A DISSERTATION PRESENTED TO
THE UNIVERSITY OF DUBLIN, TRINITY COLLEGE DUBLIN

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTERS IN ELECTRONIC & COMPUTER ENGINEERING

Fully Homomorphic Encryption and Secure Enclaves in ACH Payment Systems: A Multi-Party Computation Strategy

Leveraging the BGV Scheme and Secure Enclaves for Payment Security

TRINITY COLLEGE DUBLIN

2024

SEAN FAHEY

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Date: ______/______/__________.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Ciaran McGoldrick for his guidance and support throughout the completion of this work. I would also like to thank the departments of engineering and computer science at Trinity College for their assistance and resources throughout the program.

"The Network is the Computer"

- John Gage, Sun Microsystems

Table of Contents

1	**Abstract**	**5**
2	**Introduction**	**6**
3	**Background**	**7**
3.1	ACH	7
3.2	Garbled Circuits	7
3.3	Fully Homomorphic Encryption	8
	Learning With Errors	8
	Ring Learning with Errors	8
	Brakerski-Gentry-Vaikuntanathan (BGV) Scheme	10
3.4	Secure Enclaves	14
3.5	Use Cases	14
4	**Implementation**	**15**
4.1	Scheme Selection	15
4.2	Tools & Libraries	15
4.3	Parameter Selection	16
4.4	Processing & Validation	20
	Networking	20
	Homomorphic Comparison	21
	Key Generation And Storage	22
5	**Discussion**	**25**
5.1	Computation Time	25
	Impact of Local Process Allocation on Compute Time	25
	Suboptimal Resource Allocation and Resolution	26
	Regulatory and Compliance Implications	26
	Scalability Considerations	27
6	**Conclusion**	**28**
7	**Further Work**	**29**
	MPI Parallelisation	29
	FHE in secure enclaves	29
8	**Appendix A**	**30**
8.1	1.1	30
9	**Appendix B**	**31**
9.1	1.1	31
9.2	1.2	32

REFERENCES **33**

1 Abstract

The Automated Clearing House (ACH) system, a cornerstone of financial transactions in the United States, processes vast sums daily, necessitating robust security measures to safeguard against fraud, unauthorised access, and data breaches. Traditional security mechanisms, while effective to a degree, are increasingly challenged by sophisticated cyber threats and the inherent vulnerabilities of centralized systems. This paper proposes a novel approach to fortify ACH payment security by integrating Fully Homomorphic Encryption (FHE) and secure enclaves into the transaction processing workflow.

Leveraging the Brakerski-Gentry-Vaikuntanathan (BGV) scheme of FHE, this study demonstrates how encrypted data can be securely computed without decryption, thus maintaining data confidentiality even on untrusted platforms. Additionally, the use of Intel Software Guard Extensions (SGX) secure enclaves ensures that sensitive data and cryptographic operations are isolated from the rest of the system, providing a fortified layer of security against both external attacks and insider threats.

The integration of these technologies addresses the dual challenges of transparency and security posed by the emerging cryptocurrency frameworks and the traditional ACH system. By enabling secure, trustless validation of transactions, this approach not only enhances the security of the ACH network but also aligns with the transparency and programmability seen in decentralized financial systems.

This paper details the implementation of the BGV scheme within the ACH transaction processing, discusses the architectural considerations for deploying secure enclaves, and evaluates the system's performance and security through various computational and theoretical analyses. The findings suggest that the proposed solution significantly mitigates the risk of data breaches and unauthorised access, thereby strengthening the trust and reliability of the ACH system in the digital age.

2 INTRODUCTION

The demand for secure payment systems has drastically increased ever since payments were first virtualised. The concept of secure payments now encompasses a wide range technologies and protocols designed to ensure the security and privacy of financial transactions. Governments have enacted various Know Your Customer (KYC) laws for private institutions, and many state bodies have launched their own services like FedNow and FedWire to centralise payments in an effort to prevent fraud.

Furthermore, the emergence of cryptocurrencies has highlighted the demand for transparent, trusted payment systems. While cryptocurrencies offer a decentralised and openly programmable framework, they also present challenges in scalability and regulatory compliance [1] that are currently addressed by legacy systems including ACH. By integrating Fully Homomorphic Encryption (FHE) into ACH, we aim to merge the transparency and security offered by blockchain technology with the efficiency and regulatory compliance of established financial networks.

The current ACH system faces several security challenges, including the risk of unauthorised access, data breaches, and fraudulent transactions [2]. Existing security measures such as end-to-end encryption provide some level of protection, but they may not be sufficient to address the increasing number of cyber attacks. The centralised nature of the ACH system makes it a prime target for attackers, as a single point of failure could potentially compromise the entire network.

The proposed solution aims to enhance the security of ACH payments by leveraging fully homomorphic encryption and secure enclaves for trustless payment validation. This approach enables secure computation on encrypted data, thereby reducing the risk of data breaches, and minimising the consequences in the event of a breach.

3 Background

3.1 ACH

The Automated Clearing House (ACH) system is a payments network that facilitates electronic financial transactions in the United States, processing an annual throughput exceeding 55 trillion USD in 2019 [3]. This system is governed by a not-for-profit organisation, NACHA (National Automated Clearing House Association), which sets the rules and standards. This is separate from the ACH operators, which are primarily the Federal Reserve or The Clearing House, a private company owned by several of the largest US banks. In the ACH system, the Originator initiates a transaction by submitting payment instructions to a financial institution where they have an account, which is called the Originating Depository Financial Institution (ODFI). The ODFI aggregates transactions into batches and forwards them to an ACH Operator at set intervals, usually twice daily. The ACH operator sorts and distributes the transactions to the corresponding Receiving Depository Financial Institution (RDFI). The RDFI then credits or debits the receiver's account based on the transaction code. ACH is used for several types of transactions, including direct deposits, payroll, consumer bills, taxes, etc.
A key aspect of ACH is that after the ACH Operator processes the batches, it facilitates the settlement among the financial institutions involved, as the transactions between any given ODFI and RDFI will result in a difference that must be settled. This settlement typically occurs at the end of the processing day through the Federal Reserve's National Settlement Service. The settlement process involves adjusting the reserve accounts that each financial institution maintains at the Federal Reserve. Hence, final settlement happens at the Federal Reserve by adjusting accounts of registered institutions and not, as is often misstated, by ACH itself.

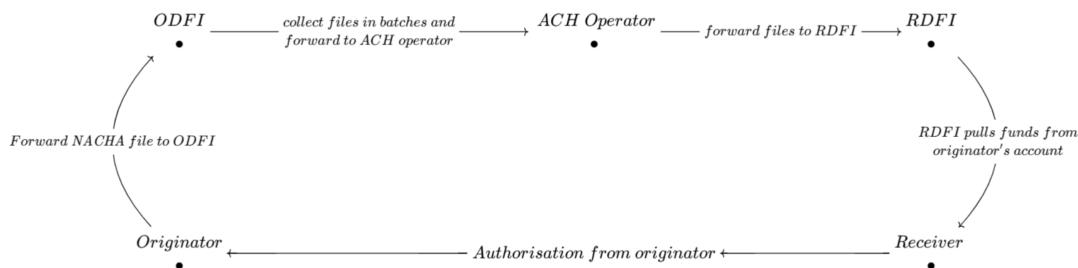

Figure 1 – The ACH Payment Process Flow

3.2 Garbled Circuits

A garbled circuit is a cryptographic technique used for secure two-party computation, where two parties can jointly compute a function without revealing their inputs to each other [4]. In a garbled circuit, the function to be computed is represented as a boolean circuit, and is passed in full from one party to another for computation.

The circuit is first garbled, meaning that the gates are encrypted in such a way that the inputs and outputs of each gate are represented by random strings or ciphertexts. This concept has been used for more advanced methods for secure multi-party computation such as homomorphic encryption.

3.3 Fully Homomorphic Encryption

Craig Gentry's 2009 paper [5] introduced the first fully homomorphic encryption scheme, allowing both addition and multiplication on encrypted data without decryption. Consider two plaintext messages, m_1 and m_2 be two plaintext messages, and $E(x)$ be the encryption function. The corresponding ciphertexts c_1 and c_2 are obtained by encrypting m_1 and m_2, respectively:

$$c_1 = E(m_1) \tag{3.1}$$

$$c_2 = E(m_2) \tag{3.2}$$

The homomorphic property of the encryption scheme allows for computations on the ciphertexts such that:

$$E(m_1 + m_2) = c_1 + c_2 \tag{3.3}$$

Similarly, for multiplication:

$$E(m_1 \times m_2) = c_1 \times c_2 \tag{3.4}$$

This property allows computations to be performed on the encrypted data without the need to decrypt it first, enabling secure computation on untrusted platforms.

