7-2023

Applying Homomorphic Encryption to a Cross Domain Problem

Cheyenne Dailey
cjd9104@rit.edu

Applying Homomorphic Encryption to a Cross Domain Problem

Cheyenne Dailey

Applying Homomorphic Encryption to a Cross Domain Problem

CHEYENNE DAILEY

July 2023

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Applying Homomorphic Encryption to a Cross Domain Problem

CHEYENNE DAILEY

Committee Approval:

Dr. Marcin Łukowiak *Advisor* Date
RIT Department of Computer Engineering

Dr. Stanisław Radziszowski Date
RIT Department of Computer Science

Dr. Sonia Lopez Alarcon Date
RIT Department of Computer Engineering

Dr. Michael Kurdziel Date
L3 Harris Technology

Acknowledgments

I would not have been able to accomplish all I have, obtaining a Master's degree, without the support of many individuals in my life. First and foremost, I would like to thank my family. My parents, April and Ted, for pushing me to be the best I can be and supporting me through all the ups and downs. To my sister, Sierra, who constantly pushed me to keep going and get it done even when I wanted to give up.

I would like to thank my friends for their time and support through long hours of coding. Sidney Davis, for listening to long winded rational explanation and offering help when coding became difficult. Anna Nicolais, for the support and friendship when things got tough. And to all other friends, who supported me through college and were willing to listen to the long rants of how my progress was going.

Lastly, I would like to thank my advisors and committee members. Dr. Kurdziel and Dr. Lopez Alarcon for their positive remarks. Dr. Radziszowski, for always pushing me to put out my best work and explore all possibilities. And a special thanks to Dr. Lukowiak, for putting up with me for longer than originally intended and making sure that I was able to get to where I am today.

Abstract

The Cross Domain Problem (CDP) strives to ensure protected data transference across varying security domains. In order to accomplish this, a Cross Domain Solution (CDS) is needed. A common method to protect data is to focus on risk management between trusted parties; however, untrusted parties pose ongoing concern. The problem is determining a method that transfers classified data through various security domains without exposing any information to intermediary parties. Attempts to mitigate this problem have been made utilizing Homomorphic Encryption (HE), a type of encryption that allows for computations to be executed on encrypted data without needing to decrypt it. Research studies have demonstrated the feasibility of applying an HE scheme paired with a cipher to successfully create a CDS for untrusted parties.

By researching recent enhancements in the fields of homomorphic encryption, lightweight ciphers, and hybrid homomorphic ciphers a pair was found with the hope of practical main steam use has been achieved. The homomorphic scheme, BFV, has been around for many years with thorough testing and new optimizations applied. The cipher, Pasta, is a hybrid homomorphic cipher specifically catered to the application of homomorphic decryption. Together, a software test case was created that would mimic the required behavior needed to create a CDS.

The final implementation offered testing of homomorphic decryption with both 3-Round and 4- Round Pasta with acceptable speeds given the processing power available. Along with the rounds changing, size of key, plaintext, and multiplicative depth influenced overall performance. Verifying the usability post decryption, comparison of values at any index demonstrated the ability to search and compare specific plaintext or metadata values for viable information about transmission through encountered gateways. In both variations, the speed was favorable, proving to be at least 5 times faster than similar implementations.

Contents

Signature Sheet **i**

Acknowledgments **ii**

Abstract **iii**

Table of Contents **iv**

List of Figures **vii**

List of Tables **viii**

Acronyms **ix**

1 Introduction **2**

1.1 Motivation 2

1.2 This Work 3

2 Background **5**

2.1 Cross Domain Solution 5

2.2 Ciphers 6

2.2.1 Symmetric Key Ciphers 7

2.2.2 Asymmetric Key Ciphers 9

2.2.3 Lightweight Ciphers 9

2.3 Homomorphic Encryption 11

2.3.1 Partially Homomorphic Encryption 12

2.3.2 Somewhat Homomorphic Encryption 12

2.3.3 Fully Homomorphic Encryption 13

2.4 Hybrid Homomorphic Encryption 13

2.4.1 Hybrid Homomorphic Encryption Ciphers 14

3 Lightweight Ciphers **16**

3.1 NIST Lightweight Cryptography Finalists 16

3.1.1 ASCON 17

3.1.2 Elephant 17

3.1.3 GIFT-COFB 18

3.1.4 Grain-128AEAD 18
3.1.5 ISAP 19
3.1.6 PHOTON-Beetle 20
3.1.7 Romulus 20
3.1.8 SPARKLE 21
3.1.9 TinyJAMBU 21
3.1.10 Xoodyak 22
3.1.11 Performance of Competition Finalists 22
3.2 NIST Lightweight Cryptography Winner 23

4 Hybrid Homomorphic Encryption Ciphers 25
4.1 Fasta 25
4.2 Pasta 26

5 Fully Homomorphic Encryption Schemes 29
5.1 Basic Preliminaries and Notation 29
5.2 Brakerski-Gentry-Verauteren Scheme 30
5.2.1 Optimizations 31
5.3 Brakerski/Fan-Verauteren Scheme 32
5.3.1 Optimizations 33
5.4 Homomorphic Operation Examples 34
5.4.1 Homomorphic Addition 34
5.4.2 Homomorphic Subtraction 35
5.4.3 Homomorphic Multiplication 35
5.4.4 Homomorphic Sum 36
5.4.5 Homomorphic Rotation 36

6 Component Selection 37
6.1 Cipher Decision 37
6.1.1 Pursuit of Lightweight Cipher 37
6.1.2 Pursuit of Hybrid Homomorphic Encryption Cipher 38
6.2 Homomorphic Encryption Scheme Decision 39
6.2.1 Library 39
6.2.2 Scheme 39

7 Implementation 41
7.1 Hybrid Homomorphic Encryption 41

7.2 Hybrid Homomorphic Decryption Circuit 42
7.2.1 Linear Layer 44
7.2.2 S-Box 45
7.3 Cross Domain Solution Scenario 46
7.3.1 Application to the Cross Domain Problem 48
7.4 Additional Test Cases 51
7.4.1 Case 1 51
7.4.2 Case 2 51

8 Results **54**
8.1 3-Round Pasta Results 55
8.2 4-Round Pasta Results 57
8.3 Instance Comparison 58
8.4 Comparison to Previous Work 59

9 Conclusion **62**
9.1 Future Work 63

Bibliography **64**

A Source Code **69**

A Source Code **70**

A Source Code **71**

A Source Code **72**

A Source Code **73**

A Source Code **74**

A Source Code **75**

A Source Code **76**

A Source Code **77**

A Source Code **78**

List of Figures

2.1 Cross Domain Problem 6
2.2 Basic Encryption and Decryption Flow 6
2.3 Block Cipher 8
2.4 Stream Cipher 8
2.5 Authenticated Encryption with Associated Data 11
2.6 Hybrid Homomorphic Encryption Overview 14

3.1 ASCON Encryption and Decryption 23

4.1 High-level design of Fasta 26
4.2 Design of Pasta 28

7.1 HHE Encryption 42
7.2 CDS Use Case 47
7.3 High Level Design Breakdown of a CDS 49
7.4 Test Case 2 Design 53

List of Tables

3.1 ASCON Family Variants 17
3.2 Elephant Family Variants 18
3.3 GIFT-COFB Family Variants 18
3.4 Grain-128AEAD Family Variants 18
3.5 ISAP Family Variants 19
3.6 PHOTON-Beetle Family Variants 20
3.7 Romulus Family Variants 20
3.8 SPARKLE Family Variants 21
3.9 TinyJAMBU Family Variants 22
3.10 Xoodyak Family Variants 22

4.1 HHE Components with Pasta 27

7.1 Selected Classification Values 50

8.1 HHE Decryption Circuit with 3-Round PASTA Performance 55
8.2 HHE Decryption Circuit with 4-Round PASTA Performance 57
8.3 Parameter Comparison 60
8.4 Timing Comparison 60

Acronyms

AEAD Authenticated Encryption with Associated Data

AES Advanced Encryption Standard

BEHZ Bajard, Eynard, Hasan, and Zucca

BFV Brakerski/Fan-Vercauteren

BGV Brakerski-Gentry-Vaikuntanathan

CBC Cipher Block Chaining

CDP Cross Domain Problem

CDS Cross Domain Solution

CFB Cipher Feedback

CKKS Cheon, Kim, Kim and Song

COFB COmbined FeedBack

CRT Chinese Remainder Theorem

CTR Counter

DES Data Encryption Standard

EaM Encryption-and-MAC

ECB Electronic Code Book

EKE Encrypted Key Exchange

EtM Encrypt-then-MAC

FHE Fully Homomorphic Encryption

FHEW Fastest Homomorphic Encryption in the West

FPGA Field-Programmable Gate Array

GHS Gentry-Halevi-Smart

HE Homomorphic Encryption

HHE Hybrid Homomorphic Encryption

HPS Halevi, Polyakov, and Shoup

HTTPS Hypertext Transport Protocol Secure

KEP Key Encryption Protocol

LWC Lightweight Cipher

LWE Learning with Errors

MAC Message Authentication Code

MtE MAC-then-Encryption

NIST National Institute of Standards and Technology

OFB Output Feedback

PHE Partially Homomorphic Encryption

RLWE Ring Learning with Errors

RNS Residue Number System

SFTP Secure File Transfer Protocol

SPN Subsitution-Permutation Network

SWHE Somewhat Homomorphic Encryption

TCP Transmission Control Protocol

TFHE Fast Fully Homomorphic Encryption over the Torus

WSL Windows Subsystem for Linux

YASHE Yet Another Somewhat Homomorphic Encryption

Chapter 1

Introduction

1.1 Motivation

Today, digital data is everywhere and used by almost everyone. Smart phones, computers and the IoT allow for instant access and transfer of data between parties. During data transfers, cryptography is implemented to convert data into a format that is unreadable for unauthorized parties. Most users are unaware of these transmission protocols, such as Hypertext Transport Protocol Secure (HTTPS), Secure File Transfer Protocol (SFTP), Transmission Control Protocol (TCP) and others which are built-in throughout the internet. The goal of a secure transfer is to ensure that pertinent data relating to the security level, source, destination, and message contents are never revealed to unauthorized parties.

Transferring information between authorized parties sends encrypted bytes of data across networks from a source to a destination. However, during transfers its possible for encrypted data to pass through third-party routers with unknown security levels, allowing an attacker the opportunity to monitor messages. While the message data generally remains secure, the attacker could deduce information based on network traffic flow analysis. Common characteristics, such as source and/or destination IP addresses, can be evaluated by flow analysis, identifying patterns of possibly related data.

The exposure of origin, destination and other routing information is where the Cross Domain Problem (CDP) originates. There is a need to protect information, regardless of security level, during transmission across domains of unknown classifications. The solution to this problem is known as a Cross Domain Solution (CDS). Previous solutions have focused on protected networks that manage the risks associated within the transfer, which is only a partial solution. The goal of a CDS is to protect the authentication, confidentiality, and integrity of data across any intermediate party.

In 2018, Cody Tinker examined the feasibility of Homomorphic Encryption (HE) as a CDS [1, 2]. This work confirmed that the CDP can be addressed with the use of an HE scheme paired with a Lightweight Cipher (LWC). In his implementation, Yet Another Somewhat Homomorphic Encryption (YASHE) [3] was chosen as the HE scheme and SIMON [4] was chosen as the LWC. The results of Tinker's work demonstrated that it could be done; however, the overall results showed that the practicality of mainstream use was still not where it needed to be. Since this preliminary test, advancements have taken place in both fields, LWCs and HE schemes, opening up the possibility of a solution that yields better results in efficiency, complexity, and security. In addition to new LWC enhancements, there has also been work towards Hybrid Homomorphic Encryption (HHE) ciphers which are specifically designed to work with HE, ;ending themselves even further to such applications as this. By comparing current options and conducting further testing, a viable instance may be found.

1.2 This Work

The objective of this thesis was to research advancements in recent years to determine a cipher and HE scheme pair, what some are calling HHE, that will provide improved performance when applied as a CDS. The original focus was on evaluating

LWC finalists from the NIST Lightweight Cryptography Competition [5] along with recent enhancements to HE schemes. The LWCs, while possible candidates, contained more complex computations that would not lend themselves as nicely to the type of implementation this work set out to achieve. However, while researching the ciphers, HHE catered ciphers were found, such as Rasta [6] and its newer variants. After comparison of the variants and weighing their pros and cons, the choice was made to use Pasta [7] for the base preliminary encryption method.

When researching the progress made towards HE schemes in recent years, there were two routes to consider. The first regarding optimizations made to existing second generation schemes, Brakerski/Fan-Vercauteren (BFV) [8] and Brakerski-Gentry-Vaikuntanathan (BGV) [9], and the second, looking at the newer third generation schemes, Fast Fully Homomorphic Encryption over the Torus (TFHE) [10] and Fastest Homomorphic Encryption in the West (FHEW) [11]. The former was chosen primarily on the basis of being around for over 10 years, but also completing extensive testing in both performance and security. The third generation schemes, while promising, require additional testing and optimization to improve the trade off between performance and security. After delving into the existing schemes, the final choice for the HE portion was to use BFV.

With the selection of both parts, and familiarization of design and functionality, a prototype application was constructed to mimic the desired CDS behavior. Initial application logic for Pasta and BFV was pulled from [12] and [13] respectively. The timing of various segments of implementation were captured and recorded for comparison. Furthermore, the application was designed to test two renditions of the homomorphic decryption circuit, one using 3-round Pasta and one using 4-round. With preliminary verification of successful decryption, and to confirm the ability to compare specific values and perform trivial computations on the resulting homomorphically encrypted data, testing was performed for proof of possible future use cases.

Chapter 2

Background

2.1 Cross Domain Solution

The CDS is a controlled interface that provides the ability to access or transfer information between different security domains, whether that be done manually or automatically [14]. The messages transferred in these instances typically pertain to classified information that require security clearances. During transmission, not all networks used to move the data have the proper security levels. Therefore, networks should not be able to identify the classification level, or the path taken to arrive at the current router. The purpose of a CDS is to solve the CDP, depicted in Figure 2.1, to ensure that data reaches its intended target while maintaining the security of contents and involved parties.

Current solutions use protected platforms with specialized software applications which function as a guard between security domains [15] and focus on security policies and risk management. The information passed between these domains is subjugated to meet acceptance criteria prior to transmission. While this addresses part of the problem, it restricts transfers to networks with the proper security level.

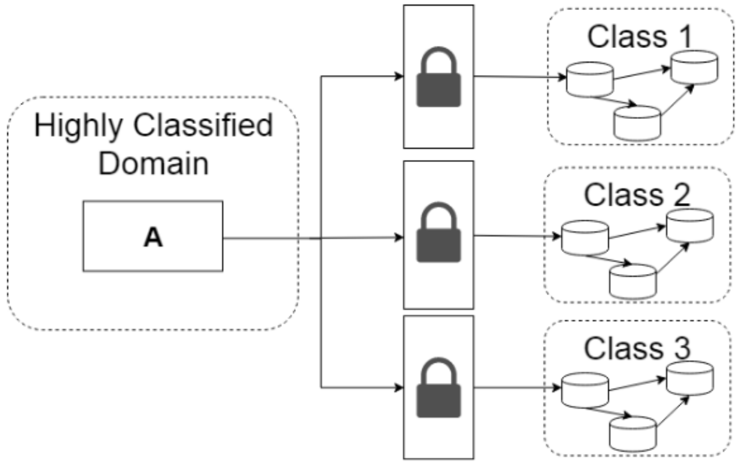

Figure 2.1: Cross Domain Problem

2.2 Ciphers

Ciphers, also known as encryption algorithms, are methods for encrypting and decrypting data. Encryption algorithms are computational procedures that makes the information unreadable to anyone besides the intended recipient [16]. As shown in Figure 2.2, encryption transforms the original message, the plaintext, into the ciphertext using the key. When decrypting data, the ciphertext reverts back to the original plaintext, using the same key in this instance. Not all ciphers execute the same algorithm, so there may be other variables necessary to complete the computation; however, the baseline described is constant among ciphers.

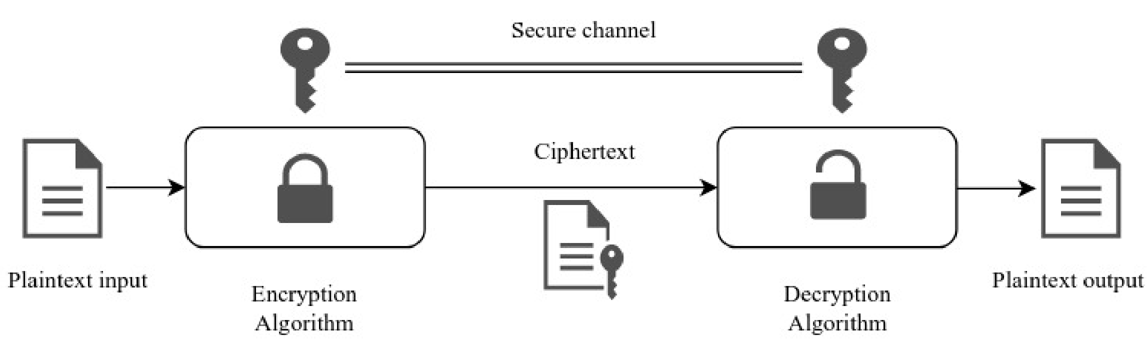

Figure 2.2: Basic Encryption and Decryption Flow

Ciphers use one of two types of keys, symmetric keys or asymmetric keys. A key is a string of bits created to scramble and unscramble data when using a cipher. Protocols such as Key Encryption Protocol (KEP) generate randomized keys of specified lengths from a set of all possible keys, known as a key space.

When generating keys, one must consider the strength and security a key will offer with the cipher of choice. The security strength of a cipher is dependent on how hard it is to break the code, determine the key used, mathematically [17]. Distributing generated keys to authorized parties can be a concern since secure sharing is essential yet sometimes challenging. A common approach to mitigate the issue is to use Encrypted Key Exchange (EKE) to share a key over an unsecured network without exposure.

2.2.1 Symmetric Key Ciphers

A symmetric key cipher [18], also known as a private key cipher, is used for both encryption and decryption. For example, Party A wishes to converse securely with Party B. Party A generates a key and shares it with Party B, using something like EKE. The key is then used to encrypt messages sent between key holders. When a message is received, the key is used to decrypt the encrypted data. By requiring the key for decryption, only those authorized can revert the received ciphertext back to the original plaintext. This method protects against attackers, who manage to obtain the ciphertext during transmission, from easily deciphering the message. There are two types of ciphers that utilize this symmetric structure, block ciphers and stream ciphers.

