times \mathcal{A}[\mathcal{J}] \mod p; \\
i \leftarrow i + 1; & \quad i \leftarrow i + 1; \\
j \leftarrow j + M; & \quad j \leftarrow j + M; \\
M \leftarrow M \times 2; & \quad M \leftarrow M \times 2;
\end{aligned}\]
circuit for both forward and inverse NTT. Note that the NTT-based multiplication technique returns a polynomial of degree $2N - 1$, which should be reduced to a polynomial of degree $N - 1$ by diving it by $\Phi(x)$ and keeping the remainder of the division operation. When the reduction polynomial $\Phi(x)$ is of a special form such as $x^N + 1$, the NTT is known as Fermat Theoretic Transform (FTT) [83] and the polynomial reduction can be performed easily as described in [84] and [85].

5.2.1.4 Relinearization

Relinearization takes a ciphertext and set of evaluation keys $(E_{Ki,j})$ as inputs, where $i \in [0, l - 1]$ and $j \in [0, \left\lceil \log(q)/r \right\rceil - 1]$, $l$ is the number of small prime numbers and $r$ is the level index. Algorithm 6 describes relinearization as implemented in this work. We pre-compute the CRT and NTT of the evaluations keys (since they are fixed) and in the computations we perform the multiplications and additions in the NTT domain. The result is evaluated by taking $l$ INTT and one ICRT at the end. An

Algorithm 6: Relinearization with $r$ bit windows

- **input**: Polynomial $c$ with $(n, \log(q))$
- **output**: Polynomial $d$ with $(2n, \log(nq\log(q)))$

\begin{enumerate}
  \item $\{\tilde{c}_\tau\} = \text{CRT}(c)$ ;
  \item $\{\tilde{C}_\tau\} = \text{NTT}(\{\tilde{c}_\tau\})$ ;
  \item for $i = 0$ to $l - 1$ do
    \item load $E_{Ki,0}, E_{Ki,1}, \cdots, E_{Ki,\left\lceil \log(q)/r \right\rceil - 1}$ ;
    \item $\{D_i\} = \{\sum_{\tau=0}^{\left\lceil \log(q)/r \right\rceil - 1} \tilde{C}_\tau \cdot E_{Ki,r}\}$ ;
    \item $\{d_i\} = \text{INTT}(\{D_i\})$ ;
  \item $d = \text{ICRT}(\{d_i\})$ ;
\end{enumerate}

$r$-bit windowed relinearization involves $\left\lceil \log(q)/r \right\rceil$ polynomial multiplications and additions, which are performed again in the NTT domain. Since operand coefficients are kept in residue form, before relinearization we need to compute the inverse CRT of $\tilde{c}_\tau$.  

141
5.2.2 Architecture Overview

5.2.2.1 Software/Hardware Interface

The performance of the NTRU based FHE scheme heavily depends on the speed of the large degree polynomial multiplication and relinearization operations. Since the relinearization operation is reduced to the computation of many polynomial multiplications, a fast large degree polynomial multiplication is the key to achieve a high performance in the NTRU-FHE scheme. Having a large degree $N$ increases the computation requirements significantly, therefore a standalone software implementation on a general-purpose computing platform fails to provide a sufficient performance level for polynomial multiplications. The NTT-based polynomial multiplication algorithm is highly suitable for parallelization, which can lead to performance boost when implemented in hardware. On the other hand, the overall scheme is a complex design demanding prohibitively huge memory requirements (e.g., in homomorphic AES key requirements exceed 64-GB of memory). Therefore, a standalone architecture for SWHE fully implemented in hardware is not feasible to meet the requirements of the scheme.

In order to cope with the performance issues we designed the core NTT-based polynomial multiplication in hardware, where the polynomials have relatively small coefficients (i.e., 32-bit integers) to use it in more complicated polynomial multiplications and relinearization evaluations. The designed hardware is implemented in an FPGA device, which is connected to a PC with a high speed interface, e.g. PCI Express (PCIe). The PC handles simple and non-costly computations such as memory transactions, polynomial additions and etc. In case of a large polynomial multiplication or NTT conversion (in case of relinearization), the PC using the CRT technique, computes an array of polynomials whose coefficients are 32-bit integers
from the input polynomials of much larger coefficients. The array of polynomials with small coefficients are sent in chunks to the FPGA via the high-speed PCIe bus. The FPGA computes the desired operation: polynomial multiplications or only NTT conversion. Later, the PC receives the resulting polynomials from the FPGA and if necessary, i.e. before modulus switching or relinearization, evaluates the inverse-CRT to compute the result.

5.2.2.2 PCIe Interface

The PCIe is a serial bus standard used for high speed communication between devices which in our case are PC and the FPGA board. As the target FPGA board, we use Virtex-7 FPGA VC709 Connectivity Kit and can operate at 8 GT/s, per lane, per direction with each board having 8 lanes. The system is capable of sending the data packets in bursts. This allows us to achieve real time data transaction rate close to the given theoretical transaction rate as the packet sizes become larger.

5.2.2.3 Arithmetic Core Units

In order to achieve multiplication of two large degree polynomials, we designed hardware implementations for basic arithmetic building blocks to perform operations on the polynomial coefficients such as modular addition, modular subtraction and modular multiplication.

For compute–heavy operations using a large number of multiplication operations such as modular exponentiation and polynomial multiplication, it is a common practice, especially on word-oriented architectures, to perform partial reduction for the intermediate operations [86]. For example, when multiplying two 32–bit numbers with respect to a 32–bit modulus $p$, it is sufficient to achieve a result that is 32 bits in length, which can still be larger than the modulus $p$. This increases complexity of
modular addition and modular subtraction operations because of the massive number of operations realized in a single clock cycle for multiplication of two polynomials of degree $2^{14}$ and $2^{15}$. Therefore, we conclude that the most efficient method for the these modular operations is to achieve full modular reduction, and we design our building blocks to work with only fully reduced integers. Also, we base our design on an architecture to perform modular arithmetic operations for 32–bit numbers.

**32-bit Modular Addition** The modular addition circuit, which is illustrated in Figure 5.11b, takes one clock cycle to perform one modular addition operation where operands $A$, $B$ and the modulus $p$ are all 32-bit integers and $A, B < p$. As noted before, it is guaranteed that the result will be less than the modulus $p$. Since the largest values of $A$ and $B$ are $p - 1$, and thus the largest value of $A + B$ is $2p - 2$, at most one final subtraction of the modulus $p$ from $A + B$ will be sufficient to achieve full modular reduction after addition operation.

