Master's Thesis

Investigating Resilience of Levelled Fully Homomorphic Encryption System Against Data Scientific Attacks

Untersuchung der Resistenz eines Abgestuften Homomorphen Verschlüsselungssystems Gegen Datenwissenschaftliche Angriffe

Aaruni Kaushik

February 15, 2023

supervised by

Prof. Dr. Claus Fieker

Rheinland-Pfälzische Technische Universität
Kaiserslautern-Landau

Statutory Declaration

I hereby declare that I have developed and written this thesis completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others are clearly marked.

Aaruni Kaushik Location, Date

Abstract

In this thesis we look at the Levelled Fully Homomorphic Encryption scheme described by Craig Gentry, Amit Sahai, and Brent Waters in [GSW13]. We describe an explicit implementation of the scheme in `python`, and provide a method to choose parameters for encryption. We also explain our attempt at attacking the system using data science tools instead of classical cryptanalysis.

Acknowledgements

I would like to thank Dr Claus Fieker for agreeing to guide my reading course and supervise my thesis, Ms Yvonne Weber for sharing her own thesis for reference, and all of my friends and family who have supported my work.

Contents

1 Introduction

1.1 Fully Homomorphic Encryption (FHE)

Homomorphic encryption is a form of encryption that permits computations on encrypted data without having to first decrypt. The decryption of this computation is identical to the same computation performed on unencrypted data. This type of encryption can be used for privacy-perserving cloud storage and computation. It allows data to be encrypted and outsourced to commercial cloud environments for processing, while never being decrypted. This ensures that users can still avail all the benefits of cloud computing, while also guaranteeing privacy of data even against the offsite service provider.

Some simple ways in which homomorphic cryptography may be applied are :

- Navigation
  An online map provider can run the "difference" operation homomorphically on the encrypted copies of your location and destination without ever finding out where you are and where you are going.

- Healthcare
  A medical institute could safely offload data management to a homomorphically encrypted cloud provider, and then edit patient records directly on the cloud without ever revealing what changes were made to whom.

- e-Voting
  A ballot system can be designed where each vote is a homomorphic addition, and it can not be determined to which candidate you have given your vote, while keeping the final vote count accurate.

1.2 Brief History

The problem of constructing a fully homomorphic cryptographic scheme was first proposed by Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos in 1978 [RAD78], but had remained merely an academic question for over 30 years as no one could come up with a practical design. Craig Gentry, using lattice-based cryptography, described the first plausible construction for a fully homomorphic encryption scheme in 2009 [Gen09]. This came to be known as *First Generation FHE*. The basic idea of this scheme is to hide the message inside a small error, called *noise*, which the decryption process is resilient against. The level of noise compounds with every homomorphic operation, and beyond a point grows to a level where decryption would be impossible. To combat this, an expensive operation called *bootstrapping* must be performed, which reduces noise.
Building on improvements and relying on the hardness of the (Ring) Learning With Errors (RLWE) problem, the distinguishing characteristic of the second-generation cryptosystems is that they all feature a much slower growth of the noise during the homomorphic computations, and that they are efficient enough for many applications even without invoking bootstrapping, instead operating in the leveled FHE mode, allowing a limited number of operations before the noise gets out of hand.
Third Generation FHE was sparked off by the efforts of Craig Gentry, Amit Sahai, and Brent Waters in 2013 [GSW13] who proposed a new technique for building FHE schemes that avoid expensive relinearization steps in homomorphic multiplication. The notable schemes of this

generation are FHEW [DM14] TFHE [Chi+16]. The scheme we focus on in this thesis is from [GSW13], the first Third Generation FHE scheme.
In 2016, Cheon, Kim, Kim, and Song (CKKS) proposed a scheme [Che+17] which includes an efficient rescaling operation that scales down an encrypted message after a multiplication. This avoids the expensive bootstrapping involved in earlier schemes. This is the current generation of Homomorphic Encryption, called the *Fourth Generation FHE.*

1.3 Organization

In Section 2, we list out the notation we use. In Section 3, we introduce the mathematical building blocks required to proceed with the thesis. In Section 4 we formally present the cryptosystem given by Gentry, Sahai, and Waters [GSW13], and provide proofs of correctness of the system. Using the proofs, we deduce a method for explicitly choosing parameters for the system, and reformulate the system for better readability. In Section 5 we do some investigational cryptanalysis of the system using data science, and conclude our findings. Explicit implementation, and data scientific background are provided in Appendix 7.

1.4 Related Works

The data scientific techniques we employ are provided in detail in the article [KKP23]. My (Aaruni Kaushik) contribution to [KKP23] was in actually implementing and optimizing the TDA pipeline 7.1.3 designed by the other authors, generating the data for analysis, and running all computations.

2 Notation

This section will list out explicitly all the notation we use

Π	Cryptosystem
$\mathcal{P}$	Set of Plaintexts
$\mathcal{C}$	Set of Ciphertexts
$\mathcal{E}$	Encryption algorithm
$\mathcal{D}$	Decryption algorithm
$\mathcal{D}_{\{0,1\}}$	Decryption algorithm for parity
$\mathbb{Z}$	Set of Integers
$\mathbb{Z}_q$	Set of Integers modulo q
$\mathbb{L}$	Multiplicative depth of scheme
λ	Parameter for security level
χ	LWE distribution
m	Size of the error vector drawn from χ
$\leftarrow$	Means we are drawing a sample observation from a distribution
$\lfloor x \rfloor$	Greatest integer function applied to x
$\lceil x \rceil$	Lowest integer function applied to x
$\lfloor x \rceil$	x rounded off to the closest integer
LSB(x)	Least Significant Bit of the binary representation of x
$\langle \vec{a}, \vec{b} \rangle$	Scalar product between $\vec{a}$ and $\vec{b}$

3 Prerequisites

In this section we aim to address simple definitions and properties so that the next section begins to make sense. We assume very little and try to eliminate the need for an external reference to make sense of our work. While there are quite a few definitions, we mostly only reiterate standard definitions which an initiated reader could ignore. The only non-standard definitions are in Section 3.3, and are taken from [GSW13].

3.1 Cryptographic Background

Definition 3.1 (Cryptosystem):
A **cryptosystem** is a 5-tuple $\Pi := (\mathcal{P}, \mathcal{C}, \kappa, \mathcal{E}, \mathcal{D})$, where

1. $\mathcal{P}$ is the set of *Plaintexts*. This is where all messages live.

2. $\mathcal{C}$ is the set of *Cyphertexts*. This is where all encryptions live.