With the same key used for encryption and decryption, a new private key is necessary for every secure group conversation. Private key management is critical in order to prevent the shared key from being leaked, stolen, or used in other transactions.

2.2.1.1 Block Ciphers

Block ciphers are designed to accept a fixed input of size b bits producing a ciphertext of equal size, shown in Figure 2.3. Should the plaintext be larger than the fixed b bits, the message is broken down into smaller blocks, as the name implies.

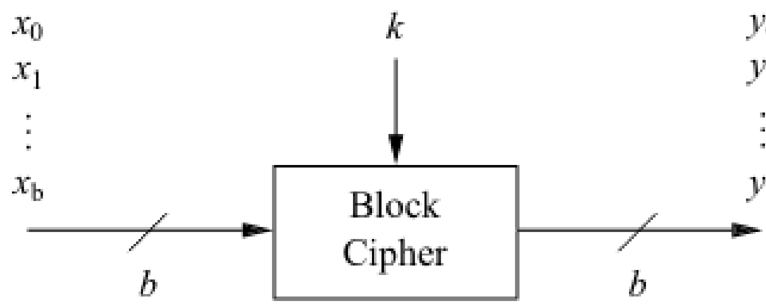

Figure 2.3: Block Cipher

Block ciphers typically operate with block sizes of 8-bytes or 16-bytes of plaintext. For larger sizes, different mode options are available: Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB), Counter (CTR) [18].

2.2.1.2 Stream Ciphers

Stream ciphers are designed to encrypt bits individually, depicted in Figure 2.4.

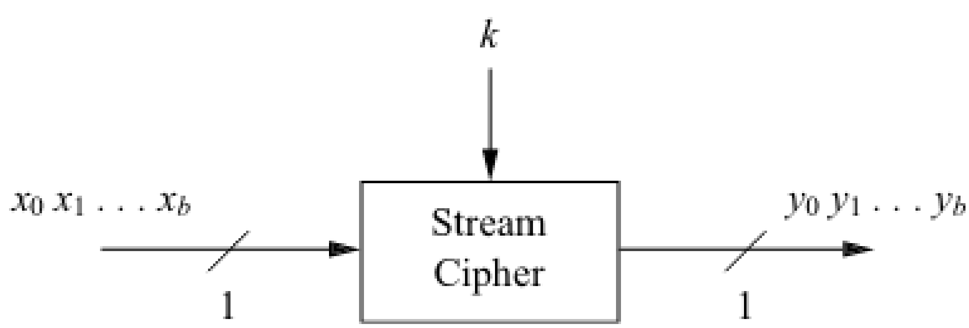

Figure 2.4: Stream Cipher

The key is transformed into a key stream where each bit is XORed with the incoming plaintext bit. There are two types of key streams, synchronous and asynchronous. A synchronous key stream uses only the key, while an asynchronous key stream uses both the key and the outputted ciphertext.

2.2.2 Asymmetric Key Ciphers

An asymmetric key cipher, also known as a public key cipher, is where authorized parties possess a private key, as in symmetric cryptography, and a public key [18]. For example, say Party A generates a pair of keys using a KEP; one public key shared with other authorized parties and one private key kept secret by Party A. These parties, say Party B and C, then encrypt messages to Party A using this public key; however, they cannot decrypt any messages encrypted with the public key. In order to decrypt the data, the private key is needed which is held solely by Party A.

An asymmetric key creates a secure, one-way communication method. In the case that both parties would like to send encrypted messages, Party B would need to generate their own key pair and share their public key with Party A. A public key can be shared with multiple authorized parties since decryption is dependent on the private key, held by the key generator. Asymmetric keys remove the need to generate a new key for every secure conversation.

2.2.3 Lightweight Ciphers

Ciphers are designed to run efficiently on desktop and server environments, restricting performance on devices where resources are limited. A lightweight cipher aims to provide solutions for these resource-constrained devices [19]. For hardware applications, attention to resource consumption is critical to minimize area used on device. For software applications, the number of registers along with the number of bytes of memory used must also be considered. LWC focus on these aspects, being conscious of the software and hardware footprints consumed.

In 2013, National Institute of Standards and Technology (NIST) started a study on lightweight cryptography to examine the current performance of NIST-approved cryptographic standards. In the process of the study, workshops were conducted to share standardization processes and collect public feedback. Five years later (2018), NIST invited candidates to compete in standardizing one or more lightweight ciphers following the criteria explained in [5]. Submissions were required to implement Authenticated Encryption with Associated Data (AEAD) along with the option of hashing. Fifty-seven candidates responded to the request, with 56 being accepted into Round 1 of the competition. The original 56 competitors were reduced during eliminations, sending 32 into Round 2. Finally, after another round of eliminations, 10 contenders were labeled as finalists in 2021. As of February 2023, ASCON [20] has been selected the winner for standardization.

2.2.3.1 Authenticated Encryption with Associated Data

Authenticated Encryption [21] is a method of enhancing encryption methods by incorporating authentication to prove its integrity. In addition to the base encryption, a Message Authentication Code (MAC) or tag is generated at encryption and decryption to verify that the data was not tampered with during transmission. The associated data portion of AEAD [22] is additional information that is incorporated with the original plaintext message. Associated data is often used as a header to expose non-confidential data in network packets to aid in routing. Figure 2.5 demonstrates the overall concept of AEAD; however, there are a few different ways that the MAC can be calculated.

There are different methods to how a MAC is generated. The first is Encrypt-then-MAC (EtM), where the plaintext is encrypted first then the resulting ciphertext is used to create the tag with a secondary key. The ciphertext and MAC are transmitted together so that on decryption, the MAC can be validated. Another type

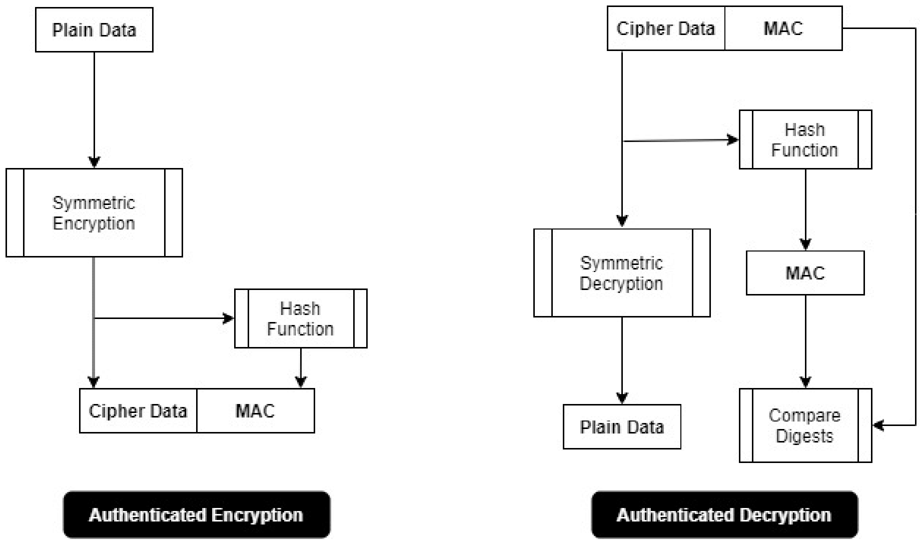

Figure 2.5: Authenticated Encryption with Associated Data

is Encryption-and-MAC (EaM), where the plaintext is used to generate the MAC and encrypted without the MAC. Both the ciphertext and the MAC use the same key. The last approach is MAC-then-Encryption (MtE), where the plaintext is used to generate the MAC first. The result is then appended onto the plaintext and encrypted with the same key a second time. In this instance, only the ciphertext is sent, given the MAC is baked into it.

2.3 Homomorphic Encryption

HE is an encryption method for the execution of operations on encrypted data, such as addition and multiplication [23]. Typically, in an effort to preserve the plaintext, data must be decrypted before normal computations can be executed, compromising user privacy. HE allows for computations to be done without changing the nature of the encrypted data [24] on any domain and without the need to decrypt the data. In other words, operations on ciphertexts will also execute similarly on the underlying plaintext, preserving the information.

Craig Gentry stated that given a ciphertext, anyone should be able to apply HE to output a ciphertext that encrypts the result for any desired function as long as it can be efficiently computed [25]. The final result should not reveal the given ciphertext, the function operated on the ciphertext, or any plaintext values. HE can be broken down into three types of schemes to address certain desires for speed and complexity: Partially Homomorphic Encryption (PHE), Somewhat Homomorphic Encryption (SWHE) and Fully Homomorphic Encryption (FHE).

2.3.1 Partially Homomorphic Encryption

One of the first types of HE schemes developed, PHE was designed to allow unlimited computations on encrypted data while constraining the computations to one operation, either addition or multiplication. Notable examples of PHE schemes include RSA in 1978 and El-Gamal in 1985 [23]. Due to this restraint, applications of PHE are limited. One instance where this type of scheme performs well is e-voting [26].

2.3.2 Somewhat Homomorphic Encryption

SWHE schemes are designed to allow for the execution of a set of operations, typically addition and multiplication, with a limited number of calls to each. When using SWHE, computations on ciphertext must be monitored in order to ensure that the operations execute successfully. When too many computations are executed, i.e. the multiplicative depth supported is exceeded, the noise growth can skew results from their expected output. This affects the use cases, requiring an understanding of the schemes application complexity and depth prior to using SWHE schemes. Instances where complexity and depth of computations are unknown limit the use of SWHE schemes. If a SWHE is used without that knowledge, resources can be depleted before the application completes its function.

2.3.3 Fully Homomorphic Encryption

FHE schemes are designed to combine the positive aspects of both PHE and SWHE, allowing for an unlimited number of operation calls using a set of operations. Due to the lack of limitations in execution and operations, FHE is the most flexible of the HE schemes. Creating a true FHE was difficult to construct, until 2009 when Craig Gentry had a breakthrough [25]. Using Gentry's FHE as a baseline, many others have published their own schemes, adding improvements over the years. Some examples include the leveled FHE BGV and BFV [21, 27, 28, 8]. The reason this solution is not currently predominant in mainstream encryption is due to underlying computational challenges within FHE schemes that have yet to be resolved.

2.4 Hybrid Homomorphic Encryption

While the coined term *Hybrid Homomprohic Encryption* (HHE) is still relatively new, introductions from Latuter et al. [29] defining the overall concept of HHE, has been around for a few years. HHE is the notion of encrypting the plaintext with a symmetric cipher first before sending it to the corresponding server. Once at the intermediate point, the data will use homomorphic operations to decrypt the data, converting the instance from the symmetric cipher into a homomorphic ciphertext, as seen in Figure 2.6. In the homomorphic state, remaining computations can be executed against the information. An example of this dual encryption state use case would be to compare a specific value for traffic redirection without exposing the plaintext data on the server.

Following Figure 2.6, the data (m) is encrypted with the selected symmetric cipher and the symmetric key (*sym_key*), generating *ct*. Moving into the HE space, the ciphertext *ct* is encrypted homomorphically to produce *hhe_ct*. The same procedure is taken to homomorphically encrypt the *sym_key*, generating *hhe_key*. A translation of

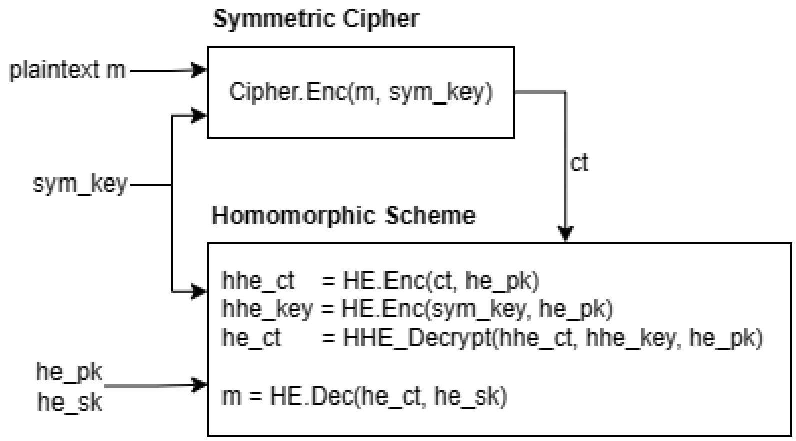

Figure 2.6: Hybrid Homomorphic Encryption Overview

the symmetric cipher's decryption algorithm that uses HE operations is developed to create the *HHE_Decrypt* function within the HE space. Using the function, along with *hhe_ct* and *hhe_key*, a ciphertext solely encrypted by the HE scheme is derived, *he_ct*. From here, operations could be executed on *he_ct*; however, should no computation be executed, if *he_ct* is decrypted, it should result in the original data *m*.

Many factors can influence the cipher selection though a key feature needed is finding a simplistic decryption circuit, with minimal operations, that does not sacrifice security. Reduced rounds and minimal complex computations result in an overall lower multiplicative depth. Multiplicative depth is the number of consecutive multiplications performed on a ciphertext within HE. Due to the sizing factor of multiplications in HE, higher depths are the main culprit behind slower, impractical application of HHE.

2.4.1 Hybrid Homomorphic Encryption Ciphers

HHE ciphers are specifically designed with compatibility to HE implementations in mind. This compatibility stems from the focus on maintaining a low multiplicative depth and minimizing computational complexity. Typically, One route to achieve this is to create a cipher with a reduced number of rounds. There are a few examples of existing HHE ciphers that have varying benefits for overall use or specific compatibility with an HE scheme. Examples include Rasta [6], Fasta [30], Pasta [7], and others.

Chapter 3

Lightweight Ciphers

3.1 NIST Lightweight Cryptography Finalists

The NIST Lightweight Cryptography Competition launched in August 2018 looking for LWC to be considered for lightweight cryptographic standards [5]. Competitors were allowed to submit for both AEAD and HASH functions as long as they adhered to the requirements.

Competitors were required to submit AEAD algorithms and up to a family of 10 algorithms with varying internal or external variables. The AEAD algorithms are required to support four inputs: variable-length plaintext, variable-length associated data, a fixed-length nonce of at least 96-bits, and a fixed-length key of at least 128-bits. The output was required to be a variable-length ciphertext. All algorithms submitted needed to provide a minimum of 128-bit security.

The competition began with 57 submissions, which turned into 56 candidates going into Round 1 after initial review. Evaluations were completed prior to selecting 32 candidates to continue into Round 2. In 2021, the 10 finalists were selected: ASCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, PHOTON-Beetle, Romulus, Sparkle, TinyJambu, and Xoodyak.

3.1.1 ASCON

ASCON, developed by Dobraunig et al. [20], was first introduced in the CAESAR competition from 2014 to 2019, becoming the primary choice for lightweight authenticated encryption. Their submission into NIST's competition included two AEAD algorithms, ASCON-128 (primary) and ASCON-128a, whose parameters are shown in Table 3.1 [31]. The rounds are broken into a triplet a, b, and c, where a is the number of rounds during initialization, b is the number of rounds during message processing, and c is the number of rounds for finalization.

Table 3.1: ASCON Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	# Rounds
ASCON-128	128	128	128	12,6,12
ASCON-128a	128	128	128	12,8,12
ASCON-80pq	160	128	128	12,6,12

The AEAD algorithm is based on duplex mode with an improved keyed initialization and finalization function using permutations. ASCON's permutations are broken down into three stages: addition of a round constant, substitution layer using a 5-bit S-box, and linear layer with 64-bit diffusion functions. Overall, the process is broken down into initialization, processing associated data, processing the plaintext, and finalization.

3.1.2 Elephant

Elephant, developed by Beyne et al.[32], submitted a family of three AEAD algorithms: Dumbo (primary), Jumbo, and Delirium. The parameters of each variant are given in Table 3.2 [31].

Elephant is a permutation-based AEAD that uses a nonce-based encrypt-then-MAC construction. The design utilizes a counter mode along with a varient of the Wegman-Carter-Shoup MAC function.

Table 3.2: Elephant Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	Permutation	# Rounds
Dumbo	128	96	64	160-bit Spongent	80
Jumbo	128	96	64	176-bit Spongent	90
Delirium	128	96	128	200-bit KECCEK	18

3.1.3 GIFT-COFB

GIFT-COFB, developed by Banik et al.[33], submitted only one algorithm to the competition. The parameters for GIFT-COFB are given in Table 3.3 [31].

Table 3.3: GIFT-COFB Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)
GIFT-COFB	128	128	128

The LWC submitted is comprised of two components, GIFT-128, a 128-bit Subsitution-Permutation Network (SPN) block cipher based on PRESENT, and COmbined Feed-Back (COFB), a block cipher based AEAD mode. GIFT-128 was designed for hardware optimization with 40-48 rounds. One round of GIFT-128 is comprised of three stages: SubCells, PermBits, and AddRoundKey.

3.1.4 Grain-128AEAD

Grain-128AEAD, developed by Hell et al.[34], submitted one algorithm to the competition. The parameters for Grain-128AEAD are given in Table 3.4 [31].

Table 3.4: Grain-128AEAD Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)
Grain-128AEAD	128	96	64

Grain-128AEAD is a bit-oriented feedback shift register that was optimized for hardware implementations. The first version, Grain v1, was selected as a finalist in the eSTREAM portfolio for hardware. For the NIST competition, version 2 was submitted which utilizes a refined, smaller version of Grain-128a. In addition to shrinking the size, version 2 added security against key reconstruction.

3.1.5 ISAP

ISAP, developed by Dobraunig et al.[35], submitted a family of four AEAD algorithms; ISAP-K-128a (primary), ISAP-A-128a, ISAP-K-128, and ISAP-A-128. At its core, ISAP uses one of two permutations, ASCON or KECCAK, as shown in Table 3.5 [31]. Rate is broken up into a tuple, where the first value represents the size of the rate for the nonce processing in re-keying and the second value is the size of the rate for all other phases. Rounds is also broken up into a 4-tuple, with s_H, s_B, s_E, and s_K. The values represent the rounds of permutation executed during authentication phase (s_H), nonce processing phase (s_B), encryption and decryption phases (s_E), and session key generation (s_K).