Learning With Errors

All encryption schemes are based on a problem which is hard to solve without knowledge of a secret key. One such example is the Learning with Errors (LWE) problem [6], which is described as follows: Given m samples (a,b), the challenge is to recover the secret vector s or to solve for s in the presence of noise e. More formally, let q be a prime number representing the modulus. We choose a prime because, by definition, it ensures that all non-zero elements in the set {1, 2, ..., q-1} are coprime to q, hence forming a Galois Field $\mathbb{F}_q$. The secret vector $\mathbf{s} \in \mathbb{Z}_q^n$ is chosen uniformly at random, where n is the dimension of s, which means sampling n integers in $\mathbb{F}_q$. For each sample, a new vector $\mathbf{a} \in \mathbb{Z}_q^n$ is chosen uniformly at random, along with a noise term e, sampled from a noise distribution such as the discrete Gaussian distribution. The public samples are then given by

$$(\mathbf{a}, b = \langle \mathbf{a}, \mathbf{s} \rangle + e \mod q)$$

where a unique vector s can reproduce the error distribution from all e terms in the lattice.

Ring Learning with Errors

The Learning with Errors was defined in Galois fields, but in our fully homomorphic scheme, we will use a variant called Ring Learning with Errors (RLWE) which deals with cyclotomic fields. A cyclotomic field is constructed by extending the field of rational numbers $\mathbb{Q}$ by adjoining an algebraic term, to get $\mathbb{Q}(\zeta_m)$, where ζ_m is a primitive m^{th} root of unity, defined as solutions to the equation

$$\zeta^m = 1 \tag{3.5}$$

A root ζ_k is primitive for m if and only if $\gcd(k,m) = 1$.

Euler's totient function, denoted by $\phi(m)$, is defined as the count of integers k such that $1 \leq k \leq m$ and $\gcd(k,m) = 1$. It is given by the formula:

$$\phi(m) = m \prod_{p|m} \left(1 - \frac{1}{p}\right) \tag{3.6}$$

where the product is taken over all distinct prime numbers p that divide m. The mth cyclotomic polynomial is given by:

$$\Phi_m(x) = \prod_{\substack{1 \leq k \leq m \\ \gcd(k,m)=1}} (x - \zeta_k) \tag{3.7}$$

Hence, $\phi(m)$ is equal to the number of factors in the mth cyclotomic polynomial $\Phi_m(x)$. The Möbius function $\mu(d)$ is an arithmetic function defined for positive integers d, which can also be used to define any cyclotomic polynomial. It takes the following values:

$$\mu(d) = \begin{cases} 1 & \text{if } d \text{ is a square-free positive integer with an even number of prime factors,} \\ -1 & \text{if } d \text{ is a square-free positive integer with an odd number of prime factors,} \\ 0 & \text{if } d \text{ is not square-free (i.e., divisible by a perfect square greater than 1).} \end{cases}$$

A *squarefree* number is one where its prime decomposition contains no repeated factors. The factorisation of the cyclotomic polynomial $\Phi_m(x)$ using the Möbius function is given by:

$$\Phi_m(x) = \prod_{d|m} (x^d - 1)^{\mu(m/d)}, \tag{3.8}$$

where d ranges over all positive divisors of m, and $\mu(m/d)$ is the Möbius function evaluated at m/d.

To construct a ring in $\mathbb{Q}(\zeta_m)$, we can quotient the polynomial ring $\mathbb{Q}[x]$ by the ideal generated by the cyclotomic polynomial $\Phi_m(x)$, denoted as $R = \mathbb{Q}[x] \,/\, (\Phi_m(x))$. In the Ring Learning With Errors (RLWE) problem, the secret $\mathbf{s}$ and the lattice points (a,b) are elements of a polynomial ring R_q, which is a reduction of R modulo q. R_q represents the ring $\mathbb{Z}[x] \,/\, (\Phi_m(x), q)$, where the coefficients of these polynomials are integers from the ring $\mathbb{Z}_q$, as opposed to the original Learning With Errors (LWE) problem, where the secret and error vectors are elements of the finite field $\mathbb{Z}_q^n$.
When $m = 2^n$ for some positive integer n, the cyclotomic polynomial $\Phi_m(x)$ takes the form:

$$\Phi_{2^n}(x) = x^{2^{n-1}} + 1$$

This form of polynomial is both compact and allows for efficient operations, which makes $m = 2^n$ a common choice when choosing parameters.

Theorem 1. *When $m = 2^n$ for some positive integer n, the cyclotomic polynomial $\Phi_m(x)$ is given by*

$$\Phi_{2^n}(x) = x^{2^{n-1}} + 1.$$

Proof.

The cyclotomic polynomial $\Phi_m(x)$ is defined as the minimal polynomial over $\mathbb{Q}$ that has as its roots the primitive m-th roots of unity, where $m = 2^n$.

The primitive m-th roots of unity can be expressed as $\zeta_k = e^{\frac{2\pi ik}{m}}$. When $m = 2^n$, the only divisors are powers of 2, hence ζ_k is primitive if k is odd. Thus, the set of primitive 2^n-th roots of unity is given by $\{\zeta_k = e^{2\pi ik/2^n} : k \text{ odd}\}$. The number of these roots is $\phi(m) = 2^{n-1}$, reflecting all odd integers up to $2^n - 1$.

We can then express $\Phi_{2^n}(x)$ as:

$$\Phi_{2^n}(x) = \prod_{\substack{k=1 \\ \gcd(k,2^n)=1}}^{2^n-1} (x - \zeta_k).$$

Noting that each ζ_k where k is odd has a complex conjugate ζ_{2^n-k}, since

$$\zeta_k \cdot \zeta_{2^n-k} = e^{2\pi ik/2^n} \cdot e^{-2\pi ik/2^n} = e^0 = 1,$$

the corresponding polynomial factor for the complex conjugate pair is

$$(x - \zeta_k)(x - \zeta_{2^n-k}) = x^2 - (\zeta_k + \zeta_{2^n-k})x + \zeta_k \cdot \zeta_{2^n-k}$$

and then by Euler's formula we can simplify to:

$$x^2 - 2x\cos\left(\frac{2\pi k}{2^n}\right) + 1.$$

Since cosine is an even function, and because of the symmetric distribution of roots on the unit circle, all terms with x^1 cancel out, leaving us with:

$$\Phi_{2^n}(x) = x^{2^{n-1}} + 1.$$

Q.E.D. □

Brakerski-Gentry-Vaikuntanathan (BGV) Scheme

The Brakerski-Gentry-Vaikuntanathan (BGV) scheme, as introduced by its authors [7], represents a significant advancement in the field of fully homomorphic encryption (FHE). It is based on the RLWE problem, and enhances performance and security, basing its security on less stringent assumptions compared to earlier schemes. This development is part of a broader effort to refine Gentry's foundational FHE scheme [5], alongside other notable variants like BFV [8] and CKKS [9], which are detailed further in subsequent sections.

Encoding Plaintext Data

Data is encoded into polynomials over $R_p = \mathbb{Z}_p[x] \,/\, (\Phi_m(x))$, where p is a prime number, referred to as the *plaintext modulus*, and $\Phi_m(x)$ is a cyclotomic polynomial, which defines the structure of the ring. The encoding process involves representing the plaintext message m as a polynomial in R_p. The simplest way to encode m is to set it as the constant term, which gives us

$$m(x) \;=\; (m, 0, \ldots, 0)$$

The above encoding method is known as constant term encoding. It is the simplest encoding scheme, but other and more efficient encoding schemes exist, such as k-ary encoding, where each bit in m is set as the coefficient in a polynomial, and then is decoded by evaluating $m(k)$. Each coefficient can hold a value in the range $[0, p)$.

Enryption The encryption procedure maps the elements of the plaintext ring R_p to ciphertext elements of the ciphertext ring R_q. The ciphertext modulus, q is a prime number, or a product of primes, typically chosen

to be very large. All coefficients in the ring R_q are taken modulo q. Given $m(x)$, the ciphertext is computed as a pair of polynomials $(c_0(x), c_1(x))$ in R_q :

$$c_0(x) = b(x) \cdot r(x) + p \cdot e_1(x) + m(x) \mod q$$

$$c_1(x) = a(x) \cdot r(x) + p \cdot e_2(x) \mod q$$

where $a(x), b(x)$ are the public key polynomials, $e_1(x)$, $e_2(x)$, and $r(x)$ are polynomials with coefficients drawn from the discrete Gaussian distribution, and are all in R_q.

Decryption
m is recovered by

$$m \ = \ (\ c_0(x) \ + c_1(x) \cdot s(x) \mod q \ ) \mod p$$

This works as long as the noise is within the noise bound. Noise estimation and tracking is done with heuristics and implementation tools, such as a noise budget which decrements by a set amount based on the specific operation.