**32-bit Modular Subtraction**

The modular subtraction circuit, which is designed in a similar manner to modular addition circuit, is illustrated in Figure 5.11a. Similarly the subtraction unit is optimized to take one clock cycle to finish one modular subtraction operation on a target device. Since the largest values of $A$ and $B$ are $p - 1$, and the smallest values of $A$ and $B$ are 0, the largest value of their subtraction can be $p - 1$, and the smallest value can be $-p + 1$, which indicates that one final addition of the modulus $p$ will be sufficient to achieve full modular reduction after subtraction operation.

**Integer Multiplication** A DSP unit takes three inputs $A$, $B$ and $C$, which are 18 bits, 25 bits and 48 bits, respectively. $A$ and $B$ are multiplicand inputs, and $C$ is the accumulate input. The output is a 48–bit integer, which can be defined as $D = A \times B + C$. Therefore, we can accumulate the results of many $18 \times 25$–
bit multiplications without overflow. Since our operands are 32 bits in length, first we need to perform a full multiplication operation of 32–bit numbers. The operand lengths of the DSP units dictate that we need to perform four 16 × 16–bit multiplication operations to achieve a 32–bit multiplication operation. Utilizing four separate DSP slices, we could perform a 32–bit multiplication with 1 clock cycle throughput. However, this brings additional complexity to the hardware and because of the overall structure of the polynomial multiplication algorithm, 1–cycle throughput is not crucial for our design. Therefore, we decided to utilize a single DSP unit and perform the required multiplication operations to achieve a 32–bit
multiplication operation on the same DSP unit. This results in a 4–cycle throughput as explained below.

Algorithm 7: 33 × 33–bit integer multiplication

<table>
<thead>
<tr>
<th>Input</th>
<th>A = {A_1, A_0}, B = {B_1, B_0}, where A_1, B_1 are high 17 bits and A_0, B_0 are low 16 bits of A and B, respectively</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>C = A \times B</td>
</tr>
<tr>
<td></td>
<td>2 R1 ← A_0 \times B_0 + 0;</td>
</tr>
<tr>
<td></td>
<td>4 R2 ← A_0 \times B_1 + R1_H(R1_H = R1 &gt;&gt; 16);</td>
</tr>
<tr>
<td></td>
<td>6 R3 ← A_1 \times B_0 + R2;</td>
</tr>
<tr>
<td></td>
<td>8 R4 ← A_1 \times B_0 + R3_H(R3_H = R3 &gt;&gt; 16);</td>
</tr>
<tr>
<td></td>
<td>10 C ← {R4, R3_{L_1}, R1_{L_1}}(R1_{L_1} = R1&amp;0xFFFF, R3_{L_1} = R3&amp;0xFFFF);</td>
</tr>
</tbody>
</table>

In our design, however, we use Barrett’s algorithm [75] for modular reduction, which requires 33 × 33–bit multiplication operations, for which the utilized method is described in Algorithm 7. Therefore, we use DSP slices to perform 17 × 17–bit integer multiplications at a time as illustrated in Figure 5.12, instead of 16 × 16–bit multiplications, where both operations have exactly the same complexity. To minimize critical path delays, we utilize the optional registers for the multiplicand inputs and the accumulate output ports of the DSP unit as shown in Figure 5.12. These registers increase the latency of a single 33 × 33-bit multiplication to 6 clock cycles. On the other hand, the throughput is still four clock cycles, which allows the multiplier unit to start a new multiplication every four clock cycles.

We use classical multiplication algorithm and accumulate the result of the previous multiplication immediately after a 17 × 17–bit multiplication operation. The result will be in the registers T_1, T_0, T_{−1}, T_{−2}. Note that the wire widths in Figure 5.12 indicate the sizes of the operands and the intermediate values in our application, not the actual widths of the corresponding wires in the DSP units.

32-bit Modular Multiplication We use Barrett’s modular reduction algorithm [75] to perform modular multiplication operations. The Montgomery reduction algo-
Figure 5.12: Multiplier Circuit.
rithm [87], which is a plausible alternative to the Barrett reduction, can also be used for modular multiplication of 32-bit integers. However, the Montgomery arithmetic requires transformations to and from the residue domain, which can lead to complications in the design. Therefore, we prefer using the Barrett’s algorithm in our implementation to alleviate the mentioned complications in the design.

We use the algorithm adapted for 32-bit modular multiplication operations as illustrated in Algorithm 8. The comparison operation (and associated addition with $2^{33}$) in Step 9 is not needed in hardware implementation, as it is equivalent to checking the carry output of addition of $U$ and 2’s complement of $V$ after Step 8. More specifically, when the operation $U - V$ results in a negative number, the actual operation in hardware, where two’s complement arithmetic is used, produces no carry. Consequently, if we use exactly the 33 bits of the result ignoring whether there is a carry or not, we will always obtain the correct result.

The subtraction $W \leftarrow U - V$ in Step 8 can be at most a 33-bit number, more precisely $3p - 1$ as explained in [88]. Therefore, two subtractions in Steps 10–11 can be necessary to obtain the final complete result at the end. As we want to finish Steps 10–11 in a single clock cycle, we perform both subtractions in the hardware implementations simultaneously, namely $W - 2p$ and $W - p$, and select the correct result using the carry bits of the subtraction results and a multiplexer as illustrated in Figure 5.13. If $W - 2p$ is positive, it is guaranteed that it is a number in the range $0 \leq W < p$, and we select this result as the output. However, if $W - 2p$ is a negative number and $W - p$ is a positive number, we select $W - p$ as the correct output. If both subtractions yield negative results, we select $W$ as the output.