Table 3.5: ISAP Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	Permutation	Rate (bits)	# Rounds
ISAP-K-128a	128	128	128	400-bit KECCAK	144,1	16,1,8,8
ISAP-A-128a	128	128	128	320-bit ASCON	64,1	12,1,6,12
ISAP-K-128	128	128	128	400-bit KECCAK	144,1	20,12,12,12
ISAP-A-128	128	128	128	320-bit ASCON	64,1	12,12,12,12

ISAP is a permutation-based AEAD algorithm created with a wider range of security against implementation attacks. The mode implemented is a nonce-based encrypt-then-MAC construction, by XORing the message with the keystream, with authentication based on hash-then-MAC paradigm. The rounds are given in a set, with the rounds per phase: authentication phase, nonce processing phase, encryption and decryption phases, and re-keying function phase.

3.1.6 PHOTON-Beetle

PHOTON-Beetle, developed by Bao et al.[36], submitted a family of two AEAD algorithms; PHOTON-Beetle-AEAD[128] (primary) and PHOTON-Beetle-AEAD[32], provided in Table 3.6 [31]. The rate is provided in two parts, a/b, with a-bit absorbing rate and b-bit squeezing rate.

Table 3.6: PHOTON-Beetle Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	Rate
PB-AEAD[128]	128	128	128	128/128
PB-AEAD[32]	128	128	128	32/128

The rate specified in Table 3.6 provides the bit absorbing rate first followed by the bit squeezing rate. PHOTON-Beetle consists of two parts, a 256-bit, 12 round PHOTON permutation and the sponge-based mode Beetle. PHOTON_{256} is comprised of four layers, AddConstant, SubCells, ShiftRows, and MixColumns.

3.1.7 Romulus

Romulus, developed by Iwata et al.[37], submitted two separate families: nonce-based AEAD, Romulus-N (primary N1) and nonce misuse-resistant AEAD Romulus M. Both families consist of three variants, shown in Table 3.7 [31].

Table 3.7: Romulus Family Variants

AEAD Variants	Family	TBC	# Rounds	Key (bits)	Nonce (bits)	Tag (bits)
Romulus-N1		SKINNY-128-384	56	128	128	128
Romulus-N2	Romulus-N	SKINNY-128-384	56	128	96	128
Romulus-N3		SKINNY-128-256	48	128	96	128
Romulus-M1		SKINNY-128-384	56	128	128	128
Romulus-M2	Romulus-M	SKINNY-128-384	56	128	96	128
Romulus-M3		SKINNY-128-256	48	128	96	128

Romulus is based on the tweakable block cipher SKINNY. The SKINNY permuatation used consists of five layers for 40 rounds: SubCells, AddConstancts, AddROundTweaky, ShiftRows, and Mix Columns. Romulus-N implements a rate-1 TBC-based combined feedback mode while Romulus-M implements a MAC-then-encrypt mode.

3.1.8 SPARKLE

SPARKLE, developed by Beierle et al.[38], submitted a family of four AEAD algorithm variants: SCHWAEMM128-128, SCHWAEMM192-192, SCHWAEMM256-128 (primary), and SCHWAEMM256-256. All variant parameters are given in Table 3.8 [31]. The b-bit SPARKLE permutation is defined by the r-bit rate and c-bit capacity, where $b = r + c$. The number of step is also broken into a tuple x, y, where x is the steps in the SPARKLE permutation that process the associated data and message, and y is the steps used in initialization and finalization.

Table 3.8: SPARKLE Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	b,r,c	Steps
SCHWAEMM128-128	128	128	128	256,128,128	7,10
SCHWAEMM192-192	192	192	192	384,192,192	7,11
SCHWAEMM256-128	128	256	128	384,256,128	7,11
SCHWAEMM256-256	256	256	256	512,256,256	8,12

SPARKLE permutaions are comprised of two main components, an ARX-box Alzette which is a 64-bit block, 32-bit key cipher and a linear diffusion layer. Similar to a Subsitution-Permutation Network, SPARKLE implements a parallel application of Alzette with branch-dependent constants.

3.1.9 TinyJAMBU

TinyJAMBU, developed by Wu and Huang[39], submitted a family of three AEAD variants; TinyJAMBU-128 (primary), TinyJAMBU-192, TinyJAMBU-256. In Table 3.9 [31], all variants and their parameters are given.

Table 3.9: TinyJAMBU Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)	State Size (bits)
TinyJAMBU-128	128	96	64	128
TinyJAMBU-192	192	96	64	128
TinyJAMBU-256	256	96	64	128

TinyJAMBU, derived from JAMBU, focused on reducing the size of the original base cipher using keyed permutations. Encryption is broken down into four stages, initialization which handles key and nonce setup, the processing of associated data, the actual encryption, and the finalization of generating the authentication tag.

3.1.10 Xoodyak

Xoodyak, developed by Daemen et al. [40], submitted a single AEAD algorithm, Xoodyakv1. Table 3.10 [31] provides the parameters for each submission.

Table 3.10: Xoodyak Family Variants

AEAD Variants	Key (bits)	Nonce (bits)	Tag (bits)
Xoodyakv1	128	128	128

Xoodyak is based on a duplex construction that features a full-state variant when fed with a secret key. Xoodoo permutations are a 384-bit permutation, inspired by Keccak, that are sized with a focus on efficiency. The five-step process for encrypting with Xoodyak are Cyclist the key, Absorb the nonce, Absorb the associated data, encrypt the plaintext, and Squeeze the tag.

3.1.11 Performance of Competition Finalists

Software and hardware performance was analyzed for the finalists by NIST, as well as outside parties. Upon examination of the software performance tests conducted by NIST [41], Renner et al. [42], and Weatherly [43], along with the hardware performance tests conducted by the GMU CERG group [44], Aagaard and Zidaric [45], and

Khairallah et al. [46], two ciphers stood out as candidates: ASCON and TinyJAMBU. ASCON demonstrated consistent performance in both hardware with smaller resource footprint than others, and software with reduced code size and fast processing, that exceeded the others, including the baseline cipher, AES. TinyJAMBU performed the best overall in software, having a small code size with the fastest processing, and moderate performance in hardware, due to low resource consumption.

3.2 NIST Lightweight Cryptography Winner

In February 2023, NIST announced the winner of the competition. ASCON was selected for lightweight cryptography standardization. While the rationale behind the final decision is yet to be released, the overall performance of ASCON, whose algorithms are depicted in Figure 3.1, was the most consistent across hardware and software and outperformed most in both departments.

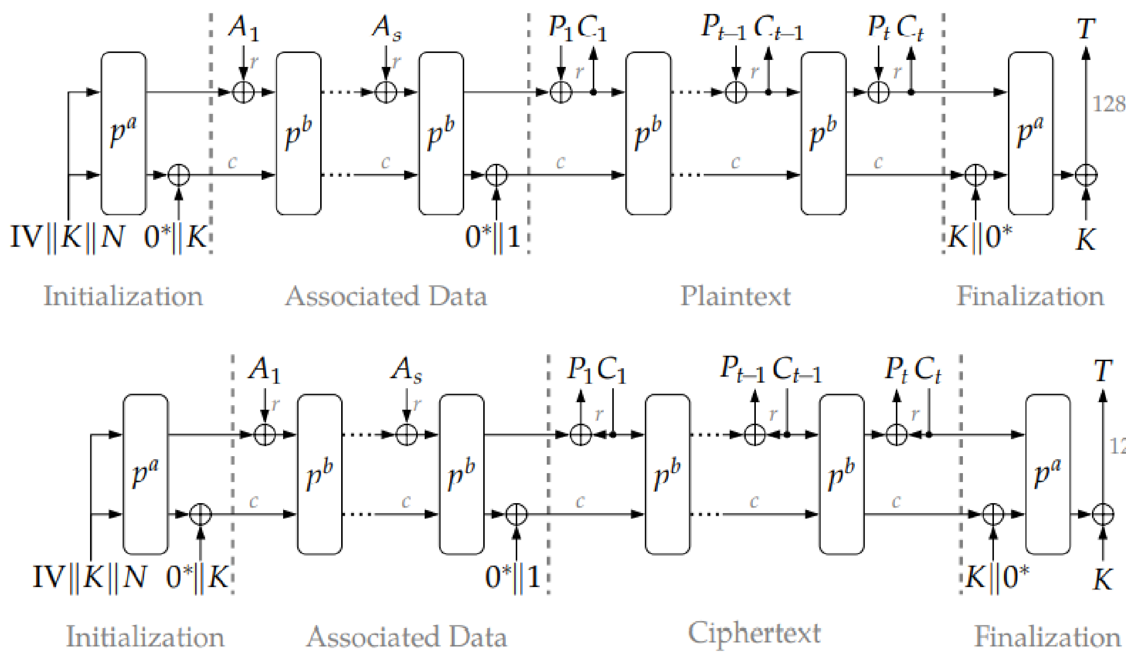

Figure 3.1: ASCON Encryption and Decryption

The features offered by ASCON, as well as its previous top performance in the CAESAR competition, made it one of the best candidates to win the competition. It accomplished lightweight and flexible hardware application, demonstrating throughput of 4.9-7.3 Gbps while using less then 10 kGE. This performance is theorized of being able to be increased further for even smaller applications or higher speeds. Given the top performance in hardware and software, it lends itself to cross-platform design scenarios where a back-end server would be needed.

Thorough testing, both from this and previous competitions, has concluded high cryptanalyic security with no indications of weaknesses. The improved initialization and finalization strengthen the already strong sponge-based design. The choice to use bit-sliced S-boxes prevents cache-timing attacks and the log algebraic degree of the S-box offers higher order protection using masking and sharing-based countermeasures.

Chapter 4

Hybrid Homomorphic Encryption Ciphers

In recent years, ciphers have been developed that specifically catered to Hybrid Homomorphic implementations. The base version, Rasta, was developed in 2018 by Dobraunig et al.[6]. Rasta is a symmetric cipher that has low AND depth, which lends itself to a lower overall multiplicative depthwithout sacrificing security. To accomplish this design, the Rasta cipher family applies permutations to the secret key, producing a keysteam that is used to obfuscate the plaintext.

Over the years, there have been variants on Rasta including Agrasta, Dasta, Fasta, Masta, Pasta and others. Many of the variants offer reduced rounds and/or improved performance. For this work, focus was directed towards Fasta and Pasta for being the newer of the variants with promising performance.

4.1 Fasta

Fasta was developed in 2021 by Cid et al.[30] as a variant of Rasta with the focus of creating a BGV-friendly linear layer. The original implementation of Fasta for use with BGV was catering to the application in the library HElib, which offers a levelled version of the HE scheme. At a high level, as depicted in Figure 4.1, the 329-bit secret key is copied into five different states of the same size, shifting four of the states by a value between 1 and 4 to the left. The result is a keystream comprised of 1645 bits. The size of the states is derived from searching for the value of m that coincides with

128-bits of security both in standalone and HHE application, along with a large, odd number of slots. The sweet spot determined for this implementation was a prime m equal to 30269, resulting in 329 slots.

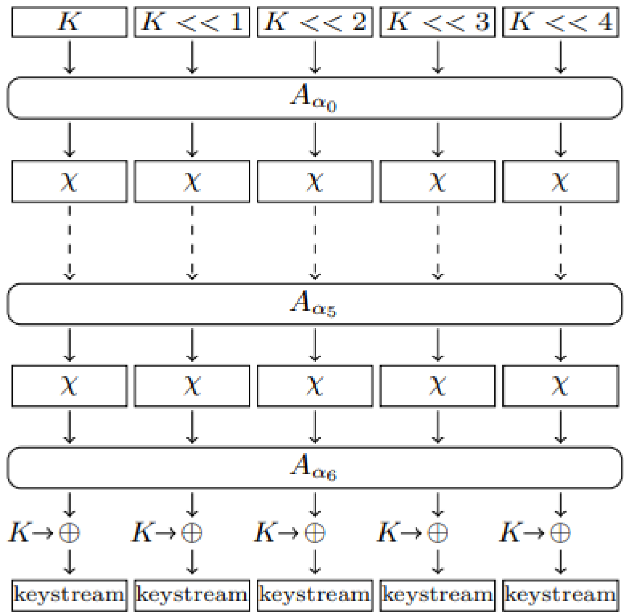

Figure 4.1: High-level design of Fasta

Keystream generation occurs over 6 rounds of an affine layer, A_{α_j}, and a non-linear transformation, χ, followed by a final affine layer and a feed-forward XOR of the states and the secret key. The affine layer is comprised of two parts, a rotation-based linear transformation and a round constant added to the state. The non-linear layer is composed of multiplication and addition of various indices within the state to define the new value at that index. These states produce a 1645 bit keystream that is added to the plaintext to produce the ciphertext and subtracted from the ciphertext to produce the plaintext.

4.2 Pasta

Pasta was developed in 2021 by Dobraunig et al.[7] as a variant of Rasta with the focus of optimization for integer HHE implementations. How this was accomplished was from the idea of converting Rasta to $\mathbb{F}_p^t$. Unlike many existing ciphers that operate over $\mathbb{Z}_2$ or $\mathbb{F}_2$, Pasta works in $\mathbb{F}_p$, where p is a large prime. The main caveat to the operating on plaintexts in $\mathbb{Z}_2$ is that the construction of binary circuits is required to handle HHE integer cases. The goal of Pasta was to efficiently and securely realize HHE over $\mathbb{F}_p$ while achieving same sized keystream and plaintext/ciphertext.

Pasta has the option of 3 rounds, Pasta-3, or 4 rounds, Pasta-4, with guaranteed 128-bit security. Table 4.1 depicts the main differences between the two variations of Pasta. Pasta defines its variables in words, where 1 word is equivalent to a 16-bit value. The two instances, 3-round and 4-round Pasta, are designed to provide at least 128 bits of security when used with prime fields $\mathbb{F}_p$ where $\log_2(p > 16)$ and the gcd(p - 1, 3) = 1. This feat is accomplished by using SHAKE128 [47].

Table 4.1: HHE Components with Pasta

# Rounds	Plaintext Size (words)	Key Size (words)	Ciphertext Size (words)
3	128	256	128
4	32	64	32

To operate in $\mathbb{F}_p^t$, a feed-forward operation replaced Rasta's truncation, preventing man-in-the-middle attacks more efficiently. This change led to the larger state sizes presented in Table 4.1. As noted in the table, the key size correlates to twice that of the plaintext and/or ciphertext. Pasta breaks its large key into two, equally sized state vectors in which computations are enacted against so that the final keystream is equal in size to the plaintext. The overall design of this keystream generation with the final application to either the plaintext or ciphertext is shown in Figure 4.2.

A single round of Pasta is comprised of a linear layer followed by the application of an S-Box. The linear layer executes a matrix multiplication and adding of a round constant to both states, then mixes the states together. To reduce the cost of the linear layer, Pasta went with 2t random elements that are used to construct two matrices. When combined, the matrices formed the single 2t x 2t matrix with a

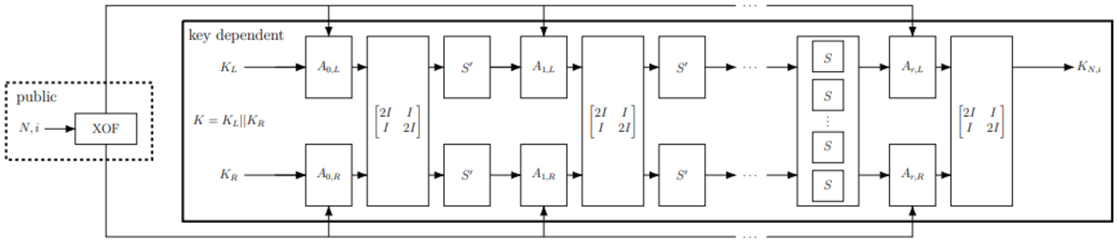

Figure 4.2: Design of Pasta

cheap mixing operation. The 2t random elements are where the two state arrays come into play in Pasta's algorithm.

Pasta has two different S-Box implementations. The primary choice is the Feistel S-Box that uses a quadratic function, rotations and masking in its execution. The corresponding equation given in Equation 4.1

$$S(\overrightarrow{x}) = \overrightarrow{x} + (\mathbf{rot}_{(-1)}(\overrightarrow{x}) \circ \overrightarrow{m})^2 \tag{4.1}$$

The second implementation is only used during the final round, the S-Box Cube. As the name implies, the state is cubed in this instance, as shown in Equation 4.2. While Feistel S-Box is the primary choice due to the minimal multiplicative depth, the S-Box Cube is added in to increase the ability to combat linearization attacks and reduce the state size.

$$S(\overrightarrow{x}) = (\overrightarrow{x})^3 \tag{4.2}$$

The rounds are executed then a final linear layer is applied at the end to generate the final keystream. This final linear layer is not followed by any additional S-box applications. Only the first of the two states is used as the final keystream. The value derived is then added to the plaintext to produce the ciphertext for encryption and subtracted from the ciphertext to produce the plaintext for decryption.

Chapter 5

Fully Homomorphic Encryption Schemes

5.1 Basic Preliminaries and Notation

The FHE schemes operate with a polynomial ring R, which is defined as $R = \mathbb{Z}[x]/(f(\text{x}))$. The function $f(\text{x})$ is a monic irreducible polynomial of degree d. The most common functions are either $\Phi_m(x)$, where the m is the m-th roots of unity for the minimal polynomial, or $x^d + 1$, where $d = 2^n$. The elements are often represented as a vector of the coefficients of the polynomial form.

When integer $q > 1$, a set of integers can be denoted by $(-q/2, q/2]$ in $\mathbb{Z}_q$. $\mathbb{Z}_q$ is only representing a set and not the same as the ring $\mathbb{Z}/\text{q}\mathbb{Z}$. This puts R_q as a set of polynomials in R with coefficients in $\mathbb{Z}_q$. The ciphertext elements are reduced by the integer q which acts as a modulo to place the value in the range $(-q/2, q/2]$. The plaintext modulus is defined as $t < q$, creating the message space R/tR. This means that R with coefficients modulo t. A radix-w system is used where w represent integers, $l_{w,q} = [\log_w(q)] + 1$.