Key Generation
The BGV scheme uses a set of 3 keys, the secret key sk, the public key pk, and the evaluation key evk. The public key consists of a set of elements in the polynomial ring R_q, where the coefficients of $\Phi_m(x)$ are in the ring of integers modulo q. The public key is generated as

$$pk = (a(x), b(x))$$

where $a(x)$ is sampled from R_q, $b(x) = a(x) \cdot s(x) + p \cdot e(x) \mod q$, and $e(x)$ is a small error polynomial, as described in the LWE problem, but this time a and b are univariate polynomials rather than vectors. The secret key $sk = s(x) \ \in R_q$ has coefficients drawn from the error distribution, with the exception of the first, as $s(x)$ must be a monic polynomial. For computational efficiency, the secret key is decomposed into a base-G representation of its coefficients. A common choice for G is 2, as it allows for fast bit-wise operations.

Relinearisation
The evaluation keys are used for performing homomorphic operations. In the BGV scheme, we will need to generate a key-switching matrix for relinearisation. The primary goal of relinearisation is to reduce the degree ciphertexts that result from homomorphic multiplication operations. Multiplying two ciphertexts not only increases the noise but also the degree of the resulting ciphertext, which needs to be brought back to a polynomial within R_q. To generate the keys, we first need to define L, the maximum muliplicative depth required. We then need to find $s(x)^i$, for $i \in \{2, \ldots, L+1\}$, and decompose the coefficients into base-2 representation. We then need to generate $(a_{i,j}, b_{i,j})$ as

$$a_{i,j} \ \in \ R_q$$

$$b_{i,j} = a_{i,j} \ \cdot \ s(x) \ + \ e_{i,j}$$

where j indexes the base-2 coefficients of the polynomials, and $e_{i,j}$ is a small error term. For relinearisation keys, $a_{i,j}$ is not randomly sampled from R_q as with pk, and is instead chosen so that it transforms $s(x)^i$ back to a polynomial in R_q. Relinearisation is effectively a form of key-switching, because when a ciphertext is multiplied, the secret key is squared, so the task now becomes to switch the secret key of the ciphertext from s^2 to s.

Automorphisms

An automorphism in a polynomial field is a bijective mapping from the field to itself that preserves the field operations. More formally, let F be a field and define σ as a mapping where $\sigma : F \rightarrow F$. Then, σ is an automorphism if σ is bijective over all elements in F, and both multiplications and addition operations preserve the field:

$$\sigma(a+b) = \sigma(a) + \sigma(b) \ \forall \ a, b \in F$$

$$\sigma(a \cdot b) = \sigma(a) \cdot \sigma(b) \ \forall \ a, b \in F$$

Let $f(x) \in F[x]$ be a polynomial with coefficients in a field F, and let E be the splitting field of $f(x)$ over F. In our case, E is just the extension field associated with the mth cyclotomic polynomial. The Galois group of $f(x)$ over F, Gal(E/F), is the group of all automorphisms of E that fix F pointwise, meaning that the elements in F will map back onto themselves under one such automorphism of E, $Aut(E)$:

$$\text{Gal}(E/F) = \sigma \in \text{Aut}(E) : \sigma(a) = a, \ \forall \ a \in F$$

A Frobenius automorphism is an element in the Galois group, which is bijective over both the base field $GF(p)$ and the extension field $GF(p^n)$. This means that it fixes the elements in the base field point-wise and therefore acts as the identity map, since for any element a in $GF(p)$, $a^p = a$ due to Fermat's Little Theorem. This is highly useful, because it allows us to use slot-wise (see here) linear mappings of plaintext polynomial elements. These linear transformations can be formulated as

$$\sum_{i=0}^{d-1} a_i \sigma_E^i(v)$$

where d is the order of the polynomial, σ_E is a Frobenius map E, v is an element in E. The a_i are constants in E, which can be solved for [10], however this is beyond the scope of this paper.

Bootstrapping

Bootstrapping is a way to reduce the noise that has built up in a ciphertext after one or more homomorphic operations. This can be done a number of ways, such as *mod-down*, where the modulus q is reduced, or recryption, where the encrypted values are extracted and placed into a new ciphertext polynomial with the noise level reset, without ever revealing the real values. This is done by encrypting the components of the $s(x)$ with pk, and applying the decryption circuit homomorphically. The 'decrypted' values are then re-encrypted, resulting in a new polynomial.

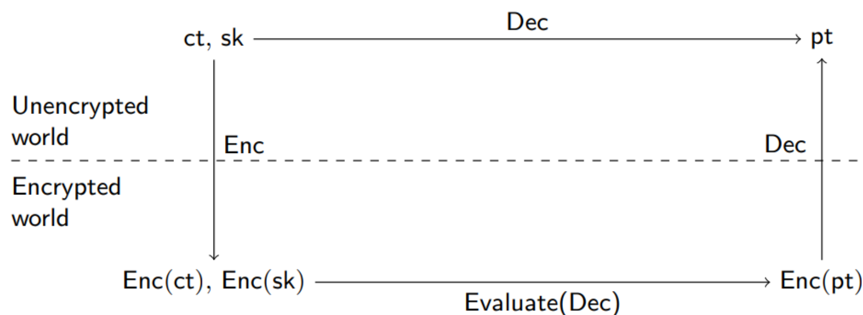

Figure 2 – Bootstrapping diagram, adapted from [11]

Ciphertext Packing

When the plaintext modulus p splits completely in $\Phi_m(x)$, we can encode multiple values into a single ciphertext polynomial. This means that the prime number p can be factored in the polynomial field into N distinct linear factors $(x - a_n)\ mod\ p$. Because these roots are pairwise co-prime, the Chinese Remainder Theorem (CRT) can be applied to pack N values into one.

Lemma 1 (Chinese Remainder Theorem). *Let $n_1, n_2, \ldots, n_k$ be pairwise coprime positive integers, and let $N = n_1 n_2 \cdots n_k$. For any given integers $a_1, a_2, \ldots, a_k$, there exists an integer x such that*

$$\begin{aligned} x &\equiv a_1 \mod n_1, \\ x &\equiv a_2 \mod n_2, \\ &\vdots \\ x &\equiv a_k \mod n_k. \end{aligned}$$

Moreover, any two solutions x, y of this system of congruences satisfy $x \equiv y \mod N$.

Hensel's Lifting Hensel's lifting is an algorithmic technique used to find solutions to polynomials modulo powers of a prime. Starting with a solution modulo a prime p, Hensel's lemma lets us to "lift" this solution to find solutions modulo higher powers of p. With respect to the BGV scheme and its applicability, Hensel's lifting effectively increases the plaintext modulus from p to p^r [12], which greatly expands the utility of the scheme. It achieves this by using a technique similar to the Newton-Raphson method.

Lemma 2 (Hensel's Lemma). *Let $f(x)$ be a polynomial with integer coefficients, and let p be a prime. Suppose that there exists an integer a such that:*

$$f(a) \equiv 0 \pmod{p}, \text{ and}$$

$$f'(a) \not\equiv 0 \pmod{p},$$

where $f'(x)$ denotes the derivative of $f(x)$. Then, for any positive integer k, there exists a unique integer a_k modulo p^k such that:

$$f(a_k) \equiv 0 \pmod{p^k}, \text{ and}$$

$$a_k \equiv a \pmod{p}.$$

Moreover, the values of a_k can be computed iteratively using the following formula:

$$a_{k+1} = a_k - \frac{f(a_k)}{f'(a_k)} \pmod{p^{k+1}}.$$

Ciphertext packing with slots also enables SIMD operations, where a single homomorphic operation can be applied to multiple data elements simultaneously, improving the efficiency and throughput of homomorphic computations in the BGV scheme. The number of slots available in a ciphertext is given by

$$N = \frac{\phi(m)}{d} \tag{3.9}$$

where $\phi(m)$ is Euler's totient function, and d is the multiplicative order of p in m [12].

3.4 SECURE ENCLAVES

Secure enclaves are isolated execution environments within a processor that provide a higher level of security and privacy for sensitive code and data. Secure enclaves are used for protecting sensitive data such as cryptographic keys or personal information and secure remote computation. They are designed to protect the confidentiality of the code and data inside them, even from the kernel. Secure enclaves also support a process called attestation, which allows a remote party to verify the identity and integrity of the enclave and the code running inside it by way of digital signatures, hashing and logs. Examples of secure enclave technologies include, Intel Software Guard Extensions (SGX), ARM TrustZone, and AMD Secure Encrypted Virtualization (SEV). TrustZone is a physical hardware implementation for secure enclaves, SEV is a hardware-assisted method for encrypting virtualisation, and SGX is a set of CPU instructions that allows programs to invoke secure enclaves on Intel processors. In this work, I used Intel's SGX solution, primarily due to availability and accessibility. In this paper, when I refer to secure enclaves, or enclaves for short, assume that I am referring to Intel SGX secure enclaves specifically.