Our implementation of the Barrett algorithm, which is illustrated in Figure 5.13 takes 19 clock cycles to complete one modular multiplication of 32–bit integers whereas its throughput is four clock cycles. We will refer the first four clock cycles
Algorithm 8: Barrett Modular Multiplication Algorithm for 32-bit Modulus

**input**: $A, B, p,$ and $T$, where $A, B < p < 2^{32}$ and $T = \lfloor \frac{2^{64}}{p} \rfloor$

**output**: $C = A \times B \mod p$

1. $X \leftarrow A \times B$;
2. $Q \leftarrow X \gg 31$;
3. $R \leftarrow Q \times T$;
4. $S \leftarrow R \gg 33$;
5. $Y \leftarrow S \times p$;
6. $U \leftarrow X \mod 2^{33}$;
7. $V \leftarrow Y \mod 2^{33}$;
8. $W \leftarrow U - V$;
9. if $W < 0$ then
   10. $W \leftarrow W + 2^{33}$;
11. if $W - 2p > 0$ then
   12. $C \leftarrow W - 2p$;
13. else if $W - p > 0$ then
   14. $C \leftarrow W - p$;
15. else
   16. $C \leftarrow W$;

Figure 5.13: Architecture for 32-bit Modular Multiplier.
as the *warm up cycles* of the multiplier and the last 15 clock cycles as the *tail cycles*. These periods of clock cycles are important for the first and last multiplication operations performed in the pipeline architecture in Figure 5.13. We will need these pieces of information to accurately estimate the number of clock cycles needed in our computations in subsequent sections.

### 5.2.3 $2^n \times 2^n$ Polynomial Multiplier

We implemented a $2^n \times 2^n$ polynomial multiplier, with 32–bit coefficients. Throughout the paper, we will use the term $2^n$ to denote the $2^n \times 2^n$ polynomial multiplier. We do not utilize any special modulus, to achieve a generic and robust polynomial multiplier as we use Barrett’s reduction algorithm for coefficient arithmetic. Instead of the classical schoolbook method for polynomial multiplication, we utilized the NTT–based multiplication algorithm, as explained in Section 5.2.1.1 and described in Algorithm 9. It should be noted that Step 5 of Algorithm 9 is implemented by coefficient–wise 32–bit modular multiplications.

**Algorithm 9: NTT–based $2^n$ polynomial multiplication**

**input**: $A(x) = A_0 + A_1 x + \cdots + A_{2^n - 1} x^{2^n - 1}$, $B(x) = B_0 + B_1 x + \cdots + B_{2^n - 1} x^{2^n - 1}$, $p$

**output**: $C(x) = A(x) \times B(x)$

1. $NTT_A(x) \leftarrow$ NTT of polynomial $A(x)$;
2. $NTT_B(x) \leftarrow$ NTT of polynomial $B(x)$;
3. $NTT_C(x) \leftarrow$ Inner products of polynomials $NTT_A(x)$ and $NTT_B(x)$;
4. $T(x) \leftarrow$ Inverse NTT of polynomial $NTT_C(x)$;
5. $C(x) \leftarrow T(x) \times ((2^n)^{-1} \mod p)$;

#### 5.2.3.1 NTT Operation

**NTT Algorithm.** We apply the NTT operation on a polynomial $A(x)$ of degree $2^n – 1$ over $\mathbb{Z}_p[x]/(\Phi(x))$. Since the result of the NTT–based multiplication will be of
degree $2^{(n+1)}$, we need to zero-pad the polynomial $A(x)$ to make it also a polynomial of degree $2^{(n+1)}$ as follows $A(x) = \sum_{j=0}^{2^n-1} A_j \cdot x^j + \sum_{j=2^n}^{2^{(n+1)}-1} 0 \cdot x^j$. When we apply the NTT transform on $A(x)$, the resulting polynomial is $A(x) = \sum_{i=0}^{2^{(n+1)}-1} A_i \cdot x^i$, where the coefficients $A_i \in \mathbb{Z}_p$ are defined as $A_i = \sum_{j=0}^{2^{(n+1)}-1} A_j \cdot w^{ij} \mod p$, and $w \in \mathbb{Z}_p$ is referred as the twiddle factor. Since the size of the NTT operation is actually $2^{(n+1)}$, we need to choose a twiddle factor $w$ which satisfies the property $w^{2^{(n+1)}} \equiv 1 \mod p$ and $\forall i < 2^{(n+1)} w^i \neq 1 \mod p$.

To achieve fast NTT operations, we utilize the Cooley–Tukey approach, as explained in Section 5.2.1.1. Cooley–Tukey approach works by splitting up the NTT–transform into two parts, performing the NTT operation on the smaller parts, and performing a final reconstruction to combine the results of the two half–size NTT transform results into a full–sized NTT operation. If the NTT operation is defined as:

$$A_i = \sum_{j=0}^{2^{(n+1)}-1} A_j \cdot w^{ij} \mod p,$$

we can split up this operation as follows

$$A_i = \sum_{j=0}^{2^n-1} A_{2j} \cdot w^{i(2j)} \mod p + \sum_{j=0}^{2^n-1} A_{2j+1} \cdot w^{i(2j+1)} \mod p,$$

which can also be expressed as $A_i = E_i + w^iO_i$, where $E_i$ and $O_i$ represent the $i^{th}$ coefficients of the $2^n$ NTT operation on the even and odd coefficients of the polynomial $A(x)$, respectively. It is important to note that if the twiddle factor of the $2^{(n+1)}$ NTT operation is $w$, the twiddle factor of the smaller $2^n$ operation will be $w^2$. Because of the periodicity of the NTT operation, we know that $E_{i+2^n} = E_i$ and $O_{i+2^n} = O_i$. Therefore, we have $A_i = E_i + w^iO_i$ for $0 \leq i < 2^n$ and $A_i = E_{i-2^n} + w^iO_{i-2^n}$ for $2^n \leq i < 2^{(n+1)}$. For the twiddle factor, it holds that $w^{i+2^n} = w^i \cdot w^{2^n} = -w^i$. Consequently, we can achieve a full $2^{(n+1)}$ NTT operation.
with two small $2^n$ NTT operations utilizing the following reconstruction operation

$$A_i = E_i + w^iO_i,$$

$$A_{i+2^n} = E_i - w^iO_i. \tag{5.1}$$

The reconstruction operation is performed iteratively over very large number of coefficients. To better explain the iterative Cooley–Tukey approach, we would like to give a toy example of the NTT operation. First, we show the smallest NTT-transform circuit used in our design, which is shown in Figure 5.14a. Here, the NTT operation is applied over a polynomial of degree 1, with $w^2 \equiv 1 \mod p$. Therefore, the two outputs of the circuit are $A + B$ and $A + wB \equiv A - B \mod p$. Utilizing the $2 \times 2$ NTT circuit, we can perform a $4 \times 4$ NTT operation as shown in Figure 5.14b. Here, since we are constructing a $4 \times 4$ NTT circuit, we have $w^4 \equiv 1 \mod p$.