The schemes describe two functions, the decomposition and power functions, denoted by $D_{w,q}$ and $P_{w,q}$. Word decomposition is derived as integer z in the interval $(-q/2, q/2]$, which can be denoted as $\sum_i^{l_{w,q}-1} z_i w^i$. The summation can be rewritten to accommodate ring element $x \in R$ when z_i is within $[0,\ w]$, creating $\sum_i^{l_{w,q}-1} x_i w^i$. The ring element is mapped to a vector of $l_{w,q}$ where the values are the decomposition's of the original values. The power function operates similarly on the same mapping except the ring element is scaled with the exponential of the radix integer. The decomposition function and power function are defined as

$$D_{w,q} : R \to R^{l_{w,q}}, x \to ([x_0]_w, [x_1]_w, ..., [x_{l_{w,q}-1}]_w)$$
$$P_{w,q} : R \to R^{l_{w,q}}, x \to ([x]_q, [xw]_q, ..., [xw^{l_{w,q}-1}]_q)$$

5.2 Brakerski-Gentry-Verauteren Scheme

Brakerski-Gentry-Verauteren (BGV) [9] expanded on the FHE scheme using weaker security assumptions in order to achieve better performance. The goal was to create an FHE scheme without needing to use bootstrapping, though a version with bootstrapping is a follow-on. Brakerski and Verauteren's original work on BV [48] was built upon to achieve this improved performance.

- KeyGen(d, q, χ): returning

$$\text{sk} = \text{SecretKeyGen(d, q, } \chi)$$
$$\text{pk} = \text{PublicKeyGen(d, q, } \chi\text{, sk)}$$
$$\text{sk'} = \text{sk} \otimes \text{sk} \in \text{R}_{qj}$$
$$\text{sk''} = \text{BitDecomp(sk', q}_i)$$
$$\tau = \text{SwitchKeyGen(sk'', sk}_{-1})$$

- Encrypt(pk, m): where m $\in$ R, set m = (m, 0) $\in \text{R}_2^2$, with r $\leftarrow \chi$, e $\leftarrow \chi^2$, returning

$$\text{ct} = \text{m} + 2 \cdot \text{e} + \text{a}_L^T \cdot \text{r} \in \text{R}_{qL}^2$$

- Decrypt(sk, ct): assuming the ciphertext ct is encrypted with s$_j$, returning

$$\text{m} = [[\langle \text{ct, s}_j \rangle]]_2$$

- Refresh(ct, $\tau_{sk''\rightarrow sk_{-1}}$, q_j, q_{j-1}): performs

$$\text{Expand: } \mathrm{ct}_1 = \mathrm{Powersof2}(\mathrm{ct}, \mathrm{q}_j)$$
$$\text{Switch Moduli: } \mathrm{ct}_2 = \mathrm{Scale}(\mathrm{ct}_1, \mathrm{q}_j, \mathrm{q}_{j-1}, 2)$$
$$\text{Switch Keys: } \mathrm{ct}_3 = \mathrm{SwitchKey}(\tau_{sk''\rightarrow sk_{-1}}, \mathrm{ct}_2. \ \mathrm{q}_{j-1})$$

- Add(pk, ct_1, ct_2): Two ciphertexts encrypted under s_j returning

$$\mathrm{ct}_3 = \mathrm{ct}_1 + \mathrm{ct}_2$$
$$\mathrm{ct}_4 = \mathrm{Refresh}(\mathrm{ct}_3, \tau_{sk''\rightarrow sk_{-1}}, \mathrm{q}_j, \mathrm{q}_{j-1})$$

- Mult(pk, ct_1, ct_2): Two ciphertexts encrypted under s_j returning

$$\mathrm{ct}_3 = \mathrm{L}^{long}_{ct_1,ct_2}(\mathrm{x}\otimes\mathrm{x})$$
$$\mathrm{ct}_4 = \mathrm{Refresh}(\mathrm{ct}_3, \tau_{sk''\rightarrow sk_{-1}}, \mathrm{q}_j, \mathrm{q}_{j-1})$$

5.2.1 Optimizations

Since original development in 2011, variants have emerged with improvements. The main addition to the design was developed by Gentry-Halevi-Smart (GHS) which offered scaled messages [49] that use Residue Number System (RNS). RNS operates on large integers by decomposing them into smaller numbers that fit into machine words of 64-bits. The main benefit of using a RNS variant is that the requirement of moduli $q_i = 1 \bmod t$ does not need to be met to perform modulus switching.

The library OpenFHE [13] offers four different versions of BGV with different RNS optimizations. The first is FIXEDMANUAL which uses a manual modulus switching implementation with the BGVrns variant. The second is FIXEDAUTO which still uses the BGVrns, but automatically implements modulus switching after the first multiplication. The third option, FLEXIBLEAUTO, uses theGHS with RNS with automatic modulus switching after the first multiplication.

The last option, and the default mode, is FLEXIBLEAUTOEXT, which uses the same design as FLEXIBLEAUTO with the addition of automatic modulus switching before the first homomorphic multiplication. While the other modes can run faster, the default offers the fastest speeds for smaller ring dimensions paired with a smaller ciphertext modulus while still satisfying the same level of security. In addition to the speed in certain cases, FLEXIBLEAUTOEXT also supports larger plaintext moduli.

5.3 Brakerski/Fan-Verauteren Scheme

Brakerski/Fan-Verauteren (BFV) [8], also referred to as FV, is considered a second generation FHE scheme developed in 2012. BFV is an extension of Brakerski's encryption scheme [50] where implementation of Learning with Errors (LWE) is converted to Ring Learning with Errors (RLWE). BFV can be used as a SWHE with the ability to relinearize ciphertexts or an FHE with the use of bootstrapping. The leveled FHE scheme offers noise that grows linearly with the depth of evaluation circuits.

- KeyGen(d, q, χ_{key}, χ_{err}, w): returning

$$\text{pk} = (\text{b,a}) = ([\text{-}(R_q\chi_{key}) + \chi_{err}],\ R_q)$$
$$\text{sk} = \chi_{key}$$
$$\text{evk} = \gamma = ([P_{w,q)}(\chi_{key})^2 \text{ - } (\chi_{err} + R_q\chi_{key})]_q,\ R_q)$$

- Encrypt(pk, m): where m $\in R_t$, p_0 = pk[0], p_1 = pk[1], u $\leftarrow R_2$, e_1, $e_2 \leftarrow \chi$, returning

$$\text{ct} = ([p_0 \cdot u + e_1 + \Delta * m]_q, [p_1 \cdot u + e_2]_q)$$

- Decrypt(sk, ct): where s = sk, $c_0 = ct[0]$, c_1 = ct[1], returning

$$\text{m} = ([\tfrac{t}{q} \cdot [c_0 + (c_1 \cdot s)]_q])_t \in R$$

- Add(ct_1, ct_2): where $\mathrm{ct}_1 = (c_{1,0}, c_{1,1})$ and $\mathrm{ct}_2 = (c_{2,0}, c_{2,1})$, returning

$$\mathrm{ct}_{add} = (c_{1,0} + c_{2,0}, c_{1,1} + c_{2,1})$$

- Mult(ct_1, ct_2, evk): returning ct_{mult}, which represents (c_0, c_1, c_2)

$$\mathrm{ct}_{mult} = (([\frac{t\cdot(ct_1[0]\cdot ct_2[0])}{q}])_q,\ ([\frac{t\cdot(ct_1[0]\cdot ct_2[1]+ct_1[1]\cdot ct_2[0])}{q}])_q,\ ([\frac{t\cdot(ct_1[1]\cdot ct_2[1])}{q}])_q)$$

- Relin(ct_{mult}, rlk): where ct_{mult} = [c_0, c_1, c_2] is a degree 2 ciphertext, returning ct' = [$\mathrm{c'}_0$, $\mathrm{c'}_1$] as a 1 degree ciphertext

$$\mathrm{c'}_0 = [\mathrm{c}_0 + \textstyle\sum_{i=0}^{l} \mathrm{rlk[i][0]} \cdot \mathrm{c}_2^{(i)}]_q$$
$$\mathrm{c'}_1 = [\mathrm{c}_1 + \textstyle\sum_{i=0}^{l} \mathrm{rlk[i][1]} \cdot \mathrm{c}_2^{(i)}]_q$$

5.3.1 Optimizations

Since original development in 2012, variants have emerged with improvements. Like BGV, BFV also makes use of RNS optimizations, specifically impacting the execution of multiplications. For BFV, there are two different RNS variations that have been created. The first was introduced by Bajard, Eynard, Hasan, and Zucca (BEHZ) [51], with the goal of eliminating the need for multi-precision arithmetic and suggesting techniques to use full RNS in BFV-like schemes. The second was proposed by Halevi, Polyakov, and Shoup (HPS) [52], which focuses on optimizing decryption and multiplication in the RNS by using Chinese Remainder Theorem (CRT) to manipulate large coefficients in the ciphertext polynomials. Making improvements from the BEHZ, HPS claims simpler, faster procedures with lower noise growth.

The library OpenFHE [13] offers four different versions of BFV. The first is HPS, which uses the RNS design proposed in [52], with procedures that use mix of integers and floating-point operations. The second is BEHZ which uses the RNS proposed by [51], with procedures based on integer arithmetic. The third option, HPSPOVERQ, uses HPS with static noise estimation to choose the size of RNS moduli. The last option, and the default in the library, is HPSPOVERQLEVELED. Expanding upon HPSPOVERQ, modulus switching is applied inside homomorphic encryption to reduce computational complexity. In addition to the modes, there is the option of STANDARD or EXTENDED when it comes to the modulus Q. STANDARD mode executes encryption with a fresh modulus and the EXTENDED uses a larger modulus by using auxiliary moduli for homomorpic multiplication followed by modulus switching. STANDARD is the default used.

5.4 Homomorphic Operation Examples

Homomorphic operations have expanded beyond the basic addition and multiplication, using these computations as building blocks to create more functionality. Within the homomorphic library OpenFHE [53, 13], the functions EvalAdd, EvalSub, EvalMult, EvalSum and EvalRotate are offered, along with others that lend themselves to different use cases. Below defines how these operations affect the ciphertext. All examples will uses the following values for two ciphertexts, given as their integer values, and the plaintext modulus:

$$\text{c1} = [1, 100, 1000, 5000]$$
$$\text{c2} = [2, 20, 200, 1000]$$
$$\text{ptm} = 65537$$

The plaintext modulus *ptm* is set tp 65537 in this example since that is the value recommended when working with integers.

5.4.1 Homomorphic Addition

The two ciphertexts, *c1* and *c2*, can be homomorphically added together by performing EvalAdd on each element:

$$c3 = \text{EvalAdd}(c1, c2) = [c1 + c2]$$

$$c3 = [[1, 100, 1000, 5000] + [2, 20, 200, 1000]]$$

$$c3 = [3, 120, 1200, 6000]$$

Since the results do not exceed the plaintext modulus, the values computed are the values stored.

5.4.2 Homomorphic Subtraction

The two ciphertexts, *c1* and *c2*, can be homomorphically subtracted together by performing EvalSub on each element:

$$c3 = \text{EvalSub}(c1, c2) = [c1 + (\text{-}c2)]$$

$$c3 = [[1, 100, 1000, 5000] + [-2, -20, -200, -1000]]$$

$$c3 = [-1, 80, 800, 4000]$$

Since the results do not exceed the plaintext modulus, the values computed are the values stored.

5.4.3 Homomorphic Multiplication

The two ciphertexts, *c1* and *c2*, can be homomorphically multiplied together by performing EvalMult on each element:

$$c3 = \text{EvalMult}(c1, c2) = [c1 \cdot c2]$$

$$c3 = [[1, 100, 1000, 5000] \cdot [2, 20, 200, 1000]]$$

$$c3 = [2, 2000, 3389, 19188]$$

Since the last two elements, when multiplied together, exceed the plaintext modulus, the result is mod by *ptm*. The logic behind the last elements is as follows:

$$c3[3] = 1{,}000 \cdot 200 = 200{,}000 \bmod 65{,}537 = 3389$$

$$c3[4] = 5{,}000 \cdot 1{,}000 = 5{,}000{,}000 \bmod 65{,}537 = 19{,}188$$

5.4.4 Homomorphic Sum

A new feature offered is calculating the sum of the elements contained within a given ciphertext using EvalSum. Using *c1*, each element is added with all subsequent values and stored in its current index.

$$\text{c3} = \text{EvalSum(c1, 4)} = [\ \textstyle\sum \text{c1}\ ]$$

$$\text{c3} = [(1 + 100 + 1000 + 5000), (100 + 1000 + 5000), (1000 + 5000), (5000)]$$

$$\text{c3} = [6101, 6100, 6000, 5000]$$

Since the results do not exceed the plaintext modulus, the values computed are the values stored.

5.4.5 Homomorphic Rotation

A ciphertext, *c1*, can be rotated to the left or to the right by x spaces using EvalRotate.

$$\text{c3} = \text{EvalRotate(c1, -2)} = \text{c1 ¿¿ 2}$$

$$\text{c3} = [1, 100, 1000, 5000] \text{ ¿¿ } 2 = [1000, 5000, 1, 100]$$

$$\text{c4} = \text{EvalRotate(c2, 3)} = \text{c2 ¡¡ 3}$$

$$\text{c4} = [2, 20, 200, 1000] \text{ ¡¡ } 3 = [1000, 2, 20, 200]$$

In order to differentiate between which direction the array is shifted, the index value is either set to a positive value for a left shift or a negative value for a right shift.

Chapter 6

Component Selection

6.1 Cipher Decision

The first choice required was selecting a cipher that would be paired with an HE scheme. The original choice and the choice implemented differ due to discoveries made while attempting to develop the application. About halfway through the design, focus shifted from using a lightweight cipher to using a hybrid homomorphic encryption cipher.

6.1.1 Pursuit of Lightweight Cipher

The original direction of this research was to investigate the latest improvements to lightweight cryptography. Given the taxing computations that can result from heavy, robust ciphers, lightweight options were thought to offer small, less complex decryption circuits that would translate better into hybrid homomorphic application. To find the best candidates, the NIST Lightweight Competition had a selection of well vetted contenders with 10 finalists on which to focus. After analyzing the hardware and software results [41, 42, 43, 44, 45, 46], it was evident that the top two contenders were ASCON [20] and TinyJAMBU [39]. All demonstrated strong performance in both hardware and software, with ASCON edging the others out in hardware while TinyJAMBU did the same in software.

The initial cipher choice was TinyJAMBU; however, ASCON ended up winning the competition. The smaller code size and overall more simplistic logic offered by TinyJAMBU seemed to be the better choice. Upon initial testing and implementation, it turned out that implementing TinyJAMBU in this scenario was more complex and challenging then originally theorized. The logic needed to translate the decryption circuit into a homomorphically compatible circuit would have resulted in poor performance overall. ASCON was reconsidered for a short period of time after TinyJAMBU proved more difficult. It had similar complexity issues that made it less compatible with homomorphic encryption where it stands now.

6.1.2 Pursuit of Hybrid Homomorphic Encryption Cipher

When the possibility of achieving a working product seemed slim, research led to the discovery of hybrid homomorphic encryption ciphers. Rasta, being one of the predominant choices, was outdated with many newer variants that provided improvements to performance. The latest two, Fasta and Pasta, appeared to have the most promising results. First looking into Fasta for advertised fast processing, it was noted that the design was catered specifically towards BGV. If that HE scheme had been selected, Fasta would have been the choice; however, the HE choice was to use BFV which required a different set of evaluations between the ciphers.

The final decision came down to the operating field along with what could add to the HE community. Pasta is designed to cater towards integers while Fasta works more with binary circuits. With the HE scheme choices broken down between BGV and BFV, it made more sense to work with Pasta since both share the use of integers more. Performing a deeper dive between the two ciphers also revealed that translating Pasta's decryption circuit to homomorphic decryption would be more straight forward given that it was not specifically designed to work with one instance. The flexibility to possibly work with various HE schemes in the future made Pasta the right choice.

6.2 Homomorphic Encryption Scheme Decision

6.2.1 Library

There are multiple libraries available with open-source HE schemes that offer a variety of functionality. Most have a limited number of supported schemes, reducing the flexibility of the user's choice. Long standing libraries such as HELib [54] and SEAL [55] cater to one, maybe two schemes. To avoid limiting the possibilities of the implementation, a library that had multiple options was searched for, leading to Palisade [56] which later became OpenFHE [13].

Developed by Polyakov, Rohloff, and Cousins, OpenFHE came about in 2022 as an extension of the original Palisade. Developers from various libraries came together in order to create a cohesive destination for post-quantum HE code. It is designed for usability, performance, modularity and cross-platform support. Of the existing libraries, OpenFHE has source code for all major FHE schemes, including BGV [9], BFV [8], Cheon, Kim, Kim and Song (CKKS) [57], FHEW [11], and TFHE [10]. While OpenFHE remains under continued development, the features available are sufficient for creating a prototype for this project. While this implementation only uses one of the FHE schemes, the parallelism between implementations opens up the possibility for future extensions that use other schemes.

6.2.2 Scheme

Looking at the new development in HE, there are now fourth generation schemes. The second generation schemes are BFV and BGV, the third generation schemes are FHEW and TFHE, and the fourth generation scheme is CKKS. CKKS makes use of machine learning which is more complex than this implementation required, so it was not considered. Now, the main research now was to look between second and third generation sets and decide upon which to perform a deeper dive.

Third generation schemes were developed in 2015 and 2018. When it comes to new technological enhancements, especially in the post-quantum field, that is still relatively new. While new does not equal bad, the testing and optimizations performed on the ciphers is not up to par with that of older schemes. FHEW and TFHE are both designed to evaluate arbitrary Boolean circuits with bootstrapping after each gate evaluation. While the schemes demonstrate improved speeds with stronger assumptions and optimized bootstrapping, there are reported disadvantages at this time in the relation between performance and security. For increased performance, security is sacrificed and vice versa. Due to this, focus was directed towards second generation FHE schemes.

Second generation schemes have been around for over 10 years with extensive testing and new optimizations added recently. With verified security and performance metrics, it seemed logical to look at BFV and BGV for current enhancements that could cater toward this implementation. Both schemes offered variants that make use of RNS to improve noise growth and computational complexity. Comparing the two in [27], it was determined that BFV offers more robust noise estimate inaccuracies while BGV requires precise noise estimates to avoid decryption failure. Furthermore, the leveled versions of BFV have one less bit per multiplication level, reducing noise further. For smaller plaintext moduli, BFV offers faster performance; whereas for larger plaintext moduli, BGV is the suggested choice. This comparison of performance and reduced noise growth supported the final decision of BFV.

Chapter 7

Implementation

The overall implementation of the project was to develop a working translation of the Pasta decryption circuit into a homomorphic compatible instance. Using the OpenFHE library's BFVrns, the design was implemented where BFV operations were used to replicate the behavior of Pasta to homomorphically decrypt. Once that was successfully designed and tested, follow-on test cases were created to show potential use.