Due to the security requirements of secure enclaves, there are several restrictions on how they can be used. Only C and C++ programs can run inside them. To transfer data between the enclave and the host it must be passed to a buffer and sent back and forth, threading is restricted, and the standard C and C++ libraries cannot be used, and instead a modified version provided by SGX must be used.

The Enclave Definition Language (EDL) file defines the interface between the trusted enclave code and the untrusted application code. This file defines the functions that the untrusted application can call within the enclave (ECalls) and the functions the enclave can call outside the enclave (OCALLs). These functions enable communication between the two otherwise independent programs *app* and *enclave*.

3.5 USE CASES

Beyond its applicability in secure payment systems, the combination of fully homomorphic encryption and secure enclaves extends to a broad spectrum of applications across various sectors. In essence, any system that requires processing data and where confidentiality is important can benefit from such technology integration. In law enforcement and security, for example, this technology enables the confidential cross-checking of identification documents against sensitive databases, such as blacklists or watch lists maintained by organisations such as INTERPOL, without revealing the contents of either. In the domain of computational biology, the technology could be used for privacy-preserving machine learning on sensitive data [13], such as sequenced genomes, allowing for the discovery of new compounds with potential therapeutic applications, such as the discoveries made in protein folding [14]. Additionally, in compliance with KYC laws, financial institutions could process and verify customer data on untrusted platforms, such as public clouds infrastructure, without exposing sensitive information. More generally, this approach allows for the secure extraction of the statistical properties from private datasets without giving away the individual data points. This set of diverse applications underscore the transformative potential of FHE and secure enclaves to extend the amount of usable data that can be used in discovery and decision making.

4 IMPLEMENTATION

4.1 SCHEME SELECTION

Gentry's first FHE scheme is considered the first generation of FHE, and is not generally used in practice. The second generation of FHE schemes include the Brakerski-Gentry-Vaikuntanathan (BGV) [7] and Brakerski/Fan-Vercauteren (BFV) [8] schemes, which build upon the initial scheme, but the improvements allow for a broader range of use cases and have a large number implementations. Cheon-Kim-Kim-Song (CKKS) [9] is considered a third generation FHE scheme, and works differently to the first two. CKKS is designed for efficient encrypted arithmetic on approximate numbers, and is more robust to noise and significantly less computationally expensive. For this application, which involves verifying exact values in transaction data, CKKS is not suitable due to its approximate nature. For this work, I have selected the BGV scheme, as it provides exact integer arithmetic, and has been extensively studied and optimised [11,15–17] .

4.2 TOOLS & LIBRARIES

A wide array of libraries exist for fully homomorphic encryption, for different languages, hardware platforms, applications and compute paradigms [18], and is an area of active and continuous development. I selected HElib as the FHE library to work with, as it is written in C++, which is both fast and object-oriented. It is built on top of NTL, a library for number theoretic computation.

Given that the files needing to be transferred between nodes in the network are large and have structured data of variable sizes, I opted to use Protocol Buffers with gRPC. The `.proto` file [19] defines the structure of the data to be transferred and the data types of each field, which correspond to each field in the NACHA file.

Protocol Buffers with gRPC were chosen for data serialisation and transfer due to their compatibility with Intel SGX secure enclaves and their performance advantages. Protocol Buffers provide a compact and efficient binary format for structured data, reducing the overhead of data transfer and storage. gRPC, built on top of Protocol Buffers, offers a high-performance, scalable, and secure framework for remote procedure calls (RPCs). The use of these technologies enables efficient communication between the enclave program and other components of the system, while ensuring data integrity and confidentiality. Moreover, the limited set of libraries available for use within SGX enclaves makes Protocol Buffers a suitable choice, as it is one of the supported libraries. This compatibility lays the groundwork for further vertical integration and interoperability between the enclave program and the rest of the system, potentially simplifying development and maintenance efforts.

The system also required a number of language-agnostic programs, for example the client which uploads data to be stored in the enclave, which were written in scripting languages including Python and bash.

Message Passing Interface (MPI) is a message-passing system for parallel computing architectures. MPI facilitates communication among processes, for both point-to-point and broadcast communication. There are several implementations of the standard, and for this work I chose OpenMPI for its ease-of-use and C++ compatibility. In MPI, the same program is run on all nodes, and each nodes is given a unique identifier. The program is written in such a way to divide the tasks evenly among the number of processes. Each node can provide up to *nproc* processes, where *nproc* is the number of processing cores on the machine.

4.3 Parameter Selection

HElib provides several additional programs which help with parameter selection and creating contexts from which keys are generated according to the requirements. The `params` program generates a set parameters for the BGV scheme given a plaintext modulus p, with the option to specify others[20]: Given the plaintext modulus p and a range of values for m (the cyclotomic polynomial order), it iteratively tries to find an m in which p splits completely, and ensures that p and m are coprime and the multiplicative order of p modulo m is not too large for efficiency purposes. It then searches for a generator of the multiplicative group $\mathbb{Z}_m^*$ that yields an acceptable depth-cost tradeoff for the Hensel lifting process in bootstrapping. The parameters output are $(m, p, mvec, gens, ords)$, where $mvec$ is the vector of prime power factors of m, $gens$ is the vector of generators for each prime power factor, and $ords$ is the vector of multiplicative orders of the generators, which are needed in the creation of bootstrapping keys.

The `create-context` program attempts to create a BGV context using the parameters provided from the `params`, plus values for r, c and $Qbits$, the Hensel lifting parameter, the number of columns in the key-switching matrices and the number of bits is the modulus chain (the length of the base-2 representation of q). It constructs a BGV context and generates a secret key, along with the necessary key-switching matrices and Frobenius matrices for bootstrapping. The context and keys are written to output files. The `test-bootstrap` program tests the generated context to ensure it supports bootstrapping. Algorithm 1 searches for a suitable bootstrappable BGV context by iterating over different parameter sets generated by the `params` program, creates a context for each parameter set using `create-context` and tests it using `test-bootstrap`. If a bootstrappable context is found, the algorithm returns the context and its parameters. If no suitable context is found after exhausting all parameter sets, the algorithm returns a failure indication.

Algorithm 1 Finding Bootstrappable BGV Context

Require: Plaintext modulus p, optional parameters r and m
1: **if** r is not provided **then**
2:     $r \leftarrow 1$
3: **end if**
4: **if** m is not provided **then**
5:     Generate parameters using `params` program with p
6: **else**
7:     Generate parameters using `params` program with p and m
8: **end if**
9: **for** each set of parameters $(m, p, mvec, gens, ords)$ **do**
10:     Create a temporary parameter file with the current parameters
11:     Attempt to create a BGV context using `create-context` program
12:     **if** context creation is successful **then**
13:         Test the generated context for bootstrappability using `test-bootstrap`
14:         **if** context is bootstrappable **then**
15:             **return** the bootstrappable context and its parameters
16:         **end if**
17:     **end if**
18: **end for**
19: **return** failure (no suitable bootstrappable context found)

Effect of m on Performance

The graph in figure 1 contrasts the computational efficiency between BGV contexts where m is a power of 2 and where it is not. As we can see, the small gains in efficiency compound as m increases. Several of the most common operations, such as modular reductions, integer multiplication, and polynomial arithmetic operations benefit the most from working with powers of 2.

When m is a power of 2, modular reduction can be performed using a bitwise AND operation with $2^n - 1$. Multiplications by 2^n can be efficiently implemented as left and right bitwise shifts, respectively. Polynomial multiplication is a non-linear operation and is computationally expensive. The Number Theoretic Transform (NTT) is an algorithm similar to the Fast Fourier Transform (FFT), but adapted to work over finite fields. The NTT allows point-wise multiplication and addition, which significantly reduces compute time[21]. The NTT works as follows: Given a vector of coefficients $a = [a_0, a_1, \ldots, a_{n-1}]$ representing a polynomial $A(x) = a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, the (forward) NTT is defined as:

$$\hat{A}[k] = \sum_{j=0}^{n-1} a_j \, \omega^{jk} \mod p \text{ , for } k = 0, 1, \ldots, n\text{-}1$$

where ω is a primitive n^{th} root of unity modulo p (ω has multiplicative order m in p). The inverse-NTT transforms the sequence back to polynomial coefficients and is given by:

$$a_j = \frac{1}{n} \sum_{k=0}^{n-1} \hat{A}[k] \, \omega^{-jk} \mod p \text{ for} j = 0, 1, \ldots, n-1$$

where ω^{-1} is the modular multiplicative inverse of ω modulo p, and $\frac{1}{n}$ denotes the modular multiplicative inverse of n modulo p.