In a similar fashion, we can achieve an $8 \times 8$ NTT operation utilizing two $4 \times 4$ NTT operations, as shown in Figure 5.15. Here, since we are constructing an $8 \times 8$ NTT circuit, we have $w^8 \equiv 1 \mod p$. Also in Figure 5.15, we can see that if the twiddle factor of the $8 \times 8$ NTT operation is $w$, the twiddle factor of the $4 \times 4$ NTT operation is $w^2$. The overall architecture for iterative computation of NTT is shown in Figure 5.14.
in Figure 5.16. Note that, in a full $2^{(n+1)}$ NTT circuit, the twiddle factor $w^{16484}$ is used in $8 \times 8$ NTT circuits.

**Coefficient Multiplication and Accumulation.** In order to parallelize multiplication and accumulation operations we utilize $3 \cdot K$ DSP units to achieve $K$ modular multiplications in parallel, with a 4–cycle throughput, where $K$ is a design parameter that depends on the number of available DSP units in the target architecture. In our design, $K$ is chosen as a power of 2.

To be able to feed the DSP units with correct polynomial coefficients during multiplication cycles, we utilize $K$ separate Block RAMs (BRAM) to store the
Figure 5.16: NTT Circuit
polynomial coefficients as shown in Figure 5.17 (e.g. $K = 128$). The algorithm used to access the polynomial coefficients in parallel is described in Algorithm 10. The algorithm takes the BRAM content (i.e., the coefficients of $A(x)$), the degree $N = 2^n$, the current level $m$, and the number of modular multipliers $K = 2^\kappa$ as input, and generates the indexes in a parallel manner. Every four clock cycles, we try to feed modular multipliers the number of coefficients which is as close to $K$ as possible. Ideally, it is desirable to perform exactly $K$ modular multiplications in parallel, which is not possible due to the access pattern to the powers of $w$. Algorithm 10, on the other hand, achieves a good utilization of modular multiplication units.

For level $m$, we use the $2^m \times 2^m$ NTT circuit. The coefficients are arranged in $2^m \times 2^m$ blocks. For example when $K = 256$, for the first level of the NTT operation, where $m = 2$, we need to multiply every 4th coefficient of the polynomial with $w_2 = w^{16384}$. Since the coefficients are perfectly dispersed, we can read 256 coefficients to feed the 256 multipliers in four clock cycles. This is perfect as the throughput of our multipliers are also four cycles. When the multiplication operations are complete, with an offset of 19 cycles (four clock cycles are for the warm up of the pipeline whereas 15 clock cycles are the tail cycles necessary in a pipelined design to finish the last operation), the results are written back to the same address of the RAM block as the one the coefficients are read from.

We provide formulae for the number of multiplications in each level and an estimate of the number of clock cycles needed for their computation in our architecture. Suppose $N = 2^n$ and $K = 2^\kappa (n > \kappa)$ are the number of coefficients in our polynomial and the number of modulo multipliers in our target device, respectively. The coefficients are stored in BRAMs, with a word size of 32 bits and an address length of 10 bits (1024 coefficients per BRAM). For ideal case, the number of modular multipliers should be 4 times the number of BRAMS required to store a single poly-
Algorithm 10: Parallel access to polynomial coefficients

input: \( A(x) = A_0 + A_1 x + \ldots + A_{2N-1} x^{2N-1} \), \( n \), \( m \), and \( \kappa < n \)
output: \( B_i[j] \)

2 \( m\text{Cnt} \leftarrow 2^{m-1} - 1 \); /* number of multiplications in a block */
4 \( b\text{Size} \leftarrow 2^m \); /* size of a block */
6 \( \text{BRAMCnt} \leftarrow 2^{\kappa-2} \); /* number of BRAMs */
8 if \( b\text{Size} \leq 2^{\kappa-2} \) then
10 \( \text{for } t = 0 \text{ to } 1024 \text{ do} \)
12 \( \text{for } i = 0 \text{ to } \text{BRAMCnt} \text{ do in parallel} \)
14 \( \text{for } j = i + b\text{Size} - m\text{Cnt} \text{ to } i + b\text{Size} \text{ do} \)
16 \( \text{for } k = 0 \text{ to } 3 \text{ do} \)
18 \( \text{Access } \text{BRAM}_j[t + 2k] ; \)
20 \( j \leftarrow j + 1 ; \)
22 \( i \leftarrow i + b\text{Size} ; \)
24 \( t \leftarrow t + 8 ; \)
else
26 \( \text{for } i = 0 \text{ to } \text{BRAMCnt} \text{ do in parallel} \)
28 \( \text{for } j = 0 \text{ to } 1024 \text{ do} \)
30 \( \text{for } k = 2^{m-\kappa+1} \text{ to } 2^{m-\kappa+2} \text{ do} \)
32 \( \text{Access } \text{BRAM}_i[k + j] ; \)
34 \( j \leftarrow j + 2^{m-\kappa+2} ; \)
36 \( i \leftarrow i + 1 ; \)
Figure 5.17: The architecture for NTT transformation of a polynomial of degree $N$ over $F_p$, where $\lceil \log_2 p \rceil = 32$.

The formula for the number of multiplications for the level $m > 1$ can be given as $M = 2^{n+1-m} \cdot (2^{m-1} - 1)$. Also, using $K = 2^\kappa$ multipliers, the number of clock cycles to compute all multiplications in a given level $1 < m \leq n + 1$ can be
formulated as

\[
CC_m = \begin{cases} 
4 + 4 \cdot \left\lfloor \frac{\mathcal{M}}{\alpha \cdot \lceil K/\alpha \rceil} \right\rfloor + 15 & \kappa \geq m \\
4 + 4 \cdot (\frac{\beta}{K} + 1) \cdot 2^{n+1-m} + 15 & \kappa < m, 
\end{cases}
\]

where \( \alpha = 2^{\kappa-m} \cdot (2^{m-1} - 1) \) and \( \beta = 2^{m-1} - 2^\kappa \). In the formula, the first (4) and the last terms (15) account for the warm up and the tail cycles.