3. κ is the bijective map between the set of *Encryption Keys* and the set of *Decryption Keys*

4. $\mathcal{E}$ is the *Encryption algorithm*, which uses an encryption key e to encrypt a plaintext to give a ciphertext.

$$\begin{aligned} \mathcal{E}_e : \mathcal{P} &\to \mathcal{C} \\ p &\mapsto \mathcal{E}_e(p) \end{aligned}$$

5. $\mathcal{D}$ is the *Decryption algorithm*, which uses a decryption key d to decrypt a ciphertext to return a plaintext. It is the inverse operation of $\mathcal{E}$.

$$\begin{aligned} \mathcal{D}_d : \mathcal{C} &\to \mathcal{P} \\ \mathcal{E}_e(p) &\mapsto p \end{aligned}$$

Definition 3.2 (Security Properties):
For a given cryptosystem Π we define the following **security properties**:

1. Π is said to be *one way (OW)* if it is infeasible for an attacker to decrypt an arbitrary ciphertext.

2. Π is said to be *indistinguishable (IND)* if it is infeasible for an attacker to associate a given ciphertext to one of several known plaintexts.

3. Π is said to be *non-malleable (NM)* if it is infeasible for an attacker to modify a given ciphertext in a way such that the corresponding plain text is sensible in the given language respectively context.

Remark 3.3 :
Loosely, a problem is infeasible if it takes too many resources to compute. For the purposes of this thesis, we simplify this notion to mean that a problem is infeasible if there is no solution with better time complexity than $O(2^n)$, that is, the best solution to the problem is to simply try every possible answer until you find one that fits.

Definition 3.4 (Active Attack):
An active attack on a cryptosystem is one in which the attacker actively changes the communication by, for example, creating, altering, replacing or blocking messages.

Definition 3.5 (Passive Attack):
A **passive attack** on a cryptosystem is one in which the attacker only eavesdrops plaintexts and ciphertexts. The attacker cannot alter any messages they see.

Remark 3.6 :
We only outline passive attacks in this thesis.

Definition 3.7 (Attack Scenario):
One can choose one, or a combination, of the following kinds of attacks:

1. In a **Cipehertext Only Attack** (COA), the attacker receives only ciphertexts, for example, a database of encrypted passwords.
2. In a **Known Plaintext Attack** (KPA), the attacker receives pairs of plaintexts and corresponding ciphertexts.
3. In a **Chosen Plaintext Attack** (CPA), the attacker can choose arbitrary plaintexts and is able to receive the corresponding ciphertexts.
4. In a **Adaptive Chosen Ciphertext Attack** (CCA), an attacker is able to adaptively choose ciphertexts and to receive the corresponding plaintexts. The attacker is allowed to alter the choice depending on what is received. So the attacker has access to the decryption algorithm $\mathcal{D}_d$ and wants to get to know the decryption key d.

Example 3.8 (Known Plaintext Attack):

- The Caesar cipher is vulnerable to Known Plaintext Attack.
- The attack against the Enigma machine was (at least in part), a Known Plaintext Attack. [Sin00]

Definition 3.9 (Security Model):
A **security model** is a security property together with an attack scenario.

Example 3.10 :
IND-CPA is a security model, where one would check the indistinguishability property of a given cryptosystem Π with respect to a Chosen Plaintext Attack.
Let's say Alice sends a secret message to Bob using a cryptosystem Π, and Eve wants to read this message. In the CPA attack, Eve is allowed to encrypt as many messages as she wants using the encryption algorithm $\mathcal{E}$ from Π. If Eve is then able to find a way to meaningfully distinguish between the encryptions and find the decryption keys, then the indistinguishability property of the cryptosystem is voided, and Eve has successfully carried out an IND-CPA attack.
On the other hand, if the system is immune to this particular class of attacks, then the system is said to be IND-CPA secure.

Remark 3.11 :
In an asymmetric cryptosystem, mounting the Chosen Plaintext Attack is trivial, as the encryption parameters are published openly, and can be done on the attacker's computer without any limits. The success of this attack depends wholly on the security provided by the system.

Remark 3.12 :
The attack used in this thesis is a Chosen Plaintext Attack.

Definition 3.13 (Homomorphic Cryptosystem):
A **Homomorphic Cryptosystem** is a cryptosystem Π along with one or more evaluation functions $f' : \mathcal{C} \to \mathcal{C}$ such that, for some function $f : \mathcal{P} \to \mathcal{P}$

$$f'((\mathcal{E}(a)), \mathcal{E}(b), f) = f(a, b)$$

Types of homomorphic encryption are:

- Partially homomorphic encryption
- Somewhat homomorphic encryption
- Leveled fully homomorphic encryption
- Fully homomorphic encryption

Definition 3.14 (Partially homomorphic encryption):
Partially homomorphic encryption are schemes that support the evaluation of only one type of operation, e.g., addition or multiplication.

Definition 3.15 (Somewhat homomorphic encryption):
Somewhat homomorphic encryption schemes can evaluate two types of operations, but only for a subset of functions.

Definition 3.16 (Leveled fully homomorphic encryption):
Leveled fully homomorphic encryption supports the evaluation of arbitrary functions composed of multiple types, but bounded (pre-determined) number of operations.

Definition 3.17 (Fully homomorphic encryption):
Fully homomorphic encryption allows the evaluation of arbitrary functions composed of multiple types of operations of unbounded depth and is the strongest notion of homomorphic encryption.

Remark 3.18 :
It is nice to have a homomorphic cryptosystem where f' is the "natural" map of f from $\mathcal{P}$ to $\mathcal{C}$. For example, if $\mathcal{P} = \mathbb{Z}_q$ and $\mathcal{C} = \mathbb{Z}_q^{N \times N}$, then we would like to obtain a scheme where f is simple integer addition (modulo q), and f' is simply matrix addition (modulo q). The system described in [GSW13] was the first homomorphic cryptosystem to provide this nicety.

Remark 3.19 :
By definition, Fully Homomorphic Cryptography Schemes are Malleable, and therefore, can never have all three security properties defined in Definition 3.2.

Definition 3.20 (LWE Problem):
For security parameter λ, let $n = n(\lambda)$ be an integer dimension, let $q = q(\lambda) \geq 2$ be an integer, and let $\chi = \chi(\lambda)$ be a distribution over $\mathbb{Z}_q$, The $\mathrm{LWE}_{n,q,\chi}$ problem is to distinguish the following two distributions:

- Uniform samples $(\vec{a_i}, b_i) \leftarrow \mathbb{Z}_q^{n+1}$
- Samples $(\vec{a_i}, b_i) \in Z_q^{n+1}$ where:
  $\vec{a_i} \leftarrow \mathbb{Z}_q^n$ uniformly
  $\vec{s} \leftarrow \mathbb{Z}_q^n$ uniformly, $e_i \leftarrow \chi$, and set $b_i := \langle \vec{a_i}, \vec{s} \rangle + e_i$

The $\mathrm{LWE}_{n,q,\chi}$ assumption is that the $\mathrm{LWE}_{n,q,\chi}$ problem is infeasible.