Note that the way functions operate in BFV is that the operation is performed element by element, as described in Section 5.4. This means that the value at a[0] is added or multiplied by the element in b[0] and then stored in the result c[0]. This design, along with the inability to cleanly index the array while it is encrypted, adds a few extra steps to translate the decryption circuit originally given by Pasta [12].

7.1 Hybrid Homomorphic Encryption

Before decrypting the instance, the plaintext must first be encrypted. The overview of the encryption process is given in Fig. 7.1. Pasta's encryption function is used first, shown in Appendix A lines 100 to 105. This logic is taken directly from [12], where a plaintext, m, is encrypted using the Pasta's symmetric key, *k_pasta*.

The result, *ct_pasta* in this instance, is sent along with the symmetric key to be encrypted under HE. In addition to those values, the BFV public and private key pair

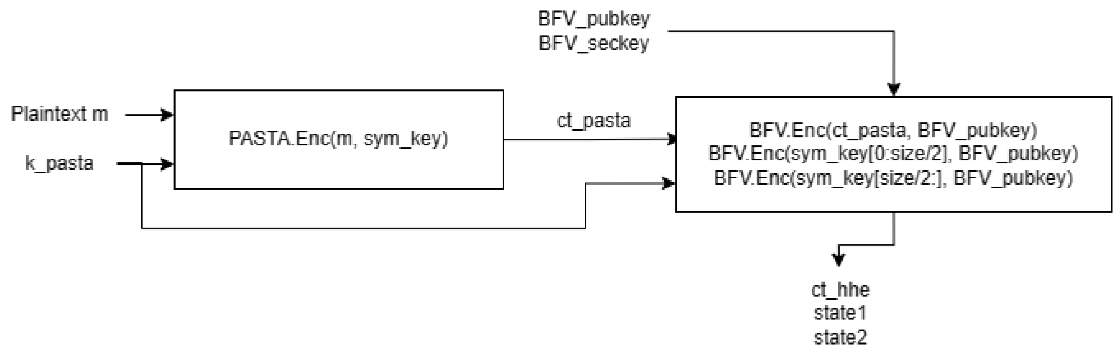

Figure 7.1: HHE Encryption

is passed as well. Given Pasta's symmetric key, k_{pasta}, which is double the length of the ciphertext, the value is split evenly into two state arrays, state1 and state2. The state arrays are comprised of the words stored in k_{pasta}, where each index houses one word. These values are encrypted with BFVrns and the homomorphic public key. Lastly, before switching from Pasta encrypted to BFV encrypted, the ciphertext, ct_{pasta}, must be encrypted a second time with BFVrns and the homomorphic public key. The values passed on to the decryption circuit are *state1*, *state2* and ct_{hhe}.

7.2 Hybrid Homomorphic Decryption Circuit

As explained in Section 4.2, the main logic of the decryption circuit follows the generation of the keystream that is subtracted from the ciphertext to obtain the original plaintext. In the main logic of the decryption circuit, Algorithm 1, a declaration of the number of rounds, R, is needed. Pasta offers the option of 3 rounds or 4 rounds with different constraints on key, plaintext, and ciphertext size as stated in Table 4.1. Reference to the function can also be found in Appendix A, called HHE_Decrypt. Three values are required to do these operations: the doubly encrypted ciphertext along with the two states, *state1* and *state2*

A key thing to note is that ***rc*** represents the round constant. This constant is a randomly generated vector of length equal to the state size. In order to generate

Algorithm 1 HHE Decrpyion Circuit

Require: $3 \leq R \geq 4$
Require: $state1, state2, ct_{hhe}$
  $r \leftarrow 0$
  while $r < R$ **do**
    $matmul(state1, rc)$
    $matmul(state2, rc)$
    $addRC(state1, rc)$
    $addRC(state2, rc)$
    $mix(state1, state2)$
    if $r < R - 1$ **then**
      $sboxFeistel(state1)$
      $sboxFeistel(state2)$
    else
      $sboxCube(state1)$
      $sboxCube(state2)$
  $matmul(state1, rc)$
  $matmul(state2, rc)$
  $addRC(state1, rc)$
  $addRC(state2, rc)$
  $mix(state1, state2)$
  $ct_{bfv} \leftarrow ct_{hhe} - state1$

the vector, which does not allow zero values, Keccak's HashSqueeze function is used, which is built off of SHAKE128 [47]. The round constant is assigned a new vector between each function call, so no $\boldsymbol{rc}$ in Algorithm 1 is the same value. The source code given in Appendix A shows the regeneration of the vector that was assigned as $\boldsymbol{rc}$. To reduce complexity, $\boldsymbol{rc}$ is passed in as a plaintext given the continual changes; however, encrypting it as a homomorphic ciphertext would be a trivial switch. The performance impact of such a change should also be minimal given that the multiplicative depth would not be impacted.

The majority of the functions referenced in Algorithm 1 are for generating the keystream, which upon completion, is the value stored in state1. Once the keystream is computed, state1 is subtracted from the ciphertext in BFV to convert the doubly encrypted value, ct_{hhe}, to solely BFV encrypted value, ct_{bfv}, shown in Appendix A line 479.

7.2.1 Linear Layer

The linear layer is comprised of three functions: matmul, addRC, and mix. Focusing first on matmul, Appendix A lines 231-273, the function executes matrix multiplication as summarized in Algorithm 2. Given the ciphertext size, X, a state, and the round constant, the state is multiplied by a set of rows calculated with the current row and the round constant. The key size is halved for these calculations due to the state being one half of the whole key.

Algorithm 2 Matrix Multiplication

Require: $state, rc$
  $X \leftarrow cipher\_size$
  $temp \leftarrow HE.Ciphertext(0)$ # Empty Ciphertext
  $row \leftarrow rc$
  while $x < X$ **do**
    $mult \leftarrow state \cdot row \leftarrow BFV.EvalMult(state, row)$
    $sum \leftarrow \sum mult[i] \leftarrow BFV.EvalSum(mult, X)$
    $masked \leftarrow sum \cdot 10..00 \leftarrow BFV.EvalMult(sum, 10..00)$
    $sum \leftarrow masked >> x \leftarrow BFV.EvalRotate(masked, 0 - x)$
    $temp \leftarrow temp + sum \leftarrow BFV.EvalAdd(temp, sum)$
    $row \leftarrow recalc\_row(row)$
  $state \leftarrow temp$

Since the state is encrypted under BFV, computations are executable against it. Following the design of Pasta's matmul function [12], the state is multiplied by the row, which is initially set as the round constant. The result, *mult*, is then summed together, where each element is added together with all trailing values. The array created from this has the sum at $sum[0]$ equal to $mult[0]$ plus all subsequent index values, the sum at $sum[1]$ equal to $mult[1]$ plus all subsequent index values, and so on. The only value of interest is stored at $sum[0]$ which is why the ciphertext is multiplied by a mask of 1 followed by 0's until equal the to length of X. Per the earlier statement of index based operations, the array must rotate the first element to the right x spaces in order to add the value to the correct location. Since BFV offers both left and right rotations, negative values are needed to indicate a right shift. Finally, the result of all computations are added into the temporary ciphertext and the row is recalculated. Once every index has been filled, the final temp array is stored back as the new state. Overall, the multiplicative depth of this function, per call, is 2.

The next two functions, addRC and mix, are pretty straight forward. For addRC, as shown in Algorithm 3 or lines 216-222 in Appendix A, the state and the generated round constant are added together and stored back in the state.

Algorithm 3 Add Round Constant

Require: $state, rc$

$state \leftarrow state + rc \leftarrow EvalAdd(state, rc)$

The mix state, Algorithm 4 or lines 280-285 in Appendix A, adds together state1 and state2 to produce a ciphertext containing the sum of both states. The resulting sum is then added to each of the states. Both of these functions have a multiplicative depth of zero given no multiplication operations needed.

Algorithm 4 Mix States

Require: $state1, state2$

$sum \leftarrow state1 + state2 \leftarrow EvalAdd(state1, state2)$
$state1 \leftarrow state1 + sum \leftarrow EvalAdd(state1, sum)$
$state2 \leftarrow state2 + sum \leftarrow EvalAdd(state2, sum)$

7.2.2 S-Box

As introduced in Section 4.2, Pasta uses two different S-Boxes in order to maintain low multiplicative depth while also combating linear attacks. Used for all rounds but the last, the Feistel S-Box, Algorithm 5 or lines 308-322 in Appendix A, has more complex logic while only having a multiplicative depth of 1.

The Feistel logic squares the given state. The result is then shifted right one to mimic the desired mask of a rotated 01..11 vector. In BFV, the shift leaves an extra value past the desired size since the size grows with multiplications. To mitigate the

Algorithm 5 SBox Feistel

Require: $state$
  $square \leftarrow state^2 \leftarrow EvalSquare(state)$
  $shifted \leftarrow square >> 1 \leftarrow EvalRotate(square, -1)$
  $shifted \leftarrow shifted \cdot 11..10 \leftarrow EvalMult(shifted, 11...110)$
  $state \leftarrow state + shifted \leftarrow EvalAdd(state, shifted)$

impact of the extra element, a mask is multiplied to maintain the array. The masked, shifted value is then added with the state and stored as the new state.

The final round in the decryption circuit uses the S-Box Cube, Algorithm 6 or lines 294-299 in Appendix A. As the name implied, the state is cubed. Since BFV does not yet offer a cube function, the state is squared first, then the product multiplied against the state again. Given that there are two subsequent multiplications, the multiplicative depth is 2.

Algorithm 6 SBox Cube

Require: $state$
  $square \leftarrow state^2 \leftarrow EvalSquare(state)$
  $state \leftarrow state \cdot square \leftarrow EvalMult(state, square)$

7.3 Cross Domain Solution Scenario

Before designing a test for a CDS, understanding the use case is necessary. As previously defined, a CDS is a form of controlled interface that provides manual or automatic access and transfer information between different security domains [14]. Since the goal is to avoid revealing route data, the scenario implemented will emulate that use case.

The idea is that there are multiple sources that are trying to send data to a specific endpoint, as depicted in Figure 7.2. That destination is defined by a correlating classification. The sources feed the information into an unknown domain, such as a broadcast network, that sends the data to gateways. These gateways have an unknown classification. All gateways are equally likely to be chosen, with no known classification; therefore, the data used to dictate the endpoint must be protected.

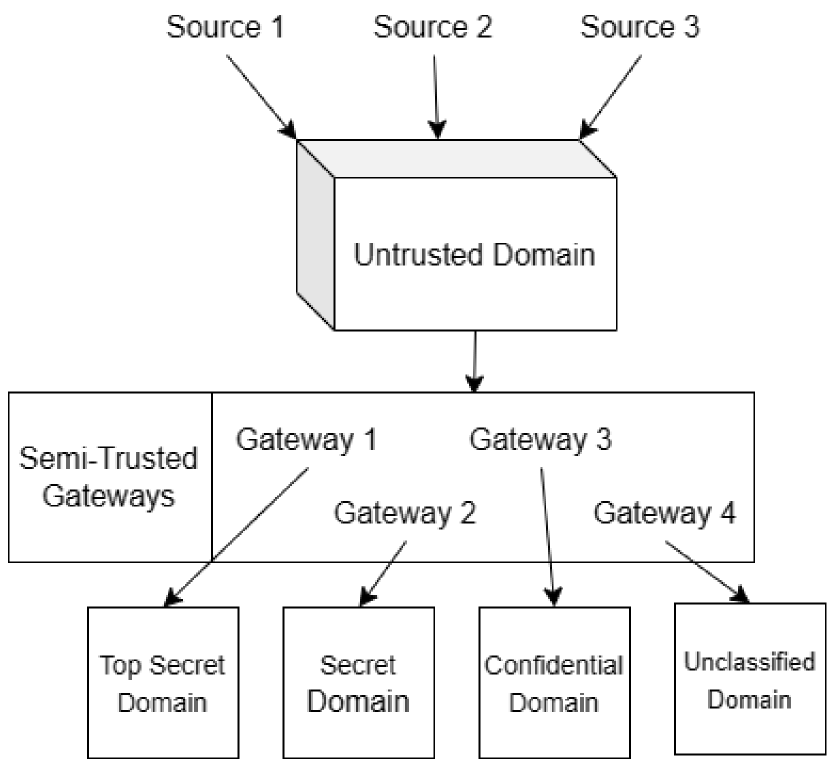

Figure 7.2: CDS Use Case

The endpoints shown in Figure 7.2 show a subset of the possible security domains. The common list used for classifications include: Top Secret, Secret, Confidential, Restricted, Official Use Only, and Unclassified. There are classifications beyond Top Secret; however, they are not the focus for this scenario. Classifications can be stacked, where multiple apply to a set of data depending on the use case, but are referenced by the highest clearance needed for that information. While multiple users could co-exist in a classification domain, they may not have the need-to-know to access the information intended for another user in the domain. This requires the need to head to a specific destination address because, while this case only shows one of each, there could be multiple domains of the same sensitivity.

To avoid revealing information to the unprotected domain and gateways but still reach the correct endpoint, identifying certain encrypted values is crucial. By comparing the actual and expected values, a single result can be checked where a zero means that the value matches. If the value does not match, a random value will be returned. Based on the result, the gateway can be told if the data should proceed to the classified domain.

7.3.1 Application to the Cross Domain Problem

The main test case for this implementation is the application to the CDP. The design mimics a CDS use case shown in Figure 7.3. A distribution center will stage most of the information, having the keys generated for both Pasta and BFV. The metadata for Pasta is encrypted, where the data houses the classification level along with any other desired information. The *m_gateway*, the classification comparison value, is encrypted under BFV. The security level of *m_gateway* could be Top Secret, Secret, Confidential, Restricted, Official, or Unclassified. In order for later comparison to be accurate, the value is placed in a plaintext array at the designated index, then encrypted. Lastly, in this stage, Pasta's symmetric key is encrypted under BFV as to not be revealed.

From there, BFV's public and private keys, along with the Pasta ciphertext and HE encrypted symmetric key, are sent to the gateway. At the gateway, the Pasta ciphertext can be doubly encrypted using BFV. Once encrypted, the hybrid homomorphic decryption circuit can be executed to create a ciphertext of the data that is solely encrypted under BFV. This hybrid homomorphic decryption is the same process as explained in the Section 7.2.

The comparison takes place next between the gateway classification value and the encrypted data, now referred to as BFV(*m_producer*). The comparison check, as describe in Algorithm 7, is a subtraction, which should result in a ciphertext of all zeros when the values match. OpenFHE has developed a subtraction method for

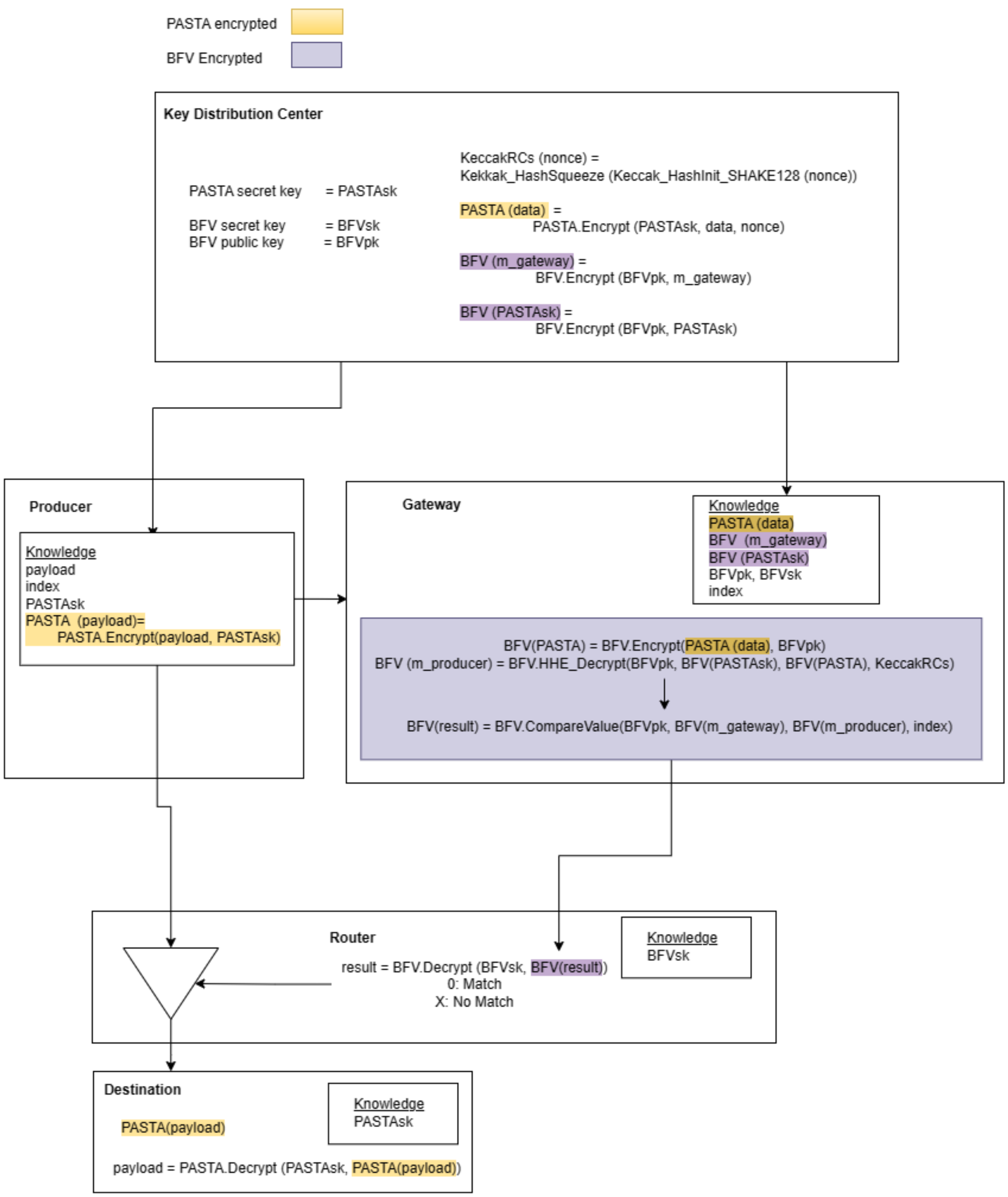

Figure 7.3: High Level Design Breakdown of a CDS

BFV that uses addition as the base in the background. This allows for a simplistic way of checking if the values are the same, subtracting the actual by the desired. If the result is zero, then its a match. With the computations executed by index, it is possible to compare multiple values at once, as depicted in Appendix A lines 591-645.