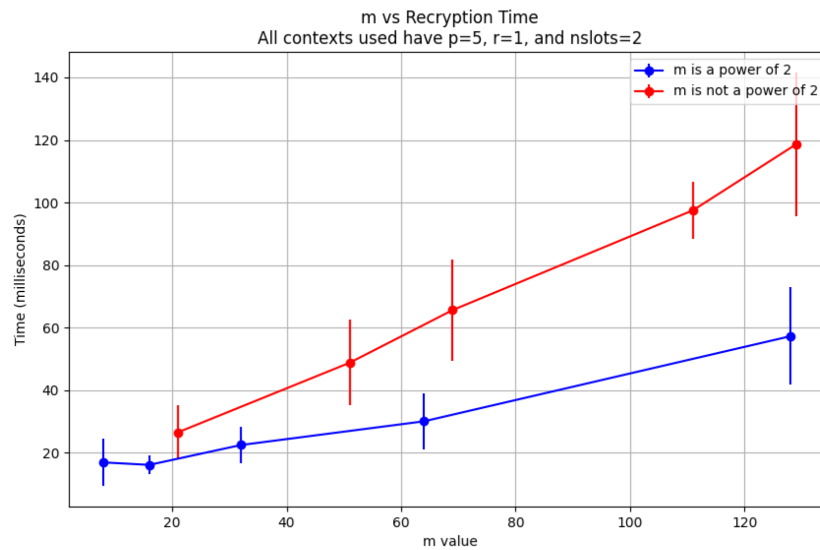

Figure 1 – Computation time when $m \in 2^n$ vs. $m \notin 2^n$

Table 1 – Parameters for Numerical Validation

hline Parameter	Value
m	32
p	5
$\phi(m)$	16
ord(p)	8
normBnd	1
polyNormBnd	1
factors	[2]
generator	15
order of generator	2
T	[1,15]
r	10
p^r	$5^{10} = 9,765,625$
nslots	2
hwt	2
ctxtPrimes	$[6,7,8,\ldots,39]$
specialPrimes	$[40,41,\ldots,57]$
number of bits	3097
security level	0

the parameters used in this work were selected from a subset identified by running Algorithm 1 and selecting for the most suitable parameters, based on the number of slots, plaintext space and the associated cyclotomic polynomial $\Phi_m(x)$.

The chosen parameters were optimised for speed, and resulted in a significantly reduced computation time of approximately 210ms for a test program involving a square operation and bootstrapping. This contrasts sharply with a larger parameter set from the paper [12], with ($m = 21845$, $p = 2$, $\phi(m) = 16384$), which took approximately 10400ms for similar operations. The smaller parameter values lead to a more manageable lattice dimension $\phi(m)$, giving faster computations - an essential feature for processing large volumes of transactions in real time. Additionally, the size of the public key for these parameters require approximately 740MB of storage, which is incompatible with storage space on a single enclave. However, this efficiency comes at the cost of security, as indicated by a security level (λ) of zero. In the BGV scheme, λ is measured as the bit strength of the encryption, which determines the difficulty level of finding the secret key with brute force. A λ of zero implies that the system does **not** provide cryptographic security under this parameter set. Given that the underlying system is congruent to a system with a non-zero security level, and given that this paper aims to describe a proof of concept rather than simulation of a production-ready system, I opted for these parameters such that I could run the system in an end-to-end manner on the hardware that was available to me.

Table 1 shows parameters that were used for verification of batch and file totals. Here, only one slot was needed and computation speed is important. The plaintext space p^r is a key parameter, as it limits the total value of transactions that can be placed in a NACHA file. This stems from the fact that all debit and credit transactions are totaled and placed in the file control, and must be validated by recomputing the totals and comparing them. Another possible approach would be to use an arbitrary plaintext space, and assign a probability to the result being a false positive, similar to a checksum. Given that there is only one correct

answer for

$$\sum_{i=1}^{n} \text{debitTransaction}_i = \textit{totalDebitEntryDollarAmountInFile}$$

and the range of values a slot can take is $[0, p)$, the probability of the result being a false positive is

$$P(error) \approx \frac{1}{p^r}$$

where p^r is large.

For table lookups, where data in the NACHA file is being compared against an encrypted LUT, all of the characters must be encoded into the ciphertext. The number of slots must be at least the maximum length of any string field, and the minimum plaintext space must be 256, to hold all ASCII characters. With these constraints, the following parameters were chosen:

Table 2 – Parameters for String Comparisons

Parameter	Value
m	451
p	2
phi(m)	400
ord(p)	20
normBnd	1.60932
polyNormBnd	3.15128
factors	[11 41]
generator orders	10, 2
T	[1, 331, 288, . . . , 413]
r	8
nslots	20
hwt	120
ctxtPrimes	[6, 7, 8, . . . , 39]
specialPrimes	[40, 41, 42, . . . , 52]
number of bits	2776
security level	19.3466
public key size	15412728

4.4 Processing & Validation

Networking

The ODFI initiates the processing pipeline by requesting the public key from the master node. It then uses this to encrypt the NACHA file contents. Only the values are encrypted, while the field names (e.g. *recordType*) are left in plaintext. It is then sent to the master node, where it is passed to the validation program. The validation program is computed across a network of worker nodes, using MPI. Each node requires the master nodes public SSH keys for initial setup. The nodes are also given access to files within an NFS system. This includes a file of Routing Transit Numbers (RTN) encrypted with the public key. The reason this is included as a separate file to the validation program, is for the purpose of accommodating updates to the list of RTNs, as new institutions may join the ACH network over time. This step also allows other clearing processes to take place, such as cross-checking data in the NACHA file with other datasets, for anti-fraud or other law enforcement purposes. The results of the validation program are computed and then gathered at one node, before being sent back to the host node, or if the host node has an allocation of processes for computation

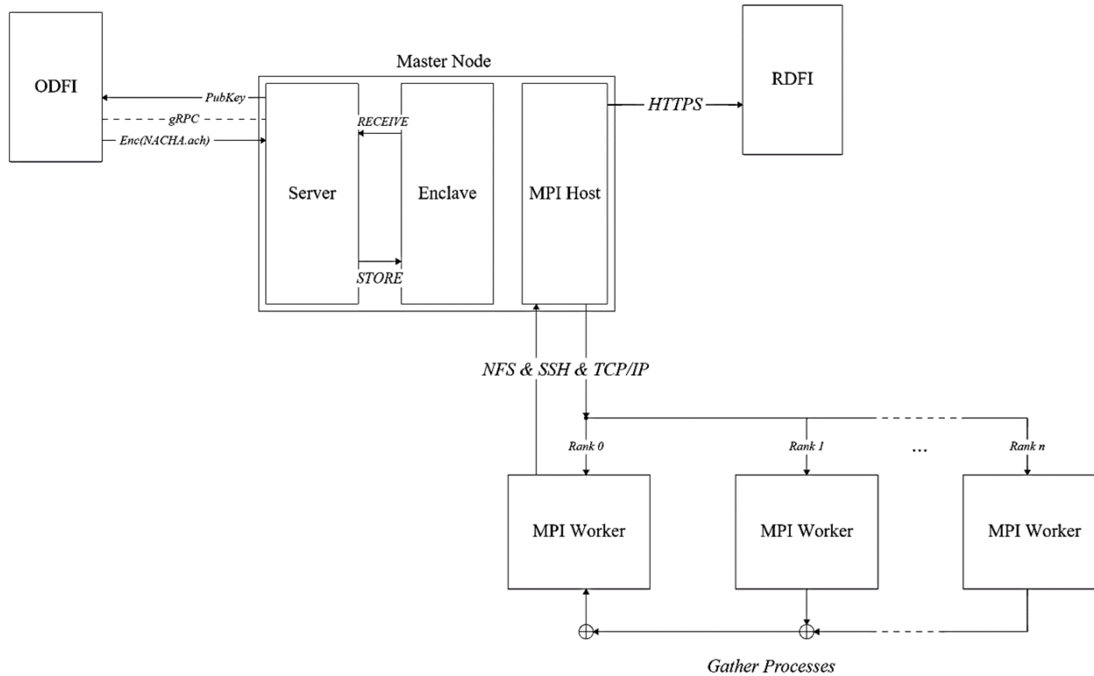

Figure 2 – System Diagram

locally, the results are gathered directly at the host, and stored in a file. The last step is decryption of the results. Each validation task in the program outputs either $E(1)$ or $E(0)$, so this step simply involves getting the secret key from the enclave (by recombining the file parts), decrypting the results and verifying that all results are equal to 1. If all results are 1, the file can be sent on to the RDFI to proceed with the settlement.