3.2 Number Theoretic Background

Definition 3.21 (Lattice):
Let $b_1, \dots, b_k \in \mathbb{R}^n$ be $\mathbb{R}$-linear independent. Then,

$$\Lambda := +\mathbb{Z}b_i$$

is called a *lattice* or, sometimes, a $\mathbb{Z}$-lattice in $\mathbb{R}^n$

Definition 3.22 (Shortest Vector Problem [Mic11]):
Given k linearly independent integer vectors $B = [b_1, \dots, b_k]$ in n-dimensional Euclidean space R^n, the Shortest Vector Problem (SVP) asks to find a nonzero linear combination $Bx = \sum_{i=1}^{k} b_i x_i$(with $x \in \mathbb{Z}^k \backslash \{0\}$) such that the norm $||B_x||$ is as small as possible.
SVP can be concisely defined as the problem of finding the shortest nonzero vector in the lattice represented by B.
The SVP problem is NP-Hard

Definition 3.23 ($GapSVP_g$ [Mic11]):
Given a lattice basis B, values d, g, the decision problem to determine if

$$\lambda_1(B) \leq d \text{ or } \lambda_1(B) > g \cdot d$$

is denoted the $GapSVP_g$.
The $GapSVP_g$ problem is also NP-Hard.

3.3 Helpful Functions

In this section we define some helper operations we will use in our cryptosystem, as found in [GSW13].

Definition 3.24 (BitDecomp):
For any vector $\vec{a} = (a_1, a_2, \dots, a_k)$ of length k, and l the number of bits in the largest $a_i \in \vec{a}$

$$\mathrm{BitDecomp}(\vec{a}) = (a_{1,0}, a_{1,1}, \dots, a_{1,l-1}, \dots, a_{k,0}, \dots, a_{k,l-1})$$

where $a_{1,0}$ is $\mathrm{LSB}(a_1)$, and $a_{k,l-1}$ is the l^{th} bit in the binary representation of a_k, and may be 0.

Definition 3.25 (BitDecomp^{-1}):
For any vector $\vec{a'} = (a_{1,0}, a_{1,1}, \ldots, a_{1,l-1}, \ldots, a_{k,0}, \ldots, a_{k,l-1})$ of length $N = k \cdot l$,

$$\text{BitDecomp}^{-1}(\vec{a'}) = (\sum 2^j \cdot a_{1,j}, \ldots, \sum 2^j \cdot a_{k,j})$$

Remark 3.26 :
We abuse our notation when we define BitDecomp^{-1}*, as it is not a true inverse of* BitDecomp*. If one starts with $\vec{a} \in \{0,1\}^N$, then these operations invert each other, but* BitDecomp$^{-1}(\vec{a})$ *is well-defined even when $\vec{a} \in \mathbb{Z}_q{}^N$.*

Definition 3.27 (Flatten):
For any vector $\vec{a'}$ of length $N = k \cdot l$,

$$\text{Flatten}(\vec{a'}) = \text{BitDecomp}(\text{BitDecomp}^{-1}(\vec{a'}))$$

Definition 3.28 (PowersOf2):
For any vector $\vec{b} = (b_1, b_2, \ldots, b_k)$,

$$\text{PowersOf2}(\vec{b}) = (2^0 b_1, 2^1 b_1, \ldots, 2^{l-1} b_1, \ldots, 2^0 b_k, s^1 b_k, \ldots, 2^{l-1} b_k)$$

Remark 3.29 :
Technically, PowersOf2 *also depends on l, but we skip writing it as a formal parameter of the function as l is a fixed parameter for a given cryptosystem, and we only use* PowersOf2 *within the context of a cryptosystem.*

Remark 3.30 :
We extend the above operations to matrices by applying them row wise to the matrix.

We now look at two useful properties of these newly defined functions.

Theorem 3.31 :
$\langle \text{BitDecomp}(\vec{a}), \text{PowersOf2}(\vec{b}) \rangle = \langle \vec{a}, \vec{b} \rangle$.

Proof.
From Definitions 3.24 and 3.28,

$$\text{BitDecomp}(\vec{a}) = (a_{1,0}, a_{1,1}, \ldots, a_{1,l-1}, \ldots, a_{k,0}, a_{k,1}, \ldots, a_{k,l-1}) \tag{1}$$
$$\text{PowersOf2}(\vec{b}) = (2^0 b_1, 2^1_b 1, \ldots, 2^{l-1} b_1, \ldots, 2^0 b_k, 2^1 b_k, \ldots, 2^{l-1} b_k) \tag{2}$$

$$\begin{aligned} \langle (1), (2) \rangle &= (a_{1,0} 2^0 b_1 + a_{1,1} 2^1 b_1 + \cdots + a_{1,l-1} 2^{l-1} b_1 + \cdots + \\ &\quad a_{k,0} 2^0 b_k + a_{k,1} 2^1 b_k + \cdots + a_{k,l-1} 2^{l-1} b_k) \\ &= (b_1 \cdot (\sum 2^j a_{1,j}) + b_2 \cdot (\sum 2^j a_{2,j}) + \cdots + b_k \cdot (\sum 2^j a_{k,j})) \\ &= (\sum a_i \cdot b_i) = \langle \vec{a}, \vec{b} \rangle \end{aligned}$$

$\square$

Remark 3.32 :
It is not obvious, but the previous proof depends on the value of l. In particular, l must be high enough that each a_i is decomposed into bits without losing any significant bits. We have defined BitDecomp *in a way to always keep this fact be true.*

Theorem 3.33 :

$$\langle \vec{a'}, \text{PowersOf2}(\vec{b}) \rangle = \langle \text{BitDecomp}^{-1}(\vec{a'}), \vec{b} \rangle = \langle \text{Flatten}(\vec{a'}), \text{PowersOf2}(\vec{b}) \rangle$$

Proof.

$\langle \vec{a'}, \text{PowersOf2}(\vec{b}) \rangle = \langle \text{BitDecomp}^{-1}(\vec{a'}), \vec{b} \rangle$:
By Definition 3.25,

$$\text{BitDecomp}^{-1}(\vec{a'}) = (\sum 2^j a_{1,j}, \sum 2^j a_{2,j}, \dots, \sum 2^j a_{k,j}) \tag{3}$$

Hence, we obtain

$$\begin{aligned}\langle \text{BitDecomp}^{-1}(\vec{a'}), \vec{b} \rangle &= (\sum 2^j a_{1,j} b_1 + \sum 2^j a_{2,j} b_2 + \dots + \sum 2^j a_{k,j} b_k) \\ &= (a_{1,0} \cdot 2^0 b_1 + a_{1,1} \cdot 2^1 b_1 + \dots + a_{k,l-1} \cdot 2^{l-1} b_k) \\ &= \langle \vec{a'}, \text{PowersOf2}(\vec{b}) \rangle\end{aligned}$$