As an example, Table 5.8 shows the number of multiplication operations required for each stage of the iterative Cooley-Tukey NTT operation, for a 32768-coefficient (64K-point) NTT operation, when the number of modular multipliers is 256. (i.e., \( N = 2^{15} \) and \( K = 256 \)).

As mentioned before, the modulo multipliers are not always fully utilized during the NTT computation. For example when \( K = 2^8 \) and \( N = 2^{15} \), for \( m = 2 \), we have to read every 4\textsuperscript{th} coefficient from the BRAMs. Because the coefficients are perfectly dispersed throughout the 64 BRAMS, we can only read 16 \( \cdot \) 2 = 32 coefficients every clock cycle, which yields a number of 128 concurrent multiplications every four clock cycles. Consequently, we can finish all the modular multiplications in the first level in 4 + 128 \( \cdot \) 4 + 15 = 531 clock cycles. Since we can use half the modular multipliers, we achieve half utilization in the first level. However, when \( m = 3 \), we have to read every 6\textsuperscript{th}, 7\textsuperscript{th} and 8\textsuperscript{th} out of every 8 coefficients. We can read 24 \( \cdot \) 2 = 48 coefficients every clock cycle from the BRAMs. This means we can only utilize 192 out of 25 modular multipliers since the irregularity of the access to the polynomial coefficients. This, naturally, results in a slightly low utilization. However, since we can read 2 coefficients from each BRAM every clock cycle, we are at almost perfect utilization, resulting in 4 + 128 \( \cdot \) 4 + 15 = 531 clock cycles for this and the rest of
Table 5.7: Powers of \( w \) needed in different levels of NTT circuit

<table>
<thead>
<tr>
<th>Level ((m))</th>
<th>Block size</th>
<th>powers of ( w )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4 \times 4</td>
<td>( w^{2^1} )</td>
</tr>
<tr>
<td>3</td>
<td>8 \times 8</td>
<td>( w^{2^3}, w^{2^2 \cdot 2}, w^{3 \cdot 2} )</td>
</tr>
<tr>
<td>4</td>
<td>16 \times 16</td>
<td>( w^{2^4}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>5</td>
<td>32 \times 32</td>
<td>( w^{2^5}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>6</td>
<td>64 \times 64</td>
<td>( w^{2^6}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>7</td>
<td>128 \times 128</td>
<td>( w^{2^7}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>8</td>
<td>256 \times 256</td>
<td>( w^{2^8}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>9</td>
<td>512 \times 512</td>
<td>( w^{2^9}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>10</td>
<td>1024 \times 1024</td>
<td>( w^{2^{10}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>11</td>
<td>2048 \times 2048</td>
<td>( w^{2^{11}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>12</td>
<td>4096 \times 4096</td>
<td>( w^{2^{12}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>13</td>
<td>8192 \times 8192</td>
<td>( w^{2^{13}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>14</td>
<td>16384 \times 16384</td>
<td>( w^{2^{14}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>15</td>
<td>32768 \times 32768</td>
<td>( w^{2^{15}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
<tr>
<td>16</td>
<td>65536 \times 65536</td>
<td>( w^{2^{16}}, w^{2^2 \cdot 2}, \ldots, w^{(2^3 - 1) \cdot 2^1} )</td>
</tr>
</tbody>
</table>

Since the operands of the both operations are accessed in a regular manner, the number of clock cycles spent on modular additions and subtractions are calculated as \( \frac{2^{(n+1) \cdot (n+1)}}{2^r} \), when there are \( 2^r \) modular adders and \( 2^r \) subtracers.

**w Generation.** Theoretically, we need an \( N \)-th root of unity in \( F_p \) for NTT of polynomials of degree \( N \). Due to the polynomial padding in our case, we need an \( 2N \)-th root of unity \( w \in F_p \) such that \( w^{2^{(n+1)}} = 1 \mod p \) and \( \forall i < 2^{(n+1)}, w^i \neq 1 \mod p \).

In every level of the NTT circuit, we use different powers of \( w \). For the level \( m \), where we use the \( 2^m \times 2^m \) butterfly circuit and the coefficients are arranged in \( 2^m \times 2^m \) blocks, we need \( w_1^m, w_2^m, \ldots, w^{2n-1}_m \) where \( w_m = w^{2^{16-m}} \). For instance, \( w^{2^{14}} \) is used in every multiplication in the \( 4 \times 4 \) butterfly circuit while \( w^{2^{13}}, w^{2 \cdot 2^{13}}, w^{3 \cdot 2^{13}} \) are used in the multiplications in \( 8 \times 8 \) butterfly circuit.
For the powers of $w$ that are used in different levels of computation for a $2^{16}$–point NTT operation, see Table 5.7. In summary, for the $2^{16}$–point NTT we need $2^{15} - 1 = 32767$ powers of $w$; namely $w, w^2, w^3, \ldots, w^{32767}$. In case of $2^{14}$ polynomial multiplier we require up to $2^{15}$–point NTT arithmetic which we only need $2^{14} - 1 = 16383$ coefficients for powers of $w$, e.g. $w^2, w^2 \cdot 2, w^2 \cdot 3, \ldots, w^2 \cdot 16383$. We precompute and store these powers of $w$ in block RAMs in a distributed fashion similar to the coefficients of the polynomials as illustrated in Figure 5.17. Alternatively, the powers of $w$ can be computed on-the-fly for area efficiency. However, since we have sufficient number of block RAMs in the target reconfigurable device, we prefer the precomputation approach.

**Reconstruction.** Once we are done with the multiplications, we utilize 64 modular adders and 64 modular subtracters to realize the addition and subtraction operations.