Algorithm 7 Application of a CDS

Require: $ct_{bfv}, ct\_size, indexes, values$
  $mask \leftarrow vector(0, ct\_size)$
  for $index \in indexes$ **do**
    $mask[index] \leftarrow 1$
  $masked \leftarrow ct_{bfv} \cdot mask \leftarrow EvalMult(ct_{bfv}, mask)$
  $comp \leftarrow vector(0, ct\_size)$
  for $index \in indexes$ **do**
    $comp[index] \leftarrow value$
  $diff \leftarrow masked - comp \leftarrow EvalSub(masked, comp)$

The resulting ciphertext from the comparison is passed to the router where it will be decrypted using BFV. The result will be an array of all zeros should the value match. If any value is not zero, then it is not a match. Based on the results, the router can allow or reject the payload message from the producer to continue to the destination. If approved for transit, at the destination, the payload can be decrypted as usual using Pasta's decryption circuit and Pasta's secret key.

For this implementation, an arbitrary set of 16-bit values were selected to act as classification indicators that could be incorporated into the payload. Table 7.1 shows the random values generated. Should the user desire a longer classification value to increase variation, it is possible that two indexes could be used in tandem to have a 32-bit value.

Table 7.1: Selected Classification Values

Classification	Hex Value
Top Secret	0x0CFE2
Secret	0x0A645
Confidential	0x028A9
Restricted	0x06D43
Official	0x0C489
Unclassified	0x057C2

Another instance that could be used in the metadata incorporated into the payload is the destination address. While the values are only 16-bits the use of two or more indexes could increase the amount of bits that could be used in comparison. For example, using two indexes an IPv4 address can be derived from 32-bits. The one caveat to the split indexed value is that the user may need to perform computations outside of the homomorphic implementation to know what value would allow those comparisons.

7.4 Additional Test Cases

Two separate test cases were derived for this project to observe usability along with the impact of increased multiplicative depth beyond the application of the case study. Each offers a different use case, from straight forward decryption circuit to arbitrary computations that demonstrate future capabilities. Both test cases were designed to work with 3-round and 4-round Pasta decryption circuits, in turn, testing large plaintext/key pairs and smaller plaintext/key pairs respectively. Over the course of the tests the multiplicative depth varies, thus impacting the results in each case.

7.4.1 Case 1

Test case 1 operates on the bare minimum. The purpose of this test is to give a baseline comparison of the decryption circuit with other tests and verify the accuracy of said circuit. When executing in a 3-round Pasta instance, the multiplicative depth is 12. For a 4-round Pasta instance, the multiplicative depth is 15. The run difference stems solely from the extra round. Appendix A lines 498-538 shows the test which decrypts the final ciphertext then compares the resulting plaintext with the original used. If the values matched, then it was successful.

7.4.2 Case 2

Test case 2 is designed to show that multiple computations could be executed following the completion of the decryption circuit. This proof of concept introduces the possibility of operations done on the ciphertext after the homomorphic decryption circuit is applied. The original idea was to demonstrate the ability to gather the sum of all elements within the ciphertext array. The sum would allow the user to calculate the average of the ciphertext, modulo the plaintext modulus. The user could then take the resulting decrypted value and divide by the size of the ciphertext. One day, possibly, there will be a division option created within HE that would allow for this to be done so decryption before final calculations is not necessary.

The instance developed for this case was slightly more complex, adding an additional multiplication into the mix. The purpose of the new design, in Algorithm 8, Appendix A lines 547-580, was to increase the multiplicative depth to emulate how additional computations would impact timing. The logic to execute this case added 2 multiplicative depth in the 3-round Pasta instance, and 3 in the 4-round implementation.

Algorithm 8 Test Case 2

Require: $ct_{bfv}, ct - size$
$mult \leftarrow ct_{bfv}^2 \cdot mask \leftarrow EvalSquare(ct_{bfv})$
$sum \leftarrow \sum mult \leftarrow EvalSum(mult, ct - size)$
$sum \leftarrow sum \cdot 10..00 \leftarrow EvalMult(sum, 10..00)$

The test case is pretty straight forward. The ct_{bfv} created from the decryption circuit is squared. In theory, the result could be multiplied by any ciphertext; however, for simplicity ct_{bfv} was reused. The sum of all elements in the product array are then summed homomorphically, with the overall sum stored in $sum[0]$. In order to obtain just that first value when decrypted, the sum is multiplied by a mask that will zero out all other array indexes, as shown in Figure 7.4.

The output of the function, *he_pt*, is the decrypted result of the computations, with a value stored in the first element of the resulting array. That value can be used to get the average of the data set by dividing by the number of elements in the ciphertext. While this cannot be homomorphically done as previously mentioned, it

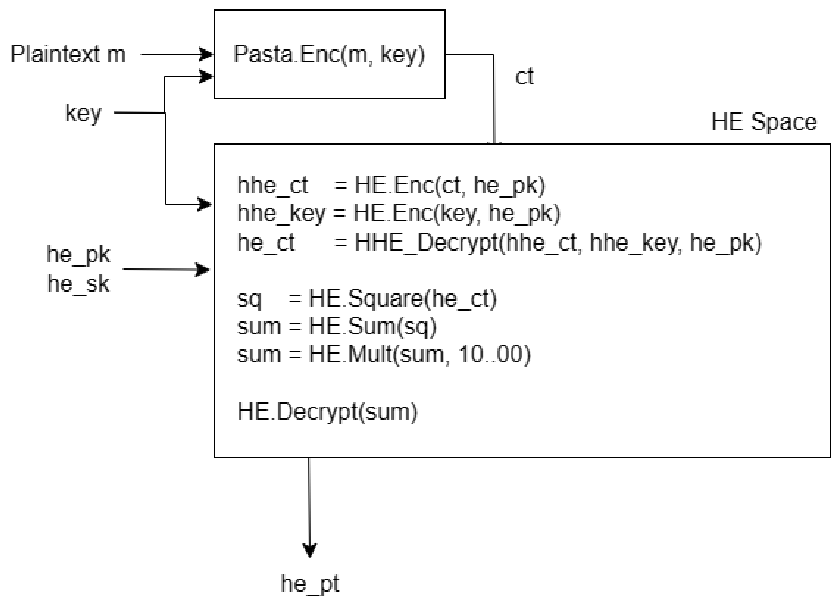

Figure 7.4: Test Case 2 Design

can be done post process without exposed additional data.

Chapter 8

Results

All test instances were executed on an HP Envy 16G laptop with an Intel i7 processor at 2.90GHz. For best results, a Linux distribution was set up using a Windows Subsystem for Linux (WSL) that had this project's implementation code along the OpenFHE library installed. Initial test runs resulted in core crashes due to memory consumption. To mitigate this issue, the allocated flash memory to the virtual environment was increased to 128G of the available 512G SSD. Another constraint added to execution was that the only application running was this implementation. If the processing power was split between applications, timing would be thrown off drastically. At runtime now, execution could complete regardless of the multiplicative depth needed (within reason).

In order to discover the proper depth to use for all stages, an approximation was chosen based on the algorithms defined in Section 7.2 along with the number of rounds needed in each instance. An initial test value was selected then executed. By viewing the outputs at each function's completion, it was possible to track when relinearization failed to maintain proper sizing. When the multiplicative depth is not high enough, the result returns a ciphertext array that contains elements beyond the original size, as well as incorrect values than what would normally result from the multiplication of the values. Through trial and error, the minimal depth for the decryption circuit was found: 2 per linear layer, 1 per Feistel S-Box, and 2 per Cube

S-Box. The increase in depth for the subsequent tests was derived in the same fashion.

For the main scenario and the test instances, the CryptoContext parameters were kept the same except for the multiplicative depth. The plaintext modulus selected was 65537, which was the recommended modulo for dealing with integers. The security parameter in BFV was set to HEStd_128_classic, which is OpenFHE's 128-bit security guarantee for BFV. This standard declaration set all other necessary parameters to ensure the application adhered to the desired security. Lastly, the maximum relinearization degree was set to 3. The generation of the CryptoContext can be seen in Appendix]A, lines 150-207.

8.1 3-Round Pasta Results

3-Round Pasta was the first to be tested. Table 4.1 shows sizes for key components utilized for the implementation. A plaintext array of 128 words, each work 16-bits, was selected and paired with a key array of 256 words, also 16-bits each. With the key split into the two arrays and used to generate the keysteam, the final ciphertext shared the format of the plaintext with 128 words in an array format.

All three test cases were run with the the same parameters, plaintext, and key, with the only variation being the multiplicative depth. Test 1 had a depth of 12, test 2 was 14, and the CDS application was 13. Timing for each test, Table 8.1, is averaged and recorded for the various stages. The post-decryption computations for test case 2 and the CDS application took minimal time, varying from milliseconds and max a second.

Table 8.1: HHE Decryption Circuit with 3-Round PASTA Performance

Depth	Crypto Context (min)	R1 (min)	R2 (min)	R3 (min)	Final LL (min)	Total (min)
12	1.77	2.81	2.97	3.01	2.97	13.54
13	2.10	4.33	4.25	4.25	3.99	18.92
14	2.17	4.32	4.25	4.18	3.95	18.88

Measurements were broken down into segments in order to observe variations at a deeper level. The "Crypto Context" is the time it takes for BFV to configure itself with the correct parameters. During this time, rotational keys are generated for all necessary left or right shifts. All columns labeled "R#" correspond with the time to complete that number round. "Final LL" monitors the time it takes to execute the final linear layer and "Total" is the total time to complete the decryption circuit at that multiplicative depth.

Seen in Table 8.1, as the multiplicative depth increases, so does the overall time it takes to run the decryption circuit. Intuitively, the smaller the required depth, the faster the time. This is due to fewer times the relinearization needs to be done to maintain the proper ciphertext size. Each round per row relatively takes the same amount of time. A key occurrence to note is that the final round tends to be the fastest despite having one more multiplicative depth used due to the reduced logic in Cube S-Box. The final linear layer was always the fastest.

Test case 1 applied the simple decryption circuit with no additional operations once ct_{bfv} was obtained, taking roughly 13.54 minutes to execute. The ciphertext was decrypted and each element was compared to the original plaintext to confirm accuracy. In the sense of cryptography where speed is critical, 13 minutes seems long, but it is much better than implementations in previous years. Looking at the results from Tinker's research [2], while there are multiple configurations, the configuration with 128-bit security took roughly 17 hours. With previous records of hours to homomorphically decrypt a ciphertext, getting the instance down to minutes is a significant improvement. Enhancements to speed could be improved before practical use; one method would be to execute the procedure on a more powerful processor.

Test case 2 took longer than the base case, as expected, given the increased multiplicative depth. One instance was processed for case 2, following Algorithm 8 with a pre-calculated value on hand to verify the results. Further testing with this case was not conducted because it was a verification test as opposed to the main focal implementation.

The CDS application had a few different instances that checked different index values to confirm accuracy of the checks. The average of the timing results was used to obtain the additional time the multiplicative depth contributed. The timing for only one check was recorded, but if multiple checks were done, the depth would increase further and result in longer times as well. However, if multiple indexes needed to be checked in one ciphertext, it would be possible to do so with the same multiplicative depth as a single index since operations are performed on a particular index.

8.2 4-Round Pasta Results

4-Round Pasta was the first to be tested. Table 4.1 gives the sizes for key components that were utilized for the implementation. A plaintext array of 32 words, each 16-bits, was selected and paired with a key array of 64 words, also each 16-bits. With the key split into the two arrays and used to generate the keysteam, the final ciphertext shared the format of the plaintext with 32 words in an array format.

All three test cases were run with the the same parameters, plaintext, and key, with only variation being the multiplicative depth. Test 1 had a depth of 15, test 2 was 18, and the application of the CDS needing 17. Timing for each test, Table 8.2, was averaged and recorded for the various stages. Similar to the 3-round instance, test case 2 and the CDS application timing results from additional computations post-decryption were minimal.

Table 8.2: HHE Decryption Circuit with 4-Round PASTA Performance

Depth	Crypto Context (min)	R1 (min)	R2 (min)	R3 (min)	R4 (min)	Final LL (min)	Total (min)
15	0.93	1.01	0.98	1.08	1.00	0.98	5.98
17	1.22	1.26	1.17	1.21	1.21	1.22	7.30
18	1.35	1.38	1.29	1.75	2.04	1.63	9.45

The base case with a multiplicative depth of 15 had great results, taking just under 6 minutes to execute. Compared to 17 hours [2], this is a drastic improvement. There is still concern that the time may be longer than that which would be deemed feasible in mainstream use, but it is a step in the right direction. Similar to the statements from 3-round Pasta, using a more powerful processor could significantly reduce the execution time as wall.

Test case 2 took longer than the base, as expected. The time increasing by 2 minutes as the multiplicative depth grew from 17 to 18. Despite increasing the base depth by three for test 2, the resulting time was still under 10 minutes.

The CDS application performed even better then test case 2 since the multiplicative depth was smaller. With a minute and a half difference from the base implementation to comparing values, the time is favorable. Again, while 7.30 minutes is fast compared to other HE instances, it could possibly be faster with parameter optimizations and processor selection.

8.3 Instance Comparison

Comparing the results in Table 8.1 and 8.2, note that while 3-round Pasta has smaller multiplicative depths, the execution time is much greater to its 4-round counterpart. The reason for this comes from the size of the ciphertext. Performing computations on a ciphertext of 128 words verses 32 words will impact overall performance. When relinearization takes place, it executes over every element in the encrypted array, increasing time for longer instances.

While the plaintext size for both instances could be up-scaled to create multiple block instances, scaling down is a difficult feat. Pasta designed both instances to default to a specific key size that will generate a keysteam of half that size. Initial testing to reduce the plaintext size for 3-round Pasta proved to increase the multiplicative depth in order to obtain the correct values. Since the trailing end of the plaintext would be zeros, the resulting addition or subtraction of the keystream would skew those values. A mitigation to this would be to incorporate the multiplication of a mask at the end of homomorphic decryption circuit to ensure correct sizing, but that increases the multiplicative depth and overall timing.

Based on the base runs and the subsequent tests, the best option for this instance is the 4-round Pasta. Typically, more rounds of the same type of logic would increase lead time due to the extra computations. The difference in time and depth demonstrates the additional impact that the size of the key and plaintext play on the results. In this instance, a smaller ciphertext and key size lend itself to better overall performance. Despite test cases 2 and 3 for 4-round also requiring an extra multiplicative depth than that of 3-round test cases, performance still exceeds that of 3-round's base case.

8.4 Comparison to Previous Work

In order to analyze how the implementation created compared to previous work, the results fared against those created by Cody Tinker in [2]. The examples pulled from Tinker's work were implemented with SIMON [4] and YASHE [3] with varying degrees of security. For the most accurate evaluation, test case δ was chosen for also having 128-bit security claims. In addition to this, test case α was used as the fastest implementation accomplished with the pair, though only offering 64-bit security. To obtain an idea of how the parameters compared, Table 8.3 was compiled with the instances' security, number of rounds, the polynomial ring degree (N), and the coefficient size ($\log_2(q)$).

The largest difference overall between the parameter configurations is the amount of round needed. The implementations accomplished in this project required only 3 or 4 rounds compared to the SIMON/YASHE taking either 32 or 44 rounds. The polynomial ring degree also differed with δ at the highest with 65536. Both of the

Table 8.3: Parameter Comparison

Instance	Security Bits	# Rounds	N	$\log_2(q)$
α SIMON/YASHE	64	32	16384	885
δ SIMON/YASHE	128	44	65536	1760
3-Round PASTA/BFV	128	3	32768	540
4-Round PASTA/BFV	128	4	32768	660

PASTA/BFV instances had the same ring degree of 32768, while α had the smallest at 16384. Lastly, the coefficient sizes varied with both SIMON/YASHE implementations having larger sizes than PASTA/BFV.

With the 4 instances decided upon, timing was compiled for all cases and given in Table 8.4. Here, three stages of the process were timed. The first is the Encrypt Key, the time in which it took to encrypt the symmetric cipher's key with the HE scheme of choice. The second is Decryption, the time in which it took for the decryption of the doubly encrypted ciphertext. The third is Evaluate Metadata, the time in which it took to complete the comparison of the classification value within the HE ciphertext.

Table 8.4: Timing Comparison

Instance	Encrypt Key (s)	Decryption (s)	Evaluate Metadata (s)
α SIMON/YASHE	5.4	2433.0	612.0
δ SIMON/YASHE	78.6	64367.6	8079.1
3-Round PASTA/BFV	0.321	1135.2	0.156
4-Round PASTA/BFV	0.382	438.0	0.256

Across all stages, the PASTA/BFV instances outperformed the previous implemented pair. Starting off with encrypting the symmetric key, both 3- and 4-Round PASTA/BFV tests completed in under a second, with α coming in at 5.4 seconds and δ taking 78.6 seconds. For this projects implementations, the time was taken to encrypt both halves of the key. The decryption times are even better, with 7.3 minutes for the 4-Round instance and just under 19 minutes for the 3-Round instance. Tinker's instances, when converted to more readable times, come out to 40 minutes for α and 17 hours for δ. That the fastest of Tinker's implementations still took twice as long as the slower of the two round tests shows the progress made in both fields, hybrid homomorphic ciphers and HE. Lastly, the evaluation of the metadata. Again, 3- and 4-Round instance completed in under a second while their counterparts took approximately 10 minutes or 2 hours.

Chapter 9

Conclusion

The progress made since Tinker tested a SIMON and YASHE pairing [1] in 2018 clearly shows how much the post-quantum field of research has grown. From initial pairings of everyday ciphers, to now having HHE ciphers specifically catered to these types of implementations, a major step forward in the post-quantum field.

The 4-round Pasta homomorphic decryption circuit created in BFV demonstrates the progression toward a feasible solution for many post-quantum applications, such as for the CDP. A homomorphic decryption taking 7.3 minutes compared to the previous 17 hours is a dramatic decrease in time. This 7.3 minute mark could still be improved upon as practical use desires, or even demands, faster speeds for cross network traffic. Still, it is nearly to the point that mainstream use may be feasible in a year or so. While 4-round Pasta performed the best, 3-round Pasta's decryption circuit still offered substantial results that prove practicality is coming soon. A time of just under 19 minutes for large message types in a CDS scenario is not an unreasonable starting point for improvement.