$\langle \text{BitDecomp}^{-1}(\vec{a'}), \vec{b} \rangle = \langle \text{Flatten}(\vec{a'}), \text{PowersOf2}(\vec{b}) \rangle$:
By Definition 3.27,

$$\text{Flatten}(\vec{a'}) = \text{BitDecomp}(\text{BitDecomp}^{-1}(\vec{a'}))$$

Then, by Theorem 3.31,

$$\langle \text{BitDecomp}(\text{BitDecomp}^{-1}(\vec{a'})), \text{PowersOf2}(\vec{b}) \rangle = \langle \text{BitDecomp}^{-1}(\vec{a'}), \vec{b} \rangle$$

□

Remark 3.34 :
The proof for Theorem 3.33 is quite simple intuitively. All we really are doing is moving around the powers of 2. As BitDecomp^{-1} *"generates" powers of 2, we can use it on the first operand of the inner product when we remove* PowersOf2 *from second operand and preserve the final answer.*
Similarly, for the second part, as BitDecomp *"removes" powers of 2, we can use it on the first operand, and apply* PowersOf2 *on the second operand to keep the final value of the scalar product the same.*

Remark 3.35 :
Flatten() *is a useful tool for us as it reduces coefficients of vectors to bits while preserving the product of its input vector with* PowersOf2(). *In our cryptosystem, we use Theorem 3.33 to hide the plain text by flattening it, without losing our ability to recover the plaintext by looking at its inner product with* PowersOf2(*secret key*)

4 GSW LFHE

In this section, we formally introduce the cryptosystem. First, we reproduce it as described by [GSW13]. We proceed by verifying that the system is correct, i.e., applying the decryption algorithm after the encryption algorithm and checking if it really gives what we started out with. Then we provide an explicit way of choosing paramaters, before finally reformulating the cryptosystem for clarity, and present our version of the same system.

4.1 GSW Encryption Scheme

The crypto system as described in [GSW13] is as follows.

Setup(1^λ, 1^L):

Choose a modulus q of $\kappa = \kappa(\lambda, L)$ bits, lattice dimension parameter $n = n(\lambda, L)$, and error distribution $\chi = \chi(\lambda, L)$ appropriately for LWE that achieves at least 2^λ security against known attacks. Also, choose parameter $m = m(\lambda, L) = O(n \log(q))$ Let $params = (n, q, \chi, m)$. Let $l = \lfloor \log_2(q) \rfloor + 1$, and $N = (n+1) \cdot l$

SecretKeyGen(*params*):

Sample $\vec{t} \leftarrow \mathbb{Z}_n^q$. Output $sk = \vec{s} \leftarrow (1, -t_1, -t_2, \ldots, -t_n) \in \mathbb{Z}_q^{n+1}$. Let $\vec{v} = \text{PowersOf2}(\vec{(s)})$

PublicKeyGen(*params, pk*):

Generate a matrix $B \leftarrow \mathbb{Z}_q^{m \times n}$ uniformly and a vector $\vec{e} \leftarrow \chi^m$. Set $\vec{b} = B \cdot \vec{t} + \vec{e}$. Set A to be the $(n+1)$-column matrix consisting of $\vec{b}$ followed by the n columns of B. Set the public key $pk = A$. (Remark: Observe that $A \cdot \vec{s} = \vec{e}$)

$\mathcal{E}_{pk}$(*params, μ*):

To encrypt a message $\mu \in \mathbb{Z}_q$, sample a uniform matrix $R \in \{0,1\}^{N \times m}$ and output the ciphertext C given below.

$$C = \text{Flatten}(\mu \cdot I_N + \text{BitDecomp}(R \cdot A)) \in \mathbb{Z}_q^{N \times N}$$

$\mathcal{D}_{\{0,1\}sk}$(*params, C*):

Observe that the first l coefficients of $\vec{v}$ are $1, 2, \ldots, 2^{l-1}$. Among these coefficients, let $v_i = 2^i$ be in $(\frac{q}{4}, \frac{q}{2}]$. Let C_i be the i^{th} row of C. Compute $x_i \leftarrow \langle C_i, \vec{v} \rangle$.
Output $\mu' = \lfloor x_i / v_i \rceil$.

$\mathcal{D}_{sk}$(*params*, C) (for q a power of 2):

Observe that $q = 2^{l-1}$ and the first $l-1$ coefficients of $\vec{v}$ are $1, 2, \ldots, 2^{l-2}$, and therefore if $C \cdot \vec{v} = \mu + small$, then the first $l-1$ coefficients of $C \cdot \vec{v}$ are $\mu \cdot \vec{g} + small$, where $\vec{g} = (1, 2, \ldots, 2^{l-2})$. Recover the LSB($\mu$) from $\mu \cdot 2^{l-2} + small$, then recover the next-least-significant-bit from $(\mu - \text{LSB}(\mu)) \cdot 2^{l-3} + small$, etc. (See [MP12] for general q case.)

Add$(C_1, C_2, +)$:

To add ciphertexts $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ output Flatten$(C_1 + C_2)$. Note that the addition of messages is over the full base ring $\mathbb{Z}_q$.

Remark 4.1 :
The paper [GSW13] also defines homorphic multiplication functions MultConst(C, α) and Mult(C_1, C_2), and homomorphic NAND function NAND(C_1, C_2). However, we choose not to focus on those. While MultConst(C, α) is provided in implementation 7.2, we completely ignore Mult(C_1, C_2) as the massive growth in noise would force us to implement bootstrapping. The homomorphic NAND is of no particular interest to us, but is quite useful when trying to build real circuits which implement this scheme.

4.2 Correctness of Decryption

Theorem 4.2 :
The decryption function for a small message space is correct, i.e.,

$$\mathcal{D}_{\{0,1\}sk}(params, \mathcal{E}_{pk}(params, \mu)) = \mu, \ \ for \ \mu \in \{0,1\}$$

Proof.

Encryption Let us begin by encrypting a plaintext $\mu \in \{0,1\}$

Recall that by definition, the ciphertext is given by

$$C = \mathcal{E}_{pk}(params, \mu) = \text{Flatten}(\mu \cdot I_N + \text{BitDecomp}(R \cdot A)),$$

where $R \in \{0,1\}^{N\times m}$ and $A \in \mathbb{Z}_q^{m\times n+1}$

Let $R \cdot A = [(RA)_{i\ j}]$, and set $D := \text{BitDecomp}(R \cdot A) \in \{0,1\}^{N\times N}$. We use a triply indexed $(RA)_{i\ j\ k}$ to mean the $(k+1)^{th}$ bit of $(RA)_{i\ j}$
Then,

$$D = \begin{bmatrix} (RA)_{1\ 1\ 0} & (RA)_{1\ 1\ 1} & \cdots & (RA)_{1\ 1\ l-1} & (RA)_{1\ 2\ 0} & (RA)_{1\ 2\ 1} & \cdots & (RA)_{1\ n+1\ l-1} \\ (RA)_{2\ 1\ 0} & (RA)_{2\ 1\ 1} & \cdots & (RA)_{2\ 1\ l-1} & (RA)_{2\ 2\ 0} & (RA)_{2\ 2\ 1} & \cdots & (RA)_{2\ n+1\ l-1} \\ \cdots \\ (RA)_{N\ 1\ 0} & (RA)_{N\ 1\ 1} & \cdots & (RA)_{N\ 1\ l-1} & (RA)_{N\ 2\ 0} & (RA)_{N\ 2\ 1} & \cdots & (RA)_{N\ n+1\ l-1} \end{bmatrix}$$