### Table 5.8: Details of NTT computation in our architecture for 32768 coefficients and 256 multiplier units.

<table>
<thead>
<tr>
<th>NTT blocks</th>
<th>number of blocks</th>
<th>number of modular multiplications</th>
<th>number of clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>$4 \times 4$</td>
<td>16384</td>
<td>16384</td>
<td>275</td>
</tr>
<tr>
<td>$8 \times 8$</td>
<td>8192</td>
<td>24576</td>
<td></td>
</tr>
<tr>
<td>$16 \times 16$</td>
<td>4096</td>
<td>28672</td>
<td></td>
</tr>
<tr>
<td>$32 \times 32$</td>
<td>2048</td>
<td>30720</td>
<td></td>
</tr>
<tr>
<td>$64 \times 64$</td>
<td>1024</td>
<td>31744</td>
<td></td>
</tr>
<tr>
<td>$128 \times 128$</td>
<td>512</td>
<td>32256</td>
<td></td>
</tr>
<tr>
<td>$256 \times 256$</td>
<td>256</td>
<td>32512</td>
<td></td>
</tr>
<tr>
<td>$512 \times 512$</td>
<td>128</td>
<td>32640</td>
<td>531</td>
</tr>
<tr>
<td>$1024 \times 1024$</td>
<td>64</td>
<td>32704</td>
<td></td>
</tr>
<tr>
<td>$2048 \times 2048$</td>
<td>32</td>
<td>32736</td>
<td></td>
</tr>
<tr>
<td>$4096 \times 4096$</td>
<td>16</td>
<td>32752</td>
<td></td>
</tr>
<tr>
<td>$8192 \times 8192$</td>
<td>8</td>
<td>32760</td>
<td></td>
</tr>
<tr>
<td>$16384 \times 16384$</td>
<td>4</td>
<td>32764</td>
<td></td>
</tr>
<tr>
<td>$32768 \times 32768$</td>
<td>2</td>
<td>32766</td>
<td></td>
</tr>
<tr>
<td>$65536 \times 65536$</td>
<td>1</td>
<td>32767</td>
<td></td>
</tr>
</tbody>
</table>

**Total clock cycles** | **7709**
as shown in Equation 5.1.

5.2.3.2 Inner Multiplication

Inner multiplication of two $2^n$ polynomials is trivial for our hardware design. We can load 256 coefficients from each polynomial every 4 cycles and feed the multipliers, without increasing the 4-cycle throughput. For a $2^n$ polynomial inner multiplication we spend $2^{(n+1)} \cdot 4/256 + 15$ clock cycles.

5.2.3.3 Inverse NTT

The Inverse NTT operation is identical to the NTT operation, except that instead of the twiddle factor $w$, we use the twiddle factor $w_i = w^{-1} \mod p$. The precomputed twiddle factors of the inverse NTT are stored in the same block RAMs as the forward NTT twiddle factors, with an address offset. Therefore, the same control block can be utilized with a simple address change for the $w$ coefficients for the inverse NTT operation.

5.2.3.4 Final Scaling

Final scaling is similar to the inner multiplication phase. We load each coefficient of the resulting polynomial, and multiply them with the precomputed scaling factor. Similar to the inner multiplication phase, we can load 256 coefficients from the resulting polynomial in 4 cycles cycle and feed the multipliers, without increasing the 4-cycle throughput. For a $2^n$ polynomial final scaling operation, we spend $2^{(n+1)} \cdot 4/256 + 15$ clock cycles.
5.2.4 Implementation Results

We developed the architecture described in the previous section into Verilog modules and synthesized it using Xilinx Vivado tool for the Virtex 7 XC7VX690T FPGA family. The synthesis results are summarized in Table 5.9. We synthesized the design and achieved an operating frequency of 250 MHz for multiplication of polynomials of degrees $N = 16,384$ for Prince and $N = 32,768$ for AES with a small word size of $\log p = 32$ bit. In Table 5.10 we summarize the timing results of the synthesized small word size polynomial multiplier.

Although we can scale our architecture for larger parameters, it becomes hard to synthesize, since we are using 50 percent of the LUTs already. Another problem is that with larger hardware it is harder to do the routing because of the butterfly circuit mapping at each level. Also, it becomes harder to fit all the necessary components, i.e. polynomials, powers of $\omega$ and resulted polynomial in the FPGA. Therefore, it becomes impossible to process a multiplication without extra I/O transactions when computing the NTT conversions.

Table 5.9: Virtex-7 XC7VX690T device utilization of the multiplier

<table>
<thead>
<tr>
<th>$N$ = (16,384/32,768)</th>
<th>Total</th>
<th>Used</th>
<th>Used (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice LUTs</td>
<td>433,200</td>
<td>219,192</td>
<td>50.59</td>
</tr>
<tr>
<td>Slice Registers</td>
<td>866,400</td>
<td>90,789</td>
<td>10.47</td>
</tr>
<tr>
<td>RAMB36E1</td>
<td>1470</td>
<td>193</td>
<td>13.12</td>
</tr>
<tr>
<td>DSP48E1</td>
<td>3600</td>
<td>768</td>
<td>21.33</td>
</tr>
</tbody>
</table>

The FPGA multiplier is used to process each component of the CRT representation of our large coefficient ciphertexts with $\log q = 500$ bits for Prince and $\log q = 1271$ bits for AES implementation. In fact we keep all ciphertexts in CRT

---

9We use the same hardware architecture for both applications. The only difference is that compared to $N = 16,384$ case, the architecture is used almost twice many times in $N = 32,768$. 

162
representation and only compute the polynomial form when absolutely necessary, e.g. for parity correction during modulus switching and before relinearization. We assume any data sent from the PC through the PCIe interface to the FPGA is stored in onboard BRAM units.

**CRT Computation Cost.** To facilitate efficient computation of multiplication and relinearization operations we use a series of equal sized prime numbers to construct a CRT conversion. In fact, we chose the primes \( p_i \)'s such that \( q = \prod_{i=0}^{l} p_i \). During the levels of homomorphic evaluation, this representation allows us to easily switch modulus by simply dropping the last \( p_i \) following by a parity correction. Also, since we have an RNS representation on the coefficients we no longer need to reduce by \( q \). This also eliminates the need to consider any overflow conditions. Thus, \( l = \log(q)/\log(p_i) \) is 25 and 41 for Prince and AES implementations, respectively.

We efficiently compute the CRT residue in software on the CPU for each polynomial coefficient as follows:

- Precompute and store \( t_k = 2^{64-k} \pmod{p_i} \) where \( k \in [0, \lceil \log(q/64) - 1 \rceil] \).
- Given a coefficient of \( c \), we divide it into 64-bit blocks as \( c = \{\ldots, w_k, \ldots, w_0\} \).
- We compute the CRT result by evaluating \( \sum t_k \cdot w_k \pmod{p_i} \) iteratively.