Subsequent testing proved application to different instances feasible that could lead to additional use cases. With the ability to compare any value, future opportunities exist; such as, routing checks without exposing data or even adding sensitive data within random data to further obscure information. Executing arbitrary calculations post homomorphic decryption demonstrates the operations and formulas that could be run against the encrypted data. The possibilities are endless.

9.1 Future Work

As the field of data security continues to grows, there are many areas where the work accomplished in this study can be expanded upon. OpenFHE [53] plans to implement bootstrapping for BFV in their library. Once available, bootstrapping to reduce the overall multiplicative depth could result in better performance and should be considered. In addition to bootstrapping, the library could develop and offer a form of division, making test case 2 complete for calculating the average of a set of encrypted data. Future additions to this library could open up many possible test improvements, pushing the limits HHE implementations.

Other opportunities of exploration come from the choices selected. Rasta [6] has multiple variants that cater to implementation with HHE instances. One of the older variants, or even a newer one, could better lend itself to a test instance such as this. The same logic applies to the HE scheme. The OpenFHE library was chosen for having the latest and greatest FHE schemes from second to fourth generation. With similar structured logic, translating the decryption circuit between instances is feasible. For example, BFV and BGV share many, if not all of the same function calls. By tweaking parameters and initialization, it stands to reason that the conversion from one to the other could be achieved.

Lastly, testing the performance and functionality on other devices may prove beneficial. Given the specs of the laptop used, it is feasible that a stronger processor could provide even faster results. The other question would be how the design behaves on hardware and the constraints presented by that. The options are endless.

Bibliography

[1] C. Tinker, "Exploring the application of homomorphic encryption for a cross domain solution." [Online]. Available: https://scholarworks.rit.edu/theses/9855/

[2] C. Tinker, K. Millar, A. Kaminsky, M. Kurdziel, M. Lukowiak, and S. Radziszowski, "Exploring the application of homomorphic encryption to a cross domain solution," in *MILCOM 2019 - 2019 IEEE Military Communications Conference (MILCOM)*, pp. 1–6, ISSN: 2155-7586.

[3] J. W. Bos, K. Lauter, J. Loftus, and M. Naehrig, "Improved security for a ring-based fully homomorphic encryption scheme." [Online]. Available: http://eprint.iacr.org/2013/075

[4] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The SIMON and SPECK families of lightweight block ciphers." [Online]. Available: http://eprint.iacr.org/2013/404

[5] "Submission requirements and evaluation criteria for the lightweight cryptography standardization process," p. 17. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf

[6] C. Dobraunig, M. Eichlseder, L. Grassi, V. Lallemand, G. Leander, E. List, F. Mendel, and C. Rechberger, "Rasta: A cipher with low ANDdepth and few ANDs per bit," report Number: 181. [Online]. Available: https://eprint.iacr.org/2018/181

[7] C. Dobraunig, L. Grassi, L. H. C. Rechberger, M. Schofnegger, and R. Walch, "Pasta: A case for hybrid homomorphic encryption," p. 42. [Online]. Available: https://eprint.iacr.org/2021/731

[8] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." [Online]. Available: http://eprint.iacr.org/2012/144

[9] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully homomorphic encryption without bootstrapping." [Online]. Available: http://eprint.iacr.org/2011/277

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast fully homomorphic encryption over the torus," report Number: 421. [Online]. Available: https://eprint.iacr.org/2018/421

[11] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Advances in Cryptology – EUROCRYPT 2015*, E. Oswald and M. Fischlin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 617–640.

[12] "Framework for hybrid homomorphic encryption," original-date: 2020-11-11T08:09:14Z. [Online]. Available: https://github.com/IAIK/hybrid-HE-framework

[13] "OpenFHE - open-source fully homomorphic encryption library," original-date: 2022-03-04T21:18:20Z. [Online]. Available: https://github.com/openfheorg/openfhe-development/blob/122f470e0dbf94688051ab852131ccc5d26be934/docs/index.rst

[14] C. of National Security Systems, "Committee of National Security Systems (CNSS) Glossary," no. 4009, 2015. [Online]. Available: https://cryptosmith.files.wordpress.com/2015/08/glossary-2015-cnss.pdf

[15] C. Chanderasekaran and W. Simpson, "Cross-domain solutions in an era of information sharing," p. 6.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, 3rd ed. MIT Press.

[17] Rafael. What is key length in cryptography and why is important? [Online]. Available: https://justcryptography.com/key-length/

[18] C. Paar and J. Pelzl, *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Berlin Heidelberg. [Online]. Available: http://link.springer.com/10.1007/978-3-642-04101-3

[19] K. A. McKay, L. Bassham, M. S. Turan, and N. Mouha, "Report on Lightweight Cryptography," p. NIST IR 8114. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf

[20] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schlaffer, "Ascon v1.2 : Submission to nist." [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf

[21] A. Chakraborti, N. Datta, A. Jha, and M. Nandi, "Structural Classification of Authenticated Encryption Schemes," p. 12. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/structural-classification-lwc2020.pdf

[22] P. Rogaway, "Authenticated-encryption with associated-data."

[23] A. Acar, H. Aksu, A. S. Uluagac, and M. Conti, "A Survey on Homomorphic Encryption Schemes: Theory and Implementation," vol. 51, no. 4, pp. 1–35. [Online]. Available: https://dl.acm.org/doi/10.1145/3214303

[24] P. V. Parmar, S. B. Padhar, S. N. Patel, N. I. Bhatt, and R. H. Jhaveri, "Survey of various homomorphic encryption algorithms and schemes," vol. 91, pp. 26–32.

[25] C. Gentry and D. Boneh, “A Fully Homomorphic Encryption Scheme,” vol. 20, no. 09, 2009.

[26] J. Benaloh, “Verifiable secret-ballot elections,” Ph.D. dissertation, September 1987. [Online]. Available: https://www.microsoft.com/en-us/research/publication/verifiable-secret-ballot-elections/

[27] I. Iliashenko and V. Zucca, “Faster homomorphic comparison operations for BGV and BFV.” [Online]. Available: http://eprint.iacr.org/2021/315

[28] A. C. Mert, E. Öztürk, and E. Savaş, “Design and implementation of encryption/decryption architectures for BFV homomorphic encryption scheme,” vol. 28, no. 2, pp. 353–362, conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[29] K. Lauter, M. Naehrig, and V. Vaikuntanathan, “Can homomorphic encryption be practical?” report Number: 405. [Online]. Available: https://eprint.iacr.org/2011/405

[30] C. Cid, J. P. Indrøy, and H. Raddum, “FASTA - a stream cipher for fast FHE evaluation,” report Number: 1205. [Online]. Available: https://eprint.iacr.org/2021/1205

[31] M. Sonmez Turan, K. McKay, D. Chang, C. Calik, L. Bassham, J. Kang, and J. Kelsey, “Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process.” [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf

[32] T. Beyne, L. Chen, Yu, C. Dobraunig, and B. Mennink, “Elephant v2.” [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf

[33] S. Banik, A. Chakraborti, T. Iwata, K. Minematsu, M. Nandi, T. Peyrin, Y. Sasaki, S. Meng Sim, and Y. Todo, “Gift-cofb v1.1.” [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf

[34] M. Hell, T. Johansson, A. Maximov, E. Ab, W. Meier, J. Sonnerup, and H. Yoshida, “Grain-128aeadv2 - a lightweight AEAD stream cipher,” p. 38. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf

[35] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer, “Isap v2.0 : Submission to nist.” [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/isap-spec-final.pdf

[36] Z. Bao, A. Chakraborti, N. Datta, J. Guo, M. Nandi, T. Peyrin, and K. Yasuda, "Photon-beetle authenticated encryption and hash family." [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf

[37] C. Guo, T. Iwata, M. Khairallah, K. Minematsu, and T. Peyrin, "Romulus," p. 57. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf

[38] C. Beierle, A. Biryukov, A. Moradi, L. Perrin, A. R. Shahmirzadi, A. Udovenko, and Q. Wang, "Schwaemm and esch: Lightweight authenticated encryption and hashing using the sparkle permutation family," p. 98. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf

[39] H. Wu and T. Huang, "TinyJAMBU: A family of lightweight authenticated encryption algorithms (version 2)," p. 40. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf

[40] J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, "Xoodyak, a lightweight cryptographic scheme," pp. 60–87. [Online]. Available: https://tosc.iacr.org/index.php/ToSC/article/view/8618

[41] "Benchmarking of lightweight cryptographic algorithms on microcontrollers," original-date: 2020-10-01T18:40:19Z. [Online]. Available: https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking

[42] S. Renner, E. Pozzobon, and J. Mottok, "A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers," vol. 12282, pp. 495–509, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-030-61078-4_28

[43] R. Weatherly. Lightweight cryptography primitives: Main page. [Online]. Available: https://rweather.github.io/lightweight-crypto/index.html

[44] K. Mohajerani, R. Haeussler, R. Nagpal, F. Farahmand, A. Abdulgadir, J.-P. Kaps, and K. Gaj, "FPGA benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process: Methodology, metrics, tools, and results." [Online]. Available: http://eprint.iacr.org/2020/1207

[45] Aagaard and N. Zidaric, "ASIC benchmarking of round 2 candidates in the NIST lightweight cryptography standardization process," p. 49. [Online]. Available: https://eprint.iacr.org/2021/049.pdf

[46] M. Khairallah, T. Peyrin, and A. Chattopadhyay, "Preliminary hardware benchmarking of a group of round 2 NIST lightweight AEAD candidates," p. 163. [Online]. Available: https://eprint.iacr.org/2020/1459.pdf

[47] M. J. Dworkin, “SHA-3 standard: Permutation-based hash and extendable-output functions,” last Modified: 2018-11-10T10:11-05:00 Publisher: Morris J. Dworkin. [Online]. Available: https://www.nist.gov/publications/sha-3-standard-permutation-based-hash-and-extendable-output-functions

[48] Z. Brakerski and V. Vaikuntanathan, “Efficient fully homomorphic encryption from (standard) LWE.” [Online]. Available: https://eprint.iacr.org/2011/344

[49] C. Gentry, S. Halevi, and N. P. Smart, “Homomorphic evaluation of the AES circuit,” report Number: 099. [Online]. Available: https://eprint.iacr.org/2012/099

[50] Z. Brakerski, “Fully homomorphic encryption without modulus switching from classical GapSVP.” [Online]. Available: http://eprint.iacr.org/2012/078

[51] J.-C. Bajard, J. Eynard, A. Hasan, and V. Zucca, “A full RNS variant of FV like somewhat homomorphic encryption schemes.” [Online]. Available: https://eprint.iacr.org/2016/510

[52] S. Halevi, Y. Polyakov, and V. Shoup, “An improved RNS variant of the BFV homomorphic encryption scheme,” report Number: 117. [Online]. Available: https://eprint.iacr.org/2018/117

[53] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V, K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, “OpenFHE: Open-source fully homomorphic encryption library,” report Number: 915. [Online]. Available: https://eprint.iacr.org/2022/915

[54] S. Halevi and V. Shoup, “Design and implementation of HElib: a homomorphic encryption library,” report Number: 1481. [Online]. Available: https://eprint.iacr.org/2020/1481

[55] “Microsoft SEAL (release 4.1),” https://github.com/Microsoft/SEAL, Jan. 2023, microsoft Research, Redmond, WA.

[56] PALISADE homomorphic encryption software library – an open-source lattice crypto software library. [Online]. Available: https://palisade-crypto.org/

[57] J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Homomorphic encryption for arithmetic of approximate numbers.” [Online]. Available: https://eprint.iacr.org/2016/421

Appendix A: Source Code

```cpp
// myImpl.cpp
#include <cstring>
#include <chrono>
#include <iostream>
#include <iterator>
#include <string>
#include <vector>
#include <ctime>

#include "cryptocontext.h"
#include "gen-cryptocontext.h"
#include "openfhe.h"

#include "include/plain.h"

#define PRINT 1
#define DEBUG 0
int g_test = 0;

// Global Variables to compare performance
usint ptm = 65537; // PTM recommended for integers
uint64_t nonce = 123456789; // Arbitrary nonce used for testing

//Two states arrays stored in a vector
std::vector<Ciphertext<DCRTPoly>> states(2, 0);
//Vector the length of the plaintext/ciphertext
std::vector<uint64_t> randRay(MY_PARAMS.cipher_size, 0);
//Final ciphertext after HHE_Decrypt
Ciphertext<DCRTPoly> final_ct;
//HE Key Pair
KeyPair<DCRTPoly> keyPair;
//HE CryptoContext
CryptoContext<DCRTPoly> cc ;

/*
 * Main function to run test cases appropriately
 */
int main(int argc, char* argv[]) {
    unsigned int mdepth = 0;
    // If 3-round test, set PASTA_3 variables
    if (*argv[1] == '3'){
        cout << "3" << endl;
        plaintext = plaintext3;
        MY_PARAMS = PASTA3_PARAMS;
        ROUNDS = PASTA_3::PASTA_R;
        mdepth = pasta3_depth;
    }
    // If 4-round test, set PASTA_4 variables
    else {
        plaintext = plaintext4;
        MY_PARAMS = PASTA4_PARAMS;
        ROUNDS = PASTA_4::PASTA_R;
        mdepth = pasta4_depth;
    }
    ZpCipherParams params = MY_PARAMS;
    std::vector<int> indexes = [1, 5, 10];
    std::vector<int64_t> compVals = [plaintext.plaintext[1], plaintext.plaintext[5],
    plaintext.plaintext[10]]

    switch(*argv[2]){
        case '1':
        default:
            mdepth += test1_depth;         // Add necessary extra depth from base value
            prepare_hhe(params, mdepth);   // Run HHE_Decryt
            test_case_1(params);           // Confirm successful HHE_Decryt
            break;
        case '2':
            mdepth += test2_depth;         // Add necessary extra depth from base value
            prepare_hhe(params, mdepth);   // Run HHE_Decryt
```

Appendix A: Source Code

```
                test_case_2(params);           // Square and sum ciphertext
                break;
            case '3':
                mdepth += test3_depth;         // Add necessary extra depth from base value
                prepare_hhe(params, mdepth);   // Run HHE_Decryt
                cds_example(params, indexes, compVals); // Compare values at indexes
                break;
        }
        return 0;
    }


    /*
     * prepare_hhe(ZpCipherParams params, unsigned int mdepth)
     *
     * Encrypts the plaintext using Pasta with the correct params for rounds
     * Calls the homomorphic decryption circuit to convert from Pasta.Enc(ptxt) to HE.Enc(ptxt)
     *
     * ZpCipherParams params - Pasta parameters for plaintext, key, and ciphertext size
     * unsigned int mdepth - multiplicative depth
     */
    void prepare_hhe(ZpCipherParams params, unsigned int mdepth){
        // Timing logic to get start time
        auto start = std::chrono::system_clock::now();

        size_t size = plaintext.plaintext.size();
        size_t num_block = ceil((double)size/ params.plain_size);
        Pasta pasta(plaintext.key, ptm);
        std::vector<uint64_t> ctxt = plaintext.plaintext;

        // Pasta Encryption - taken from [12]
        for (uint64_t i = 0; i < num_block; i++){
            block keystream = pasta.keystream(nonce, i);
            for(size_t j = i * params.plain_size; j < (i + 1) * params.plain_size && j < size; j++){
                ctxt[j] = (ctxt[j] + keystream[j - i * params.plain_size]) % ptm;
            }
        }

        // Timing logic to get end time, takes difference of start and end to get elapsed time
        auto end = std::chrono::system_clock::now();
        std::chrono::duration<double> elapsed_seconds = end-start;

        if(PRINT)  {
            cout << "Plaintext: " << plaintext.plaintext << endl << endl;
            cout << "Key: " << plaintext.key << endl << endl;
            cout << "Ciphertext: " << ctxt << endl << endl;
            cout << "Time To Encrypt: " <<  (elapsed_seconds.count()) << " sec" << endl << endl;
        }

        //Convert the unsigned ciphertext array to signed array
        std::vector<int64_t> signed_ctxt;
        for(int i = 0; i < params.cipher_size; i++){
            signed_ctxt.push_back((int64_t)ctxt[i]);
        }
        std::vector<int64_t> signed_key;
        for(int i = 0; i < params.key_size; i++){
            signed_key.push_back((int64_t)plaintext.key[i]);
        }

        // Timing start for homomorphic decryption circuit execution
        start = std::chrono::system_clock::now();

        //Call hybrib homomorphic decryption circuit to get HE.Enc(ptxt) from Pastas ctxt
        final_ct = HHE_Decryt(signed_ctxt, signed_key, params, mdepth);