$$\begin{aligned} BF &:= \mu \cdot I_N + D \\ &= \begin{bmatrix} \mu + (RA)_{1\ 1\ 0} & (RA)_{1\ 1\ 1} & \cdots & \cdots & (RA)_{1\ n+1\ l-1} \\ (RA)_{2\ 1\ 0} & \mu + (RA)_{2\ 1\ 1} & \cdots & \cdots & (RA)_{2\ n+1\ l-1} \\ \cdots \\ (RA)_{N\ 1\ 0} & (RA)_{N\ 1\ 1} & \cdots & \cdots & \mu + (RA)_{N\ n+1\ l-1} \end{bmatrix} \\ C &= \text{Flatten}(BF) \\ &= \begin{bmatrix} (\mu + (RA)_{1\ 1\ 0})' & ((RA)_{1\ 1\ 1})' & \cdots & \cdots & ((RA)_{1\ n+1\ l-1})' \\ ((RA)_{2\ 1\ 0})' & (\mu + (RA)_{2\ 1\ 1})' & \cdots & \cdots & ((RA)_{2\ n+1\ l-1})' \\ \cdots \\ ((RA)_{N\ 1\ 0})' & ((RA)_{N\ 1\ 1})' & \cdots & \cdots & (\mu + (RA)_{N\ n+1\ l-1})' \end{bmatrix} \end{aligned}$$

Decryption

Let $v_i = 2^i \in (\frac{q}{4}, \frac{q}{2}]$; $v \in \mathbb{Z}_q^N$

Let C_i be the i^{th} row of C, $i < l-1$

Then, the i^{th} row of C_i is

$$((RA)_{i\ 1\ i-1} + \mu)' \in \{0, 1\}$$

Compute $x_i \leftarrow \langle C_i, \vec{v} \rangle$:

$$\begin{aligned}
C_i &= \begin{bmatrix} (RA)'_{i\ 1\ 0} & (RA)'_{i\ 1\ 1} & \cdots & ((RA)_{i\ 1\ i-1} + \mu)' & \cdots & (RA)'_{i\ n+1\ l-1} \end{bmatrix} \\
v &= \begin{bmatrix} 2^0 & 2^1 & \cdots & 2^i & \cdots & -t_n 2^{l-1} \end{bmatrix} \\
x_i &= \langle C_i, \vec{v} \rangle \\
&= 2^i((RA)_{i\ 1\ i-1} + \mu)' + 2^0 (RA)'_{i\ 1\ 0} + 2^1 (RA)'_{i\ 1\ 1} + \cdots - t_n \cdot 2^{l-1} \cdot (RA)'_{i\ n+1\ l-1}
\end{aligned}$$

As $\vec{v} = \text{PowersOf2}(\vec{sk})$, and Flatten() preserves the product with PowersOf2() (Theorem 3.33), we have

$$\begin{aligned}
x_i &= 2^i((RA)_{i\ 1\ i-1} + \mu) + 2^0 (RA)_{i\ 1\ 0} + 2^1 (RA)_{i\ 1\ 1} + \cdots - t_n \cdot 2^{l-1} \cdot (RA)_{i\ n+1\ l-1} \\
&= 2^i \mu + (RA)_{i\ 1} - t_1 (RA)_{i2} - \cdots - t_n (RA)_{i\ n+1}
\end{aligned}$$

Then,

$$\lfloor x_i / v_i \rceil = \lfloor x_i / 2^i \rceil = \lfloor \mu + \frac{error}{2^i} \rceil = \mu$$

□

Theorem 4.3 :
The general case decryption is correct, i.e.,

$$\mathcal{D}_{sk}(params, \mathcal{E}_{pk}(params, \mu)) = \mu$$

Proof.

Encryption

As before, we run the encryption function on a $\mu \in \mathbb{Z}_q$. We get

$$\begin{aligned}
C &= \text{Flatten}(BF) \\
&= \begin{bmatrix}
(\mu + (RA)_{1\ 1\ 0})' & ((RA)_{1\ 1\ 1})' & \cdots & \cdots & ((RA)_{1\ n+1\ l-1})' \\
((RA)_{2\ 1\ 0})' & (\mu + (RA)_{2\ 1\ 1})' & \cdots & \cdots & ((RA)_{2\ n+1\ l-1})' \\
\cdots & & & & \\
((RA)_{N\ 1\ 0})' & ((RA)_{N\ 1\ 1})' & \cdots & \cdots & (\mu + (RA)_{N\ n+1\ l-1})'
\end{bmatrix}
\end{aligned}$$

Decryption

$$\begin{aligned}\vec{v} &= \text{PowersOf2}(\vec{sk})\\ &= (2^0, 2^1, \ldots, 2^{l-1}, -t_1 2^0, -t_1 2^1, \ldots, -t_1 2^{l-1}, \ldots, -t_n 2^{l-1}) \in \mathbb{Z}_q^N\end{aligned}$$

$$< C, \vec{v} > = \begin{bmatrix} 2^0\mu + (RA)_{1\ 1} - (t_1(RA)_{1\ 2} + t_2(RA)_{1\ 3} + \cdots + t_n(RA)_{1\ n+1}) \\ 2^1\mu + (RA)_{2\ 1} - (t_1(RA)_{2\ 2} + t_2(RA)_{2\ 3} + \cdots + t_n(RA)_{2\ n+1}) \\ \cdots \\ 2^{l-1}\mu + (RA)_{l-1\ 1} - (t_1(RA)_{l-1\ 2} + t_2(RA)_{l-1\ 3} + \cdots + t_n(RA)_{l-1\ n+1}) \\ \cdots \\ N - l + 1 \ \textit{lines of noise} \\ \cdots \end{bmatrix}$$

$$= \begin{bmatrix} 2^0\mu + \textit{small} \\ 2^1\mu + \textit{small} \\ \cdots \\ 2^{l-1}\mu + \textit{small} \\ \cdots \\ \textit{Noise} \\ \cdots \end{bmatrix}$$

We reconstruct our μ bit by bit from $\langle C, \vec{v} \rangle$ as follows:

1^{st} Bit:

Take μ_1' to be the $l - 1^{st}$ element of $\langle C, \vec{v} \rangle$. That is,

$$\mu_1' = 2^{l-2}\mu + \textit{small} = 2^{l-2}\mu_1 + \textit{small}$$

Then, $\mu_1 = \lfloor \frac{\mu_1'}{2^{l-2}} \rceil$

2^{nd} Bit:

Take μ_2' to be the $l - 2nd$ element of $\langle C, \vec{v} \rangle$. That is

$$\mu_2' = 2^{l-3}\mu + \textit{small} = 2^{l-3}(2\mu_2 + \mu_1) + \textit{small}$$

Then, $\mu_2 = \lfloor \frac{\mu_2' - 2^{l-3}\mu_1}{2^{l-2}} \rceil$

We continue this process till we recover μ_{l-1}, and then we assemble our plaintext

$$\mu = \sum_{i=0}^{l} 2^i \cdot \mu_{i+1}$$

$\square$

4.3 Choice of Parameters

The proof for Theorem 4.2 also gives us some information to help with choosing our parameters for encryption.