The CRT computation cost for 41 primes \( p_i \) per ciphertext polynomial is in the order of 89 ms and for 25 primes \( p_i \) per ciphertext polynomial is in the order of
14.5 ms on the CPU. The CRT inverse is similarly computed (with the addition of a word carry) before each modulus switching operation at essentially the same cost.

**Communication Cost.** The PCIe bus is only used for transactions of input/output values, NTT constants and transport of evaluation keys to the FPGA board. With 8 lanes each capable of supporting 8 GT/s transport speed the PCIe is capable to transmit a 1 MB ciphertext in about 0.13 ms. Note that the NTT parameters used during multiplication also need to be transported since we do not have enough room in the BRAM components to keep them permanently. We have two cases to consider:

- **Multiplication:** We transport two polynomials of 5 MB / 1 MB each along with the NTT parameters of 5 MB / 1 MB and receive a polynomial of 10 MB / 2 MB, which costs about 3.25 ms / 0.65 ms per multiplication for AES/Prince implementation.

- **Relinearization:** We need to transport the ciphertext we want to relinearize, the NTT parameters and a set of \( \frac{\log(q)}{16} \approx 80 / \frac{\log(q)}{16} \approx 32 \) evaluation keys (ciphertexts), where a window size of 16-bit is used, resulting in a 52 ms / 10 ms delay for AES/Prince implementation.

**Multiplication Cost.** We compute the product of two polynomials with coefficients of size \( \log(p) = 32 \) bits using 256 modular multipliers in 12720/6120 cycles, which translates to 152 \( \mu s \) / 73.4 \( \mu s \) for AES/Prince implementation. This figure is comprised of two NTT and one inverse NTT operations and one inner product computation. The addition of I/O transactions increase the timing by 79 \( \mu s \) / 26 \( \mu s \) for AES/Prince implementations. The latency of large polynomial multiplication may be broken down as follows:
• Cost of small coefficient polynomial multiplications is $41 \cdot 152 \, \mu s = 6.25 \, ms$ for AES and $25 \cdot 73.4 \, \mu s = 1.84 \, ms$ for Prince.

• The PCIe transaction of the two input polynomials, the NTT coefficients and the double sized output polynomial is $3.25 \, ms / 0.64 \, ms$ for AES/Prince implementation.

Thus, the total latency for large polynomial multiplication in the CRT representation is computed in $9.51 \, ms$ and $2.48 \, ms$ for AES and Prince implementations respectively.

**Polynomial Modular Reduction.** Since all operations are computed in a polynomial ring with a characteristic polynomial as modulus without any special structure, we use Barrett’s reduction technique to perform the reductions. Note that precomputing the constant polynomial $x^{2N}/\Phi(x)$ (truncated division) in the CRT representation we do not need to compute any CRT or inverse CRT operations during modular reduction. Thus we can compute the reduction using two product operations in about $19 \, ms$ and $4.9 \, ms$ for AES and Prince implementations respectively.

**Modulus Switching.** We realize the modulus switching operation by dropping the last CRT coefficient followed by parity correction. To compute the parity of the cut polynomial we need to compute an inverse CRT operation. The following parity matching and correction step takes negligible time. Therefore, modulus switching can be realized using one inverse CRT computation in $89 \, ms$ and $14.5 \, ms$ for AES and Prince implementations respectively.

**Relinearization Cost.** To relinearize a ciphertext polynomial

• We need to convert the ciphertext polynomial coefficients into integer representation using one inverse CRT operation, which takes $89 \, ms / 14.5 \, ms$ for
AES/Prince implementation.

- The evaluation keys are kept in NTT representation, therefore we only need to compute two NTT operations for one operand and the result. For \( l = \frac{41}{25} \) primes and \( \frac{\log(q)}{16} \approx \frac{80}{32} \) products the NTT operations take 331 ms / 38 ms for AES/Prince implementation.

- We need to transport the ciphertext, the NTT parameters and \( \frac{80}{32} \) evaluation keys (ciphertexts) resulting in a 52 ms / 4 ms delay for AES/Prince implementation.

- The summation of the partial products takes negligible time compared to the multiplications and the PCIe communication cost.

Then, the total relinearization operation takes 526 ms and 61.2 ms for AES and Prince implementation respectively. With the current implementation, the actual NTT computations still dominate over the other sources of latency such as PCIe communication latency and the CRT computations. However, if the design is further optimized, e.g. by increasing the number of processing units on the FPGA or by building custom support for CRT operations on the FPGA, then the PCIe communication overhead will become more dominant. The timing results are summarized in Table 5.11.

5.2.5 Comparison

To understand the improvement gained by adding custom hardware support in leveled homomorphic evaluation of a deep circuit, we estimate the homomorphic evaluation time for the AES and Prince circuits and compare it with the software implementations (Sections 4.1, 4.2) and GPU implementations by Wei et al [82, 89].
Table 5.11: Primitive operation timings including I/O transactions.

<table>
<thead>
<tr>
<th></th>
<th>AES Timings (ms)</th>
<th>Prince Timings (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRT</td>
<td>89</td>
<td>14.5</td>
</tr>
<tr>
<td>Multiplication</td>
<td>9.51</td>
<td>2.48</td>
</tr>
<tr>
<td>NTT conversions</td>
<td>6.25</td>
<td>1.8</td>
</tr>
<tr>
<td>PCIe cost</td>
<td>3.26</td>
<td>0.64</td>
</tr>
<tr>
<td>Modular Reduction</td>
<td>19</td>
<td>4.95</td>
</tr>
<tr>
<td>Modulus Switch</td>
<td>89</td>
<td>14.5</td>
</tr>
<tr>
<td>Relinearization</td>
<td>526</td>
<td>61.2</td>
</tr>
<tr>
<td>CRT conversions</td>
<td>89</td>
<td>14.5</td>
</tr>
<tr>
<td>NTT conversions</td>
<td>331</td>
<td>38.2</td>
</tr>
<tr>
<td>PCIe cost</td>
<td>52</td>
<td>4</td>
</tr>
</tbody>
</table>

**Homomorphic AES evaluation.** We implemented the depth 40 AES circuit following the approach in Section 4.1. The tower field based AES SBox evaluation is completed using 18 Relinearization operations and thus 2,880 Relinearizations are needed for the full AES. The AES circuit evaluation requires 5760 modular multiplications. During the evaluation we also compute 6080 modulus switching operations. This results in a total AES evaluation time of 15 minutes. Note that during the homomorphic evaluation with each new level the operands shrink linearly with the levels thereby increasing the speed. We conservatively account for this effect by dividing the evaluation time by half. With 2048 message slots, the amortized AES evaluation time becomes 439 ms.