```

```
    // Timing logic to get end time, takes difference of start and end to get elapsed time
    end = std::chrono::system_clock::now();
    elapsed_seconds = end-start;
    cout << "Time To Decrypt: " <<  (elapsed_seconds.count()/60) << " min" << endl << endl;
}


/*
 * GenerateBFVrnsContext(usint ptm, unsigned int adepth, unsigned int mdepth)
 *
 * Generates the cryptocontext for BFV (OpenFHEs accessor to BFV functions)
 *
 * usint ptm - plaintext modulus
 * unsigned int mdepth - multiplicative depth
 * ret - CryptoContext<DCRTPoly> BFV crypto context (access to BFV functions)
 */
CryptoContext<DCRTPoly> GenerateBFVrnsContext(usint ptm, unsigned int mdepth) {

    // Start timing for generating BFVs crytocontext
    auto start = std::chrono::system_clock::now();
    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds;

    cout << "Generating BFV Crypto Context...";
    start = std::chrono::system_clock::now();

    // Set parameters for BFV, the plaintext modulus, multiplicative depth, relineariation degree,
    // and security level
    CCParams<CryptoContextBFVRNS> parameters;
    parameters.SetPlaintextModulus(ptm);
    parameters.SetMultiplicativeDepth(mdepth);
    parameters.SetMaxRelinSkDeg(3);
    parameters.SetSecurityLevel(HEStd_128_classic);

    CryptoContext<DCRTPoly> cc = GenCryptoContext(parameters);

    // enable features that you wish to use
    cc->Enable(PKE);           // Public Key Encryption
    cc->Enable(KEYSWITCH);     // Key Switching
    cc->Enable(LEVELEDSHE);    // leveled HE
    cc->Enable(ADVANCEDSHE);   // Advanced HE

    // Generate Secert and Public key pair
    keyPair = cc->KeyGen();

    // Create array of rotation values to generate rotation keys
    std::vector<int> rot(MY_PARAMS.key_size/2, 0);
    for (int i = (MY_PARAMS.key_size/2 - 1); i >= 0; i--){
        rot[(MY_PARAMS.key_size/2 -1) - i] = i - (MY_PARAMS.key_size/2 - 1);
    }
    rot.push_back(30);

    // Generate keys for rotating, summation, and multiplication
    cc->EvalRotateKeyGen(keyPair.secretKey, rot);
    cc->EvalSumKeyGen(keyPair.secretKey);
    cc->EvalMultKeyGen(keyPair.secretKey);

    // Timing logic to get end time, takes difference of start and end to get elapsed time
    end = std::chrono::system_clock::now();
    elapsed_seconds = end -start;
    cout << "done - Time To Generate Context: " <<  (elapsed_seconds.count()/60) << " min" << endl << endl;

#if PRINT
  std::cout << "\nParameters BFVrns for depth " << mdepth << std::endl;
```

```
  std::cout << "p = " << cc->GetCryptoParameters()->GetPlaintextModulus() <<
  std::endl; std::cout << "n = " <<
  cc->GetCryptoParameters()->GetElementParams()->GetCyclotomicOrder() / 2 <<
  std::endl; std::cout << "log2 q = " <<
  log2(cc->GetCryptoParameters()->GetElementParams()->GetModulus().ConvertToDouble())
  << "\n" << std::endl;
#endif

  return cc;
}

/*
 * add_rc(char st)
 *
 * Add the round constant to the state
 *
 * char st - 0 or 1 for state1 or state2
 */
void add_rc(char st){
    // Homomorphically pack the random vector into a plaintext
    std::vector<int64_t> rands = unsigned2signed(randRay);
    Plaintext rand_pt = cc->MakePackedPlaintext(rands);
    // Add the random vector to the state and store back in the state
    states[st] = cc->EvalAdd(states[st], rand_pt);
}

/*
 * matmul(char st)
 *
 * Perform matrix multiplication on the state
 *
 * char st - 0 or 1 for state1 or state2
 */
void matmul(char st) {

    // Initialize temporary vectors/plaintext for use
    std::vector<int64_t> test(MY_PARAMS.key_size/2, 0);
    std::vector<int64_t> mask(MY_PARAMS.key_size/2, 0);
    mask[0] = 1;
    Ciphertext<DCRTPoly> temp = cc->Encrypt(keyPair.publicKey, cc->MakePackedPlaintext(
    test));

    // Set current row to the random vector
    std::vector<uint64_t> curr_row = randRay;
    // For every element in the ciphertext
    for (uint16_t i = 0; i < MY_PARAMS.cipher_size; i++) {
        // Format the current row into a packed plaintext
        std::vector<int64_t> signed_curr_row = unsigned2signed(curr_row);
        Plaintext row = cc->MakePackedPlaintext(signed_curr_row);

        // Multiplying row and state, mod included in calculation
        Ciphertext<DCRTPoly> mult = cc->EvalMult(states[st], row);
        // Get the sum of all elements in ciphertext
        Ciphertext<DCRTPoly> sum = cc->EvalSum(mult, MY_PARAMS.cipher_size);
        // Multiply by mask to get only first element
        Ciphertext<DCRTPoly> masked = cc->EvalMult(sum, cc->MakePackedPlaintext(mask));
        // Rotate first value to the correct index location
        Ciphertext<DCRTPoly> sumMult = cc->EvalRotate(masked, 0 - i);
        // Add the calculated value to the temp ciphertext at i index
        temp = cc->EvalAdd(temp, sumMult);

        // Calculate the next row value
        if (i !=  MY_PARAMS.key_size/2 - 1) {
            std::vector<uint64_t> temp_row;
            for (auto j = 0; j < MY_PARAMS.key_size/2; j++) {
                uint64_t tmp = ((uint128_t)(randRay[j]) * curr_row[ MY_PARAMS.key_size/2
                - 1]) % ptm;
                if (j) {
                    tmp = (tmp + curr_row[j - 1]) % ptm;
```

```
                }
                temp_row.push_back(tmp);
            }
            curr_row = temp_row;
        }
    }
    // Set state ciphertext to the temp ciphertext
    states[st] = temp;
}

/*
 * mix()
 *
 * Mix state1 and state2
 */
void mix(){
    // Add the states together
    Ciphertext<DCRTPoly> sum = cc->EvalAdd(states[0], states[1]);
    cc->EvalAddInPlace(states[0], sum);  // Add sum to state1
    cc->EvalAddInPlace(states[1], sum);  // Add sum to state2
}

/*
 * sbox_cube(char st)
 *
 * Sbox Cube lookup on the state
 *
 * char st - 0 or 1 for state1 or state2
 */
void sbox_cube(char st){
    //Each element squared and stored in its own slot
    Ciphertext<DCRTPoly> square = cc->EvalSquare(states[st]);
    // Multiply the squared and state to get cubed result, store in state
    states[st] = cc->EvalMult(states[st], square);
}

/*
 * sbox_feistel(char st)
 *
 * Sbox Feistel lookup on the state
 *
 * char st - 0 or 1 for state1 or state2
 */
void sbox_feistel(char st){
    // Inintalize mask plaintext
    std::vector<int64_t> mask(MY_PARAMS.key_size + 1, 1);
    mask[MY_PARAMS.key_size] = 0;
    Plaintext mask_pt = cc->MakePackedPlaintext(mask);

    //Each element squared and stored in its own slot
    Ciphertext<DCRTPoly> square = cc->EvalSquare(states[st]);
    Ciphertext<DCRTPoly> shifted = cc->EvalRotate(square, -1); //Shift right 1

    // Multiply result and mask to keep expected amount of words
    shifted = cc->EvalMult(shifted, mask_pt);
    // Add shifted and the current state and store back in the state
    cc->EvalAddInPlace(states[st], shifted);
}

/*
 * HHE_Decryt(std::vector<int64_t> cipher, std::vector<int64_t> key, ZpCipherParams params, unsigned int mdepth)
 *
 * Hybrid Homomorphic Decryption Circuit
 *
 * std::vector<int64_t> cipher - Pasta's encrypted ciphertext properly formatted
 * std::vector<int64_t> key - Pasta's key properly formatted
 * ZpCipherParams params - Pasta parameters for plaintext, key, and ciphertext size
 * unsigned int mdepth - multiplicative depth
```

```
 *
 * ret - Ciphertext<DCRTPoly> - return the final ciphertext, solely encrypted by the HE scheme
 */
Ciphertext<DCRTPoly> HHE_Decryt(std::vector<int64_t> cipher, std::vector<int64_t> key, ZpCipherParams params, unsigned int mdepth){

    // Generate the CryptoContext for BFV
    cc = GenerateBFVrnsContext(ptm, mdepth);

    // Initialize values for later use
    std::vector<int64_t> blank(params.plain_size, 0);
    std::vector<int64_t> key1, key2;
    for(int i = 0; i < params.key_size/2; i++) key1.push_back(key[i]);
    for(int i = params.key_size/2; i < params.key_size; i++) key2.push_back(key[i]);

    // Homomorphically pack Pasta's ciphertext and key into HE plaintexts
    Plaintext pt = cc->MakePackedPlaintext(cipher);
    Plaintext key_pt1 = cc->MakePackedPlaintext(key1);
    Plaintext key_pt2 = cc->MakePackedPlaintext(key2);
    // Homomorphically encrypt the packed plaintexts
    Ciphertext<DCRTPoly> ct = cc->Encrypt(keyPair.publicKey, pt);
    Ciphertext<DCRTPoly> res = cc->Encrypt(keyPair.publicKey, cc->MakePackedPlaintext(blank));

    // Initialize values related to size and Pasta's SHAKE_128 calls
    size_t size = cipher.size();
    size_t num_block = ceil((double)size / params.cipher_size);
    Pasta pasta(plaintext.key, ptm);

    if (PRINT) {
        cout << "Cipher Size: " << size << endl;
        cout << "Params Cipher Size: " << params.cipher_size << endl;
        cout << "Params Plain Size: " << params.plain_size << endl;
        cout << "Num Blocks: " << num_block << endl;
    }

    //Main decryption circuit - decrypt each block depending on size of ciphertext
    for (uint64_t b = 0; b < num_block; b++) {
        pasta.init_shake(nonce, b);
        // Homomorphically encrypt Pasta's key into two states
        states[0] = cc->Encrypt(keyPair.publicKey, key_pt1);
        states[1] = cc->Encrypt(keyPair.publicKey, key_pt2);

        // Ininitalize timing variables
        auto start = std::chrono::system_clock::now();
        auto end = std::chrono::system_clock::now();
        std::chrono::duration<double> elapsed_seconds;

        // Perform r rounds
        for (uint8_t r = 0; r < ROUNDS; r++){
            std::cout << "Round " << (int)r + 1 << std::endl;

            // Start of how long matrix multiplication takes for both states
            start = std::chrono::system_clock::now();
            randRay = pasta.get_random_vector(false); // Generate new random vector
            cout << "\tMatmul...." ;
            matmul(0);                                // Execute matmul on state1
            randRay = pasta.get_random_vector(false); // Generate new random vector
            matmul(1);                                // Execute matmul on state2
            end = std::chrono::system_clock::now();   // End time
            elapsed_seconds = end-start;              // Get time it took to do both matmul
            cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;
            if (DEBUG){
                decrypt_print_state(0);
                decrypt_print_state(1);
            }

```

Appendix A: Source Code

```
            // Start of how long add_rc takes for both states
            start = std::chrono::system_clock::now();
            cout << "\tAddRc...." ;
            randRay = pasta.get_random_vector(false); // Generate new random vector
            add_rc(0);                                // Execute add_rc on state 1
            randRay = pasta.get_random_vector(false); // Generate new random vector
            add_rc(1);                                // Execute add_rc on state 1
            end = std::chrono::system_clock::now();   // End time
            elapsed_seconds = end-start;              // Get time it took to do both add_rc
            cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;
            if (DEBUG){
                decrypt_print_state(0);
                decrypt_print_state(1);
            }

            // Start of how long mixing states takes
            start = std::chrono::system_clock::now();
            cout << "\tMix...." ;
            mix();                                    // Execute mix on both states
            end = std::chrono::system_clock::now();   // End time
            elapsed_seconds = end-start;              // Get time it took to do mix of states
            cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;

            // Start of how long sbox takes
            start = std::chrono::system_clock::now();
            if(r == ROUNDS - 1) {
                cout << "\tSbox Cube...." ;
                sbox_cube(0);                       // Execute Sbox Cube on state1
                sbox_cube(1);                       // Execute Sbox Cube on state2
            } else {
                cout << "\tSbox feistal...." ;
                sbox_feistel(0);                    // Execute Sbox Feistel on state1
                sbox_feistel(1);                    // Execute Sbox Feistel on state2
            }
            end = std::chrono::system_clock::now();   // End time
            elapsed_seconds = end-start;              // Get time it took to do both sbox lookups
            cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;

            if (DEBUG){
                decrypt_print_state(0);
                decrypt_print_state(1);
            }
        }

        cout << "Final Linear Layer" << endl;
        // Start of how long matrix multiplication takes for both states
        start = std::chrono::system_clock::now();
        randRay = pasta.get_random_vector(false); // Generate new random vector
        cout << "\tMatmul...." ;
        matmul(0);                                // Execute matmul on state1
        randRay = pasta.get_random_vector(false); // Generate new random vector
        matmul(1);                                // Execute matmul on state2
        end = std::chrono::system_clock::now();   // End time
        elapsed_seconds = end-start;              // Get time it took to do both matmul
        cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;
        if (DEBUG){
            decrypt_print_state(0);
        }
        // Start of how long add_rc takes for both states
        start = std::chrono::system_clock::now();
        cout << "\tAddRc...." ;
        randRay = pasta.get_random_vector(false); // Generate new random vector
        add_rc(0);                                // Execute add_rc on state 1
        randRay = pasta.get_random_vector(false); // Generate new random vector
        add_rc(1);                                // Execute add_rc on state 1
        end = std::chrono::system_clock::now();   // End time
```

```
        elapsed_seconds = end-start;                // Get time it took to do both add_rc
        cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;
        if (DEBUG){
            decrypt_print_state(0);
        }
        // Start of how long mixing states takes
        start = std::chrono::system_clock::now();
        cout << "\tMix...." ;
        mix();                                      // Execute mix on both states
        end = std::chrono::system_clock::now();     // End time
        elapsed_seconds = end-start;                // Get time it took to do mix of states
        cout << "done - Time " <<  elapsed_seconds.count() << " seconds" << endl;

        // Subract state1, the key stream, from the ciphertext to get homomorphically
        // encrypted ciphertext only
        res = cc->EvalSub(ct, states[0]);

        if (DEBUG){
            decrypt_print_state(0);
            decrypt_print(ct);
        }

    }
    return res;
}

/*
 * test_case_1(ZpCipherParams params)
 *
 * Test Case 1, Verify that the HHE_Decryt worked as expected, result should
 *              match the plaintext
 *
 * ZpCipherParams params - Pasta parameters for plaintext, key, and ciphertext size
 */
void test_case_1(ZpCipherParams params){
    cout << "Test 1 - Confirming Decrypted Results" << endl;

    // HE Decryption
    Plaintext ptxt;
    cc->Decrypt(keyPair.secretKey, final_ct, &ptxt);
    // Convert from int64_t to uint64_t to compare to original values
    std::vector<uint64_t> plain = signed2unsigned(ptxt->GetPackedValue());

    // Remove excess zero slots created at end of array
    std::vector<uint64_t> sliced;
    for(int i = 0; i < params.plain_size; i++) sliced.push_back(plain[i]);
    cout << sliced << endl;


    bool fail = false;
    // Check if there are values beyond the specified plaintext size
    if (plain.size() > params.plain_size){
        for (int i = 0; i < params.plain_size; i++){
            if (plain[i] != plaintext.plaintext[i]){
                cout << "At index " << i << " got " << plain[i] << " instead of " <<
                plaintext.plaintext[i] << endl;
                fail = true;
            }
        }
        if (plain[params.plain_size] != 0) {
            cout << "Result has more values than expected " << params.plain_size << endl;
            fail = true;
        }
    }
    else {
        // Check that the arrays are identical
        if (plain != plaintext.plaintext) fail = true;
    }

```

```
    // Report Status
    if (!fail){
        cout << "SUCCESS" << endl;
    } else {
        cout << "FAIL" << endl;
    }
}

/*
 * test_case_2(ZpCipherParams params)
 *
 * Test Case 2, square the ciphertext then take the sum of all elements
 *
 * ZpCipherParams params - Pasta parameters for plaintext, key, and ciphertext size
 */
void test_case_2(ZpCipherParams params){
    int ret = 0;

    // Initialize timing variables
    auto start = std::chrono::system_clock::now();
    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds;
    // Initialize mask vector
    std::vector<int64_t> mask (params.plain_size, 0);
    mask[0] = 1;

    // Start timing
    start = std::chrono::system_clock::now();

    // Square the ciphertext
    Ciphertext<DCRTPoly> mult = cc->EvalSquare(final_ct);
    // Take the sum of all elements in ciphertext
    Ciphertext<DCRTPoly> sum = cc->EvalSum(mult,  params.cipher_size);
    // Multiply by the mask to obtain only the first value
    Ciphertext<DCRTPoly> first = cc->EvalMult(sum, cc->MakePackedPlaintext(mask));

    Plaintext res;
    // HE Decryption of the resulting ciphertext
    cc->Decrypt(keyPair.secretKey, first, &res);
    // Convert to proper signage
    ret = signed2unsigned(res->GetPackedValue())[0];
    // Stop timing
    end = std::chrono::system_clock::now();
    // Get elapsed time
    elapsed_seconds = end-start;
    decrypt_print(mult);
    decrypt_print(sum);
    cout << "sumOf(ct * ct) = " << ret << " - Time " << elapsed_seconds.count() << "
    seconds" << endl;
}

/*
 * cds_example(ZpCipherParams params, int index, int64_t compVal)
 *
 * Implements a mock use case of looking for a specific value in the array
 *
 * ZpCipherParams params - Pasta parameters for plaintext, key, and ciphertext size
 * std::vector<int> indexes - index at which to check the value
 * std::vector<int64_t> indexes compVal - value for comparison
 */
void cds_example(ZpCipherParams params, std::vector<int> indexes, std::vector<int64_t>
compVal){
    int ret = 0;
    // Initialize timing variables
    auto start = std::chrono::system_clock::now();
    auto end = std::chrono::system_clock::now();
    std::chrono::duration<double> elapsed_seconds;

    // Start time
```

```
        start = std::chrono::system_clock::now();
        cout << "Comparing index " << index << " for value " << compVal << endl;
        // Initalize the mask vector plaintext for which index to checl
        std::vector<int64_t> vec(params.cipher_size, 0);
        for (int i = 0; i < indexes.size(); i++){
            vec[indexes[i]] = 1;
        }
        Plaintext vecpt = cc->MakePackedPlaintext(vec);
        // Initialize vector plaintext with the value at the correct index
        for (int i = 0; i < indexes.size(); i++){
            vec[indexes[i]] = compVal[i];
        }
        Plaintext ptt  = cc->MakePackedPlaintext(vec);

        // Multiply by the mask plaintext to get value at index
        Ciphertext<DCRTPoly> masked = cc->EvalMult(final_ct, vecpt);
        // Subtract the actual by the expected
        Ciphertext<DCRTPoly> diff = cc->EvalSub(masked, ptt);
        decrypt_print(diff);

        Plaintext res;
        // HE Decrypt the ciphertext
        cc->Decrypt(keyPair.secretKey, diff, &res);
        // Convert to correct formatting type
        std::vector<uint64_t> plain = signed2unsigned(res->GetPackedValue());
        // Check that the first element equals 0, i.e. it is a match
        for (int i = 0; i < indexes.size(); i++){
            if (plain[indexes[i]] != 0){
                ret = 0;
                break;
            }
            else{
                ret = 1;
            }
        }
        // Stop time
        end = std::chrono::system_clock::now();
        elapsed_seconds = end - start;


        if (ret <= 0){
            cout << "FAILURE - NO MATCH - Time " << elapsed_seconds.count() << " seconds" <<
            endl;
        } else {
            elapsed_seconds = end-start;
            cout << "SUCCESS - FOUND MATCH - Time " << elapsed_seconds.count() << " seconds"
            << endl;
        }
    }

```