As $\lfloor x_i/v_i \rceil = \lfloor \mu + \frac{error}{2^i} \rceil = \mu$, x_i/v_i can be at most $\frac{1}{2}$ away from the value of μ. So, $\frac{error}{2^i} \leq \frac{1}{2}$

$$\implies error \leq 2^{i-1}$$

Recall from Section 4.1 that $2^i \in (\frac{q}{4}, \frac{q}{2}] \implies 2^{i-1} \in (\frac{q}{8}, \frac{q}{4}]$. So, we get an upper bound for the estimate, namely,

$$error \leq \frac{q}{8}$$

Our error term depends on the value of the sampled $\vec{t}$, the random matrix B, and $\vec{e} \leftarrow \chi^m$. However, only $\vec{e}$ is a choice we make (by choosing an appropriate χ), as the other two samples are uniformly random. Therefore, we choose χ such that $\sum e_i \forall e_i \in \vec{e}$ keeps under the obtained upper bound.
As q must always be a power of 2, and it can be exponential in n, it is a straightforward choice to make.

$$q = 2^n$$

As $m > 2 \cdot n \cdot \log_2(q)$, we choose m just greater than its bound.

$$m = 2 \cdot n \cdot \log_2(q) + 1$$

Armed with this knowledge, and some trial and error with the LWE-Estimator tool [APS15], we arrive at the final parameters.
For 129 bits of security, a set of secure parameters for this scheme are calculated by [APS15] to be:

$$\begin{aligned} n =&\ 768 \\ \lambda =&\ 1 \\ q =&\ 2^{22} \\ m =&\ 2 \cdot n \cdot \lfloor \log_2(q) \rfloor + 1 \\ \chi =&\ N_{\text{discrete}}(\mu = \tfrac{q}{2^{3+\lambda} \cdot m}, \sigma^2 = 3.0) \end{aligned}$$

So, on a reasonably fast computer, it is theorised to take $10^{12} \times$ **age of the universe** to break this scheme via the best known attack. That is, the best attack should be of the complexity at least $\mathcal{O}(n^{10.33})$.

Remark 4.4 :
[APS15] cautions the reader against using it as a reference for the state of the art in assessing the cost of solving LWE or making sense of the LWE estimator.

4.4 Reformulated GSW13

With the help of things we learnt above, we are now ready to reformulate the GSW13 cryptosystem.

Remark 4.5 :
Care must be taken that q is large enough that the choice of λ make sense. That is, $\frac{q}{2^{3+\lambda}m}$ should be a large enough positive number so that the variance supplied to the Discrete Gaussian does not return negative integer samples. The `python` implementation 7.2 will intentionally crash if the Discrete Gaussian ever samples a negative number.

Setup(n,λ):

Choose q according to the desired security level and set $l = \log_2(q) + 1 = n + 1$
Choose $\chi = N_{\text{discrete}}(\mu = \frac{q}{2^{3+\lambda}m}, \sigma^2)$, where σ is chosen according to the desired security level.
Set $m = 2 \times n \times l$, and $N = (n+1) \times l$
Return parameters

$$\textit{params} = (q, n, m, l, N, \chi)$$

SecretKeyGen(*params*):

Sample $t \leftarrow \mathbb{Z}_q^n$ uniformly.
Return secret key

$$\vec{sk} = (1, -t_1, \ldots, -t_n) \in \mathbb{Z}_q^{n+1}$$

PublicKeyGen (*params*, sk):

Sample a matrix $B \leftarrow \mathbb{Z}_q^{m \times N}$and an error vector $\vec{e} \leftarrow \chi^m$
Set $\vec{b} = B \cdot \vec{t} + \vec{e}$ and $A = (\vec{b}, B) \in \mathbb{Z}_q^{m \times n+1}$.
Return public key

$$\vec{pk} = A$$

$\mathcal{C}_{pk}$(*params*, μ):

Sample $R \leftarrow 0, 1^{N \times m}$ uniformly.
Return ciphertext

$$C = \text{Flatten}(\mu \cdot I_N + \text{BitDecomp}(R \cdot A)) \in \{0, 1\}^{N \times m}$$

$\mathcal{D}_{\{0,1\}sk}$(params, C):

Set $\vec{v} = \text{PowersOf2}\, \vec{sk}$. Choose an i such that $2^i \in (\frac{q}{4}, \frac{q}{2}]$. Then set $x = \langle C_i, v \rangle$, where C_i is the $\mathrm{i}^t h$ row of C when counting from 0.
Return plaintext

$$\mu = \lfloor \frac{x}{v_i} \rceil \in \{0, 1\}$$

where v_i is the i^{th} element of $\vec{v}$ when counting from 0.

$\mathcal{D}_{sk}$(*params*, C):

Set $\vec{v} = \text{PowersOf2}\, sk$.
Let the first $l - 1$ coefficients of $\langle C \cdot \vec{v} \rangle$ be $(c_{l-1}, c_{l-2}, \cdots, c_1)$
Then recover $\mu_1 = \text{LSB}(\mu) = \lfloor \frac{c_1}{2^{l-2}} \rceil$.
Then recover the next significant bit $\mu_2 = \lfloor \frac{c_2 - 2^{l-3}}{2^{l-2}} \rceil$, and so on.
Finally, return the plaintext as

$$\mu = 2^0 \mu_1 + 2^1 \mu_2 + \cdots + 2^{l-2} \mu_{l-1}$$

Add($C_1, C_2, +$) :

For ciphertexts C_1 and C_2, output

$$C = \text{Flatten}(C_1 + C_2)$$

MultConst($C, \alpha, \cdot$) :

Let C be a ciphertext, and $\alpha \in \mathbb{Z}_q$. Set $M_\alpha := \text{Flatten}(\alpha \cdot I_n)$. Then, output

$$C = Flatten(M_\alpha \cdot C)$$

Remark 4.6 :
MultConst is not a "true" homomorphic evaluation function as defined in Definition 3.13 as one of its operands is an unencrypted number α. But it behaves in the expected manner, that is, $\mathcal{D}(MultConst(C, \alpha, \cdot)) = \mathcal{D}(C) \cdot \alpha$.

4.5 Examples

Let us now demonstrate the algorithm $\mathcal{D}$ with a small example. Note that the parameters chosen below guarantee no security, but results in small enough terms that an example may be computed by hand. Refer to Section 4.3 for choosing parameters securely.