We have also modified homomorphic AES evaluation code to compute relinearization with 16-bits windows (originally single bit). This simple optimization dramatically reduces the evaluation key size and speeds up the relinearization. The results are given in Table 5.12. We also included the GPU optimized implementation by Dai et al. [82] on an NVIDIA GeForce GTX 680. With custom hardware assistance we obtain a significant speedups in both multiplication and relineariza-
Table 5.12: Comparison of multiplication, relinearization times and AES estimate

<table>
<thead>
<tr>
<th></th>
<th>Mul Speedup (ms)</th>
<th>Relin Speedup (s)</th>
<th>AES Speedup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Section 4.1)</td>
<td>970 1× 103 1× 55 1×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU [82]</td>
<td>340 2.8× 8.97 11.5× 7.3 7.5×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU (16-bit)</td>
<td>970 1× 6.5 16× 12.6 4.4×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>9.5 102× 0.53 195× 0.44 125×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

tion operations. The estimated AES block evaluation is also improved significantly where some of the efficiency is lost to the PC to FPGA communication and CRT computation latencies.

**Homomorphic Prince evaluation.** We implemented the depth 24 Prince circuit following the approach in Section 4.2. The algorithm is completed using 1152 relinearizations, 1920 multiplications, 3072 modular reductions and 2688 modular switch operations. An important thing to note that as we did in AES implementation, we divide the evaluation time by half. The reason is that since during the homomorphic evaluation with each new level, the operands shrink linearly so the evaluation speed increases linearly. These results in a total time of 53 seconds and an amortized time of 52 ms with batching 1024 messages. Here in Table 5.13, we compare the

<table>
<thead>
<tr>
<th></th>
<th>Mul Speedup (ms)</th>
<th>Relin Speedup (s)</th>
<th>Prince Speedup (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (Section 4.2)</td>
<td>180 1× 10.9 1× 3.3 1×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU [82]</td>
<td>63 2.85× 0.89 12.3× 1.28 2.58×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU [89]</td>
<td>n/a n/a n/a n/a 0.032 103×</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ours</td>
<td>2.5 72× 0.06 181× 0.05 66×</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

results of homomorphic Prince implementations. Also, we include the homomorphic Prince implementations of Dai et al. [82, 89] on GPUs which are significantly faster compared to the CPU implementation.
Chapter 6

Conclusion

In this dissertation, we improve the existing FHE schemes to bring them closer to practice. In this direction, we proposed new FHE schemes that solve specific performance bottlenecks and further implemented various optimizations.

Significant memory requirement was one of the obstacles in the way towards practicality. We solved the problem by introducing a special ring structure that reduces the evaluation key size significantly. Another approach we pursued is to get rid of the evaluation keys completely by applying the flattening technique on our modified version of NTRU. We achieved competitive speeds: multiplication in 24.4 msec to support 5 levels and 34.3 msec for 30 levels.

To assess the performance of the DHS scheme, we homomorphically evaluated the full 10-round AES circuit in 29 hours with 2,048 message slots yielding a 51 sec per AES block evaluation making it 47 times faster than the generic bit-sliced implementation, 5.8 times faster than the AES customized byte-sliced BGV implementation by Gentry, Halevi and Smart.

Also, we presented a customized implementation of the lightweight block cipher Prince using a leveled fully homomorphic encryption scheme based on NTRU. For
this we surveyed lightweight block ciphers and analyzed them with respect to a new metric: circuit depth. Our analysis determined that the Prince block cipher is the most suitable for homomorphic evaluation as it can be implemented using a circuit of depth 24. We developed an optimized shallow circuit implementation of Prince, which yielded an amortized 3.3 seconds per block evaluation running time, one to two orders of magnitude faster than our homomorphic AES evaluation and the one proposed in [23].

In order to increase the computation speeds, we proposed hardware accelerators. First, we took initial steps to remedy the efficiency bottleneck of FHE schemes by introducing the first custom FHE architecture. For this we introduced a novel large integer modular multiplier design realizing the Schönhage-Strassen algorithm and Barrett’s reduction in hardware. Using this core we implemented the Gentry-Halevi FHE primitives, e.g. encryption, decryption, and recryption. Among these primitives we managed to improve the efficiency of the challenging recryption operation to the point where we are surpassing its software implementation performance on a high end GPU processor at a fraction of the footprint.

Eventually, we presented a custom hardware design to address the performance bottleneck in leveled somewhat homomorphic encryption evaluations. For this, we design a large NTT based multiplier, which is able to compute large degree polynomial multiplications using the Cooley-Tukey FTT technique. We extend the support of the custom core to be capable of multiplying large degree polynomials with large coefficients by using CRT representation on the coefficients. Using numerous techniques the design is highly optimized to speed up the NTT computations, and to reduce the burden on the PC/FPGA interface. Our design achieves remarkable improvements in speed of modular multiplication and relinearization of the DHS scheme compared to previous software implementations. In order to show the ac-
celeration that our architecture may provide, we estimated the homomorphic AES and Prince evaluation performances and determined a speedup of about 28 and 66 times respectively.

In summary, we presented various techniques to improve the performance bottleneck of the FHE schemes. We started with a half minute runtime for a single AND operation and achieved single order of magnitude speed up in every year. Currently, a single AND operation takes milliseconds which summarizes the significant improvements achieved over the years. Although the recent schemes have better performance, any FHE application that requires instant time response is still yet far from being practical. Only possible applications with FHE’s are ones that does offline computation where the response is not time critic. Such possible applications are auctions, certain financial data transactions, genomic data analysis and etc. In case of the time critical applications, we believe that if the current trend on the optimizations is preserved, we will be able to design practical FHE applications in the following years.
Bibliography


[54] Deukjo Hong, Jaechul Sung, Seokhie Hong, Jongin Lim, Sangjin Lee, Bon-Seok Koo, Changhoon Lee, Donghoon Chang, Jesang Lee, Kitaee Jeong, Hyun Kim,


[69] Luis Carlos Coronado García. Can schönhage multiplication speed up the rsa decryption or encryption? 2005.