HOMOMORPHIC COMPARISON

Listing 4.1 – Function to compare two encrypted values in HElib.

```
helib::Ctxt compare(helib::Ctxt val1, helib::Ctxt val2, const helib::PubKey &publicKey
    ) {
    const helib::EncryptedArray ea = publicKey.getContext().getEA() ;
    helib::Ctxt mask_entry = val1;
    mask_entry -= val2;
    power_recrypt(mask_entry, publicKey.getContext().getP() - 1);
    if (mask_entry.bitCapacity() < NOISE_BOUND) publicKey.thinReCrypt(mask_entry);
    mask_entry.negate();
    if (mask_entry.bitCapacity() < NOISE_BOUND) publicKey.thinReCrypt(mask_entry);
    mask_entry.addConstant(NTL::ZZX(1));
    std::vector<helib::Ctxt> rotated_masks(ea.size(), mask_entry);
    for (int i = 1; i < rotated_masks.size(); i++)
      ea.rotate(rotated_masks[i], i);
    totalProduct(mask_entry, rotated_masks);
    if (mask_entry.bitCapacity() < NOISE_BOUND) publicKey.thinReCrypt(mask_entry);
    return mask_entry;
}
```

Figure 4.1 shows the function used for homomorphically comparing two ciphertexts for a full match, meaning each slot in $val1$ must be equal to the corresponding slot in $val2$. The function computes `mask_entry` as the difference between the two ciphertexts, which should be $E(0)$ if they are equal, where E is the encryption function. The difference is then raised to the power of $p-1$,

Lemma 3 (Fermat's Little Theorem)**.** *Let p be a prime number. For any integer a that is not divisible by p, it holds that*

$$a^{p-1} \equiv 1 \mod p.$$

It follows that

$$a^p \equiv a \mod p.$$

If `mask_entry` is non-zero , this step transforms it into $E(1)$, otherwise it remains $E(0)$. By negating `mask_entry` and adding $E(1)$, we flip the result, so that if the two ciphertexts are equal, the result is $E(1)$, otherwise it is $E(0)$. Note that this applies to all slots, so to ensure we have a full match, we take the product of all slots which will be

$$\prod_{i=1}^{n} slot_i \in \{E(0), E(1)\}$$

.

The `power_recrypt` function is a modified version of HElib's power function, were the noise budget is checked and the ciphertext is bootstrapped if it falls below the minimum level.

Key Generation And Storage

The setup begins with key generation. The public-private key pairs for the validation program are generated at the master node from the chosen parameters. The public key is stored in the unprotected file system for ease of access, while the secret key is stored in the secure enclave. The host application has a TCP server listening for connections, and can accept requests to store or retrieve files stored inside the enclave's protected file system. The protected file system is ephemeral, given that enclaves themselves are ephemeral, hence if the enclave program crashes or is forced to stop, the private key will be lost and must be regenerated. Only the programs running locally on the master node can retrieve the secret key.

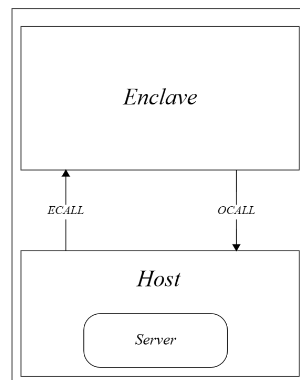

Figure 3 – Basic structure of secure enclave and host server application

Due to limits on the maximum buffer size, and for added security, the private key file is split into smaller parts using the `split` utility in Linux. Each part has a common prefix and a unique string of letters appended.

To further enhance the security, one could rename these parts such that the index is not easily discernible, and the correct ordering of these parts is kept secret elsewhere. In Listings 4.2 and 4.3, the outputs show a request to store a part of the secret key (`sec_keyaa`) in the enclave. The SGX debugger `sgx-gdb` is used to step through the running program and hits a breakpoint inside the enclave code.

Listing 4.2 – Client command to store a file in the enclave

```
$ python3 client.py STORE sec_keyaa
```

Listing 4.3 – GDB output while debugging the enclave

```
Breakpoint 1, 0x00007ffff5fbc050 in sgx_fopen_auto_key ()
(gdb) r
status: 0
Server is running and waiting for connections...
STORE
filename: sec_keyaa
data_size: 32768
Breakpoint 1, 0x00007ffff5fbc050 in sgx_fopen_auto_key ()
(gdb) s
Single stepping until exit from function sgx_fopen_auto_key,
which has no line number information.
0x00007ffff5fbbf10 in sgx_fopen_internal(char const*, char const*, unsigned char const
      () [16], unsigned char const () [16], unsigned long) ()
(gdb) s
Single stepping until exit from function _ZL18sgx_fopen_internalPKcS0_PA16_KhS3_m,
which has no line number information.
ecall_store_data (data=0x7ffff60d8120 "|HE[", data_size=32768, filename=0x7ffff60d8010
      "sec_keyaa") at Enclave/Enclave.cpp:407
```

Batch Total Circuit

Below is a simplified representation of the homomorphic circuit used to calculate the batch totals in a NACHA file. In the code implementation, the binary elements 1 and 0 are encrypted transaction codes, which must be compared. The output of the comparison circuit is still $E(0)$ or $E(1)$, so it can be used to sum the amounts.

$$\mathbf{v}_{\text{debit}} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{v}_{\text{credit}} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{T} = \begin{bmatrix} 1 & Amount_1 \\ 0 & Amount_2 \\ 1 & Amount_3 \\ 0 & Amount_4 \\ \vdots & \vdots \end{bmatrix}$$

$$\text{Total}_{\text{credit}} = \mathbf{T}\,\mathbf{v}_{\text{credit}}$$

$$\text{Total}_{\text{debit}} = \mathbf{T}\,\mathbf{v}_{\text{debit}}$$

The same logic can be used for encrypted databases and look-up tables. This application of homomorphic encryption is an area of ongoing research [22,23] due to its large number of potential use cases. The use of slots and SIMD may be leveraged to improve efficiency, however summing across individual encrypted arrays (polynomials) is not possible, and introduces complexity. However, there does exist a number of ways to perform linear algebra computations on ciphertexts. The paper [24] describes a set of methods for performing addition,

linear transformation, and weighted inner products on integer vectors, which is particularly useful for signal processing applications.

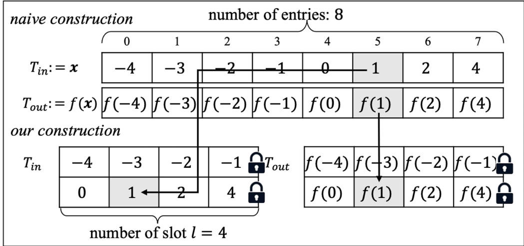

Figure 4 – An example of constructing LUTs for single-input function - adapted from [23]

5 Discussion

5.1 Computation Time

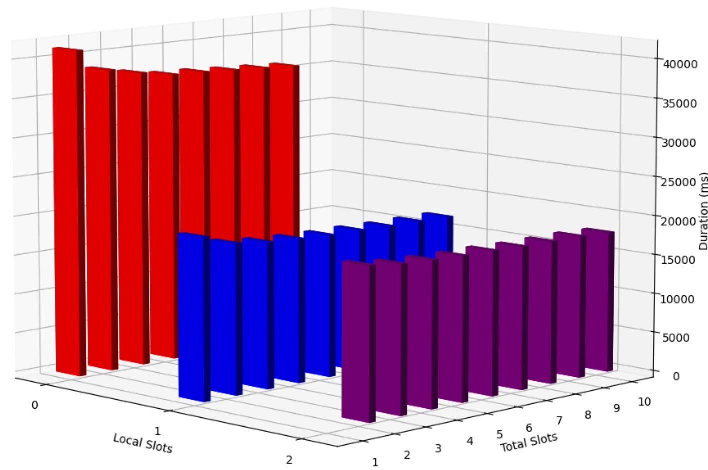

Figure 1 – Computation Times vs *nproc*

Impact of Local Process Allocation on Compute Time

The simulations conducted revealed significant insights into the impact of process distribution between the host and remote nodes on overall system performance. The data, shown in the bar plot above with axes representing total processes (nproc), local processes, and duration in milliseconds, highlights the performance improvement from running at least one process locally on the host machine. Despite the relatively low network latency between the host and worker nodes of 2.262 ± 1.14 ms, the absence of local processing resulted in a substantial increase in compute time, with a delta of up to 20 ms.

This behavior can be attributed to several factors, including the overhead associated with managing and gathering the remote processes exclusively, as well as delays in data transmission and synchronisation across the network. When processes are distributed such that at least one runs locally, the cost of overheads is substantially reduced, leading to improved computational efficiency. While adding more processes generally decreased the compute time, the reduction followed a diminishing returns pattern; each additional process contributed less to the decrease in compute time, highlighting the non-linear scalability of parallel processing in this context. This behavior is indicative of the complexities encountered in optimising distributed crypto-

graphic computations, where inter-process communication and resource contention require careful management.