Example 4.7 (Compute $\mathcal{D}_{sk}$(params, C)):
Say in our case, $n = 4, \implies q = 2^4 = 16$
Set $\vec{v} :=$ PowersOf2($\vec{sk}$)
Lets say the first 4 elements of $\langle C, \vec{v}\rangle$ are $[15, 13, 9, 2]$
Set $c_4 := 15$, $c_3 := 13$, $c_2 := 9$, and $c_1 := 2$.
Let us now calculate the LSB(μ) from c_1

$$c_1 = 2^{4-1} \cdot \mu + small = 8 \cdot \mu + small$$
$$\mu_1 = \lfloor \frac{c_1}{2^{4-1}} \rceil = \lfloor \frac{2}{8} \rceil = 0$$

We now recover the remaining bits of μ

$$c_2 = 2^{4-2} \cdot (2 \cdot \mu_2 + \mu_1) + small = 8\mu_2 + \mu_1 + small$$
$$\mu_2 = \lfloor \frac{c_2 - 4\mu_1}{2^{4-1}} \rceil = \lfloor \frac{9 - 0}{8} \rceil = 1$$

$$c_3 = 2^{4-3} \cdot (4\mu_3 + 2\mu_2 + \mu_1) + small = 8\mu_3 + 4\mu_2 + 2\mu_1 + small$$
$$\mu_3 = \lfloor \frac{c_3 - 4\mu_2 - 2\mu_1}{2^{4-1}} \rceil = \lfloor \frac{13 - 4 - 0}{8} \rceil = 1$$

$$c_4 = 2^{4-4} \cdot (8\mu_4 + 4\mu_3 + 2\mu_2 + \mu_1) + small = 8\mu_4 + 4\mu_3 + 2\mu_2 + \mu_1 + small$$
$$\mu_4 = \lfloor \frac{c_4 - 4\mu_3 - 2\mu_2 - \mu_1}{8} \rceil = \lfloor \frac{15 - 4 - 2 - 0}{8} \rceil = 1$$

Finally, we assemble the bits to get our plaintext

$$\mu = \mu_1 + 2\mu_2 + 4\mu_3 + 8\mu_4 = 0 + 2 + 4 + 8 = 14$$

5 Cryptanalysis

Homomorphic cryptosystems are by definition malleable. But also, their "homomorphiness" gives the entire ciphertext space additional structure. We theorise that this additional structure is enough to undermine the indistinguishability property (Definition 3.2), and it is not infeasible to leverage this with a Chosen Plaintext Attack (CPA) (Definition 3.7).

5.1 Basic Idea

In the system under consideration, all our ciphertexts are big square matrices. Imagine, if one could, to squash all square matrices of parameter n onto a single linear spectrum. Then, we think all encryptions of a plaintext (given some constant parameters for the system) form a distinct pile on this spectrum. This way, for a paramter n, we have 2^n clusters in a spectrum of 2^{N^2} width (as before, $N = (n+1) \times l$), as we have only 2^n plaintexts to encrypt, but our ciphertext space is $\{0,1\}^{N \times N}$, which has 2^{N^2} members. In particular, we theorise there must be a way to find these clusters such that they partition the cipher space cleanly.

Figure 1: The "Matrix Spectrum" : The colored parts indicate piles of distinct encryption, and the white parts indicate unused part of the space.

5.2 Data Science Facts About Our Data

DecisionTrees and RandomForests

Decision Trees (Definition 7.3) and Random Forests (Definition 7.6) are data classifiers which can ingest some data to train a certain model, and then predict which class an unknown data point belong to. This property makes these tools ideal candidates for training on known encryptions obtained in a CPA model, and then try to classify unknown ciphertexts.
For data generated by us, we have consistently been able to tune parameters to gain an accuracy of about 90% with different system parameters, and keypairs, reaching even perfect accuracy in some case.

Our Setup

In our experiments, for each set of parameters, we generate 200 encryptions of each bit, then split 70% of them into a training set and the remaining 30% into a testing set, transform our sets according to a TDA pipeline (Section 7.1.2), and finally train the model and score its accuracy against the testing set. Care was taken to ensure that the training data and the testing data do not overlap. That is, ciphertexts included in the testing set are never seen by the model during training (except if the encryption algorithm generated duplicates, which has an overwhelming probability to not happen.)

5.3 Our Results

We ran our analysis over a range of parameters selected in accordance to Section 4.3. We present the accuracy and time results of our attack below.

5.3.1 Accuracy

We include the exact analysis pipeline we used in Appendix 7.3. Both the RandomForest and DecisionTree classifiers work similarly, with slightly different performance. In the following table, we have noted the accuracy of whichever performed better in the respective case. While both the classifiers have parameters one could tune for improving the score for each instance of our cryptosystem, the following table shows the accuracy of the unchanging globally "optimal" parameters we have arrived at through experimentation.

n	q	m	χ	Accuracy
64	2^{18}	2305	$N_{Discrete}(\mu = 7.11, \sigma^2 = 1)$	79%
128	2^{18}	4609	$N_{Discrete}(\mu = 3.55, \sigma^2 = 1)$	98%
512	2^{22}	22529	$N_{Discrete}(\mu = 11.64, \sigma^2 = 1)$	86%
768	2^{22}	33793	$N_{Discrete}(\mu = 7.75, \sigma^2 = 1)$	76%

With these accuracy results in hand, we only have to worry about the feasibility of our attack.

5.3.2 Time Complexity

Recall, that no cryptosystem is completely unbreakable, as every scheme can be broken by the brute force attack, that is, simply trying every possible answer until we arrive at something which works. But it is not feasible to mount such an attack against a secure scheme, as it may take far too long to compute than is actually possible. A typically secure scheme has 128 bits of security, that is, it requires 2^{128} computations to complete the brute force attack. Assuming a reasonably fast computer can do around 3 billion computations per second, that will still take 10^{29} seconds, that is, 10^{11} $\times$ **the age of the universe** to complete. As long as we can show that our attack still performs better than this worst-case time of $\mathcal{O}(2^n)$, it is better than brute force, and is not considered infeasible.

n	q	m	χ	Security Time Guarantee	Time Taken
64	2^{18}	2305	$N_{Discrete}(\mu = 7.11, \sigma^2 = 1)$	$<$ 5 Minutes	02:00:00
128	2^{18}	4609	$N_{Discrete}(\mu = 3.55, \sigma^2 = 1)$	5 Minutes	04:15:46
512	2^{22}	22529	$N_{Discrete}(\mu = 11.64, \sigma^2 = 1)$	10^5 Years	230:31:38
768	2^{22}	33793	$N_{Discrete}(\mu = 7.75, \sigma^2 = 1)$	10^{18} Years	274:29:00

Below, we plot both the wall time of our attack on a single core of a relatively modern CPU, against the input parameter n. We refrain from multithreading our analysis as it currently results in a significant blowup in memory usage, and does not provide a proportional speedup. This restricts attacks against secure parameters. From the figure 5.3.2, it is clear that our approach (of complexity $\mathcal{O}(n^{2.18})$) is much better than the worst case of $\mathcal{O}(2^n)$.