Suboptimal Resource Allocation and Resolution

During the implementation phase, an unexpected increase in computation time was observed when the number of processes was increased from 4 to 5, resulting in an additional 7 seconds of compute time, representing a significant increase of approximately 36.84%. This anomaly was traced back to the hardware configuration of the machine, which comprised 2 sockets with 4 cores each. The fifth process was being allocated to a core which was already in use, leading to increased context switching overhead and memory consumption.

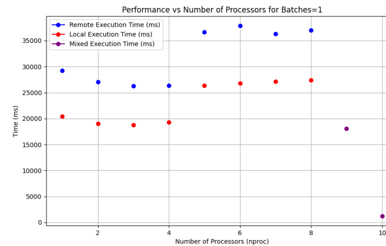

Figure 2 – Compute time of the validation program with oversubscribed cores starting at nproc=5

To address this issue, the node's architecture was changed to a single-socket configuration with an equivalent number of cores. This adjustment aligned the number of processes with the number of available cores, facilitating a more uniform distribution of the computational load across the hardware. Post-migration measurements confirmed a near-linear decrease in computation time with each additional process, underscoring the importance of aligning the hardware architecture with the computational demands of resource-intensive cryptographic computations.

Regulatory and Compliance Implications

The integration of homomorphic encryption methods and ACH payment systems presents several implications for regulatory compliance and data privacy, particularly with respect to the Gramm-Leach-Bliley Act (GLBA) in the United States. The GLBA requires financial institutions to protect the security and confidentiality of consumer data [25], a mandate which FHE and secure enclaves could help facilitate. The accuracy and integrity of transaction data maintained by FHE support the standards set in the Electronic Fund Transfer Act (EFTA), which aims to protect consumers by ensuring the correctness of electronic fund transfer services and facilitating efficient dispute resolution [26]. Specifically, a system leveraging secure enclaves and FHE can be of use in the handling of error resolution and unauthorised transfer claims, as specified under 12 CFR 1005.11[26], ensuring that all data processing and error logging remain confidential and tamper-proof.

FHE ensures that data remains encrypted, thus adhering to the requirements outlined in GLBA. The capabilities of secure enclaves also align well with these goals by providing a protected execution environment that isolates sensitive data and operations from the rest of the system. However, additional legislature may need to be introduced to define how the secure enclave hardware is managed and where it is being physically held. This approach not only helps in complying with regulations but also provides a framework for adhering to similar requirements under the General Data Protection Regulation (GDPR) for EU-based payment systems.

Scalability Considerations

The proposed architecture is engineered to function in a trustless environment, where the party that encrypts the data and the evaluator of the data do not require mutual trust or direct interaction. This framework enabled scalability by allowing an arbitrarily large number of entities to join as nodes and contribute to transaction processing, democratising participation and distributing the computational load across a potentially global network. The trustless nature of the system minimises reliance on any single point of failure, adding to the resilience of the ACH payment processing infrastructure. With respect to the scalability as relates to the management of cryptographic keys, which require substantial storage when using parameter sets with non-zero security parameter, this issue can be mitigated by splitting the key into an arbitrary number of parts across multiple enclaves. This approach to key management enhances security by decentralising the storage of secret keys, reducing the risk associated with single points of compromise.

6 Conclusion

The integration of Fully Homomorphic Encryption (FHE) using the Brakerski-Gentry-Vaikuntanathan (BGV) scheme and secure enclaves into Automated Clearing House (ACH) payment systems showcases a new paradigm in the security and privacy of financial transactions. This paper has explored the implementation of these tools to enhance the robustness of the ACH system against cyber threats, while maintaining the confidentiality and integrity of transaction data.

The use of the BGV scheme allows for computations on encrypted data without the need for decryption, thereby preserving data confidentiality even on open or untrusted platforms. Secure enclaves provide an isolated environment to protect sensitive data and control access to private keys. This also introduces a level of transparency and zero-trust akin to that seen in decentralized financial systems.

Throughout this study, various computational and theoretical analyses were conducted to evaluate the performance and security of the proposed system. The findings indicate a material improvement in the security of the ACH system, mitigating risks associated with data breaches and unauthorized access with the addition of validation layers.

Future work in this area could explore the scalability of the system, particularly in handling a larger volume of transactions and extending the framework to support global financial networks. To make such a system feasible in production, further improvements would be needed in terms of computational efficiency, given that the parameters chosen for the scheme would skew towards favoring security over efficiency. This could involve drawing upon several technologies and design improvements, ranging from the vectorisation of homomorphic operations over encrypted arrays, hardware upgrades and distributed computing.

In conclusion, the integration of FHE and secure enclaves into ACH payment systems offers a promising approach to addressing the evolving security challenges in the era of digitisation. By leveraging cryptographic tools, financial institutions can ensure the security and privacy of transactions, thereby maintaining trust and reliability in the financial ecosystem.

7 Further Work

MPI Parallelisation

The implementation of parallel processing in the validation program was done by simply distributing tasks among processes based on their rank. This approach, while effective, leaves room for optimisation, particularly at the level of bootstrapping. HElib's in-built multi-threading approach for bootstrapping reduces compute time by 90% [12] which significantly impacts the overall compute time for circuits with multiple levels. Despite this, boostrapping is by far the most expensive operation, and bootstrapping triggers should be done the minimum number of times required.

FHE in secure enclaves

While this paper has explored the complementary use of FHE and secure enclaves in enhancing the security of ACH payment systems, there exists potential for deeper integration of these frameworks.

There have been attempts at integrating the two in existing literature that explore the potential benefits of such integration. Existing literature, including the study describing the GPS framework [27], illustrates attempts to combine the scalability of homomorphic encryption with the computational capabilities of Intel SGX. This integration has demonstrated significant performance improvements in outsourced secure computations, highlighting the benefits of merging these advanced technologies.

In this implementation, while the secret key is stored within the enclave, limitations related to SGX's threading capabilities and the restricted use of the C and C++ standard libraries have prevented the full integration of the HElib and NTL libraries within the enclave environment. This has notably impacted the feasibility of generating the secret key directly within the enclave. However, the generation of RSA keys within the enclave is feasible with an SGX-compatible version of the MbedTLS library, and can be used for secure communication and signing SSL/TLS certificates.

Future work could explore the development of an FHE library specifically designed to leverage secure enclaves, particularly for operations such as sampling from the error distribution to generate keys. Such an approach would ensure that sensitive operations and data, namely the polynomial coefficients for the secret key, remain confined within the secure enclave, only accessible to the host upon explicit request. This would both enhance the security of the cryptographic operations but also expand the practical applications of FHE in secure computing environments.

8 APPENDIX A

8.1 1.1

Listing 8.1 – Function to validate file totals using HElib.

```
std::vector<helib::Ctxt> FileTotals(const helib::PubKey &publicKey, const std::vector<
     std::vector<helib::Ctxt>> &batchTotals){
    helib::Ctxt sumDebit = FHEencrypt(0L, publicKey) ;
    helib::Ctxt sumCredits = FHEencrypt(0L, publicKey) ;
    for (auto& totals : batchTotals) {
        sumDebit += totals.at(0) ;
        if (sumDebit.bitCapacity() < 100) publicKey.thinReCrypt(sumDebit) ;
        sumCredits += totals.at(1) ;
        if (sumCredits.bitCapacity() < 100) publicKey.thinReCrypt(sumCredits) ;
    }

    return {sumDebit, sumCredits} ;
}
```

9 APPENDIX B

9.1 1.1

Listing 9.1 – Function to store data in a protected file using SGX.

```
sgx_status_t ecall_store_data(char* data, size_t data_size, const char* filename) {
    SGX_FILE* file = sgx_fopen_auto_key(filename, "wb");
    if (file == NULL) {
        printf("Error␣opening␣file:␣%s\n", filename);
        return SGX_ERROR_UNEXPECTED;
    }

    for (int i = 0; i < (int) data_size; i++) {
        data[i] = 1;
        printf("%d␣", data[i]);
    }

    data[(int)data_size - 1] = '\0';
    size_t bytes_written = sgx_fwrite(data, sizeof(char), data_size, file);
    printf("bytes␣written␣%d\n", bytes_written);
    if (bytes_written != data_size) {
        printf("Error␣writing␣data␣to␣file:␣%s\n", filename);
        sgx_fclose(file);
        return SGX_ERROR_UNEXPECTED;
    }

    sgx_fclose(file);
    return SGX_SUCCESS;
}
```

9.2 1.2

Listing 9.2 – Function to read data from a protected file using SGX.

```
sgx_status_t ecall_read_data(char* data, size_t data_size, const char* filename) {
    while (*filename == '␣') {
        filename++;
    }
    SGX_FILE* file = sgx_fopen_auto_key(filename, "rb");
    printf("File␣pointer:␣%p\n", file);
    if (file == NULL) {
        printf("Error␣opening␣file:␣%s\n", filename);
        return SGX_ERROR_UNEXPECTED;
    }

    size_t bytes_read = sgx_fread(data, sizeof(char), data_size, file);
    printf("bytes␣read␣%d\n", bytes_read);

    if (bytes_read == 0) {
        printf("Error␣reading␣data␣from␣file:␣%s\n", filename);
        sgx_fclose(file);
        return SGX_ERROR_UNEXPECTED;
    }

    sgx_fclose(file);
    return SGX_SUCCESS;
}
```

REFERENCES

1 BUTERIN, V.; ILLUM, J.; NADLER, M.; SCHäR, F.; SOLEIMANI, A. Blockchain privacy and regulatory compliance: Towards a practical equilibrium. 2023. Disponível em: https://ssrn.com/abstract=4563364.

2 FURST, K.; NOLLE, D. E. Ach payments: Changing users and changing uses. 2005. Disponível em: https://ssrn.com/abstract=2318092.

3 ACH in Numbers: 106 Key Stats and Facts about ACH Payments. 2024. GoCardless.com. Disponível em: https://gocardless.com/en-us/guides/posts/ach-key-stats-facts/.

4 YAO. **A Gentle Introduction to Yao's Garbled Circuits**. 2017. MIT. Disponível em: https://web.mit.edu/sonka89/www/papers/2017ygc.pdf.

5 GENTRY, C. **A fully homomorphic encryption scheme**. 2009. Tese (Doutorado), Stanford, CA, USA, 2009. AAI3382729.

6 REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. In: **Proceedings of the 37th Annual ACM Symposium on Theory of Computing**. Baltimore: [s.n.], 2005. p. 84–93.

7 BRAKERSKI, Z.; GENTRY, C.; VAIKUNTANATHAN, V. **Fully Homomorphic Encryption without Bootstrapping**. 2011. Cryptology ePrint Archive, Paper 2011/277. https://eprint.iacr.org/2011/277. Disponível em: https://eprint.iacr.org/2011/277.

8 BRAKERSKI, Z.; FAN, V.; VERCAUTEREN, F. Fully homomorphic encryption without bootstrapping. **ACM Trans. Comput. Theory**, ACM, New York, NY, USA, v. 6, n. 1, p. 1–29, dez. 2012. ISSN 1930-8017. Disponível em: http://doi.acm.org/10.1145/2408776.2408780.

9 CHEON, J. H.; KIM, M.; KIM, K.; SONG, Y. Homomorphic encryption for arithmetic of approximate numbers. **ACM Trans. Comput. Theory**, ACM, New York, NY, USA, v. 9, n. 1, p. 1–23, jan. 2017. ISSN 1930-8017. Disponível em: http://doi.acm.org/10.1145/3055539.3055540.

10 ROMAN, S. **Field Theory**. 2nd. ed. [S.l.]: Springer, 2005. 237 p.

11 GEELEN, R.; VERCAUTEREN, F. **Bootstrapping for BGV and BFV Revisited**. 2022. Cryptology ePrint Archive, Paper 2022/1363. https://eprint.iacr.org/2022/1363. Disponível em: https://eprint.iacr.org/2022/1363.

12 HALEVI, S.; SHOUP, V. Bootstrapping for helib. **Journal of Cryptology**, v. 34, p. 5, 01 2021.

13 BRAND, M.; PRADEL, G. **Practical Privacy-Preserving Machine Learning using Fully Homomorphic Encryption**. 2023. Cryptology ePrint Archive, Paper 2023/1320. https://eprint.iacr.org/2023/1320. Disponível em: https://eprint.iacr.org/2023/1320.

14 JUMPER, J.; EVANS, R.; PRITZEL, A.; GREEN, T.; FIGURNOV, M.; RONNEBERGER, O.; TUNYASUVUNAKOOL, K.; BATES, R.; ŽíDEK, A.; POTAPENKO, A.; BRIDGLAND, A.; MEYER, C.; KOHL, S. A. A.; BALLARD, A. J.; COWIE, A.; ROMERA-PAREDES, B.; NIKOLOV, S.; JAIN, R.; ADLER, J.; BACK, T.; PETERSEN, S.; REIMAN, D.; CLANCY, E.; ZIELINSKI, M.; STEINEGGER, M.; PACHOLSKA, M.; BERGHAMMER, T.; BODENSTEIN, S.; SILVER, D.; VINYALS, O.; SENIOR, A. W.; KAVUKCUOGLU, K.; KOHLI, P.; HASSABIS, D. Highly accurate protein structure prediction with alphafold. **Nature**, v. 596, n. 7873, p. 583–589, ago. 2021. ISSN 1476-4687. [Online; accessed 15-April-2022]. Disponível em: https://doi.org/10.1038/s41586-021-03819-2.

15 SILDE, T. Short paper: Verifiable decryption for bgv. In: MATSUO, S.; GUDGEON, L.; KLAGES-MUNDT, A.; HERNANDEZ, D. P.; WERNER, S.; HAINES, T.; ESSEX, A.; BRACCIALI, A.; SALA, M. (Ed.). **Financial Cryptography and Data Security. FC 2022 International Workshops**. Cham: Springer International Publishing, 2023. p. 381–390. ISBN 978-3-031-32415-4.

16 MONO, J.; MARCOLLA, C.; LAND, G.; GÜNEYSU, T.; AARAJ, N. Finding and evaluating parameters for bgv. In: MRABET, N. E.; FEO, L. D.; DUQUESNE, S. (Ed.). **Progress in Cryptology - AFRICACRYPT 2023**. Cham: Springer Nature Switzerland, 2023. p. 370–394. ISBN 978-3-031-37679-5.

17 MA, S.; HUANG, T.; WANG, A.; WANG, X. **Accelerating BGV Bootstrapping for Large p Using Null Polynomials Over $\mathbb{Z}_{p^e}$**. 2024. Cryptology ePrint Archive, Paper 2024/115. https://eprint.iacr.org/2024/115. Disponível em: https://eprint.iacr.org/2024/115.

18 NEUBERT, J. C. **Awesome - A curated list of amazing Homomorphic Encryption libraries, software and resources**. 2024. Disponível em: https://github.com/jonaschn/awesome-he.

19 PROTOCOL Buffers. 2024. Disponível em: https://protobuf.dev/overview/.

20 HALEVI, V. S. S. **Params Utility Program for BGV Scheme**. 2023. https://github.com/homenc/HElib/blob/master/misc/params1a.cpp. Accessed: 2024-04-07.

21 MA, S.; HUANG, T.; WANG, A.; WANG, X. **Faster BGV Bootstrapping for Power-of-Two Cyclotomics through Homomorphic NTT**. 2024. Cryptology ePrint Archive, Paper 2024/164. https://eprint.iacr.org/2024/164. Disponível em: https://eprint.iacr.org/2024/164.

22 BIAN, S.; ZHANG, Z.; PAN, H.; MAO, R.; ZHAO, Z.; JIN, Y.; GUAN, Z. He3db: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. In: **Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security**. New York, NY, USA: Association for Computing Machinery, 2023. (CCS '23), p. 2930–2944. ISBN 9798400700507. Disponível em: https://doi.org/10.1145/3576915.3616608.

23 LI, R.; YAMANA, H. Privacy preserving function evaluation using lookup tables with word-wise fhe. **IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences**, 11 2023.

24 ZHOU, H.; WORNELL, G. Efficient homomorphic encryption on integer vectors and its applications. In: **2014 Information Theory and Applications Workshop (ITA)**. [S.l.: s.n.], 2014. p. 1–9.

25 TITLE V, Subtitle A of the Gramm-Leach-Bliley Act. ("GLBA")1 governs the treatment of nonpublic personal information about consumers by financial institutions. **FDIC Consumer Compliance Examination Manual**, April 2021. Disponível em: https://www.fdic.gov/resources/supervision-and-examinations/consumer-compliance-examination-manual/documents/8/viii-1-1.pdf.

26 Federal Deposit Insurance Corporation. **Electronic Fund Transfer Act**. 2024. Federal Deposit Insurance Corporation. Accessed: 2024-04-15. Disponível em: https://www.fdic.gov/sites/default/files/2024-03/fil19009b.pdf.

27 TAKESHITA, J.; MCKECHNEY, C.; PAJAK, J.; PAPADIMITRIOU, A.; KARL, R.; JUNG, T. **GPS: Integration of Graphene, PALISADE, and SGX for Large-scale Aggregations of Distributed Data**. 2021. Cryptology ePrint Archive, Paper 2021/1155. https://eprint.iacr.org/2021/1155. Disponível em: https://eprint.iacr.org/2021/1155.