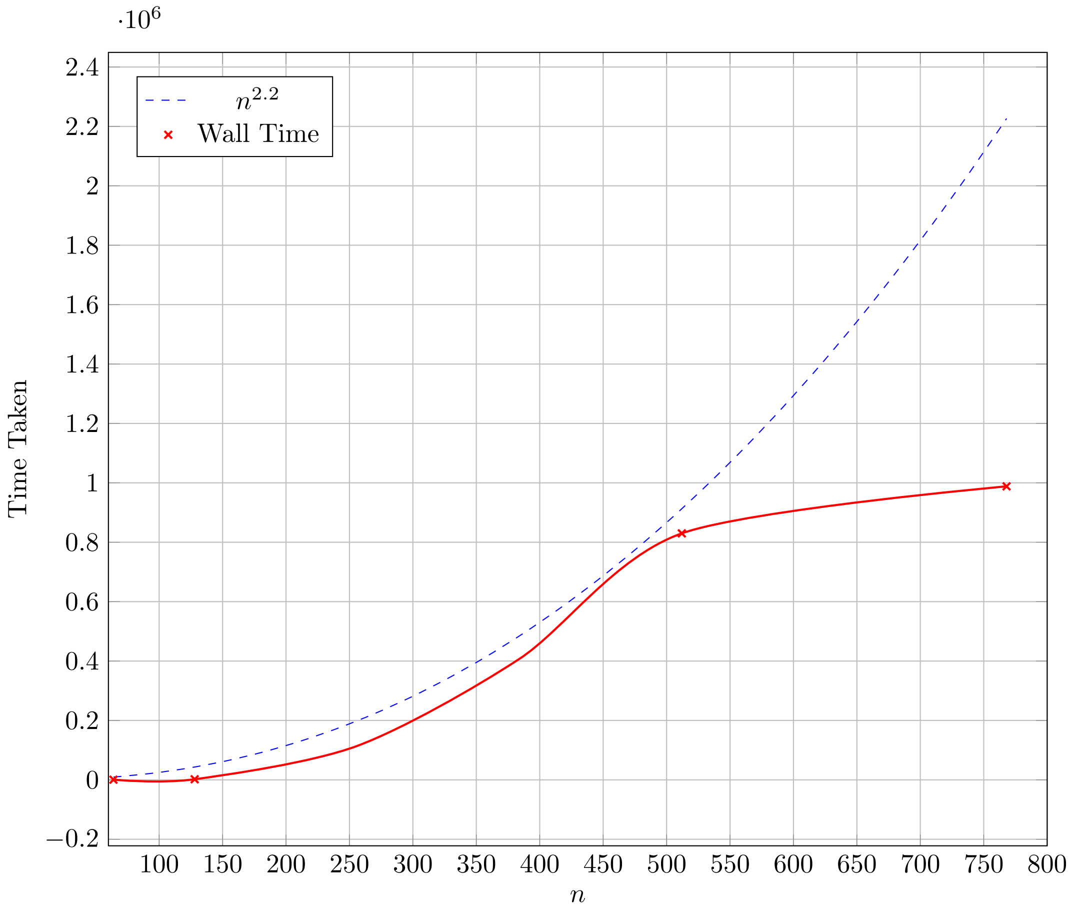

Figure 2: Time Complexity Graph

5.4 Concluding Remarks

As things appear right now, we have found a flaw in a scheme which, in theory, is supposed to be secure, depending on certain assumptions which are widely accepted to be correct. We use novel and previously unconsidered methods of attack to achieve this. Instead of classical cryptanalysis where one would try to sniff out enough leaked information out of a clever combination of ciphertexts to recover the secret key, we turn to data science for help. We completely bypass the recovery of a secret key, and use inherent structure present in this homomorphic system to find patterns in the ciphertexts, which a classical attack would perhaps never see.

We also tried this same strategy of attack against the base Learning With Errors instance on which the scheme is based, to no success.

6 Further Work

It would be interesting to investigate why this scheme is susceptible to this type of attack. As our attack is ineffective against the base LWE, and the security of this scheme depends on a proof from [Reg09], it should be possible to check whether the proof is improperly adapted in its use.

It may also be interesting to see whether other homomorphic schemes are susceptible to this class of attacks. As our attack is based in Topological Data Analysis, it may be particularly interesting to investigate this attack against Fast Fully Homomorphic Encryption over the Torus [Chi+16].

One could also look into improving the efficiency of our attack, especially in leveraging a multiprocessing attack with the same idea. At the moment, we only use the `n_jobs` parameter of the functions provided by the various data science libraries. Right now, this provides only $\sim 30\%$ decrease in time when using 32 processors as opposed to just 1.

7 Appendix

7.1 Data Scientific Background

In this section we provide some background on the data scientific concepts used in the thesis. Note that we do not aim for full mathematical precision in this section, but just enough information to add context to our work in the main body of this thesis. Much of the content in this section is sourced from [Was03].

7.1.1 Data Mining Dictionary

Statisticians and computer scientists often use different language for the same thing. Here is a dictionary that the reader may want to return to throughout this section.

Statistics	Computer Science	Meaning
Covariates	Features	The X_i's
Classifier	Hypothesis	Map $h : \mathcal{X} \to \mathcal{Y}$
Data	Training sample	$(X_1, Y_1), \ldots, (X_n, Y_n)$
Estimate	Learning	Finding a good classifier
Classification	Supervised learning	Predicting a discrete Y from X

Definition 7.1 (Classification):
The problem of predicting a discrete random variable Y from another random variable X is called classification. Classification is an instance of supervised learning.

Definition 7.2 (Classification Rule):
Consider independent and identically distributed data $(X_1, Y_1), \ldots, (X_n, Y_n)$, where

$$X_i = (X_{i1}, \ldots, X_{id}) \in \mathcal{X} \in \mathbb{R}^d$$

is a d-dimensional vector and Y_i takes values in some finite set $\mathcal{Y}$. A classification rule is a function $h : \mathcal{X} \to \mathcal{Y}$. When we observe a new X, we predict Y to be $h(X)$.

Definition 7.3 (Decision Tree Classifier):
Trees are classification methods that partition the covariate space X into disjoint pieces and then classify the observations according to which partition element they fall in. As the name implies, the classifier can be represented as a tree. The bottom nodes of the tree are called the leaves.

Example 7.4 (Decision Tree):
For illustration, suppose there are two covariates, X_1 = age and X_2 = blood pressure. Figure 7.1.1 shows a classification tree using these variables. The tree is used in the following way. If a subject has Age ≥ 50 then we classify him as $Y = 1$. If a subject has Age < 50 then we check his blood pressure. If systolic blood pressure < 100 then we classify him as $Y = 1$, otherwise we classify him as $Y = 0$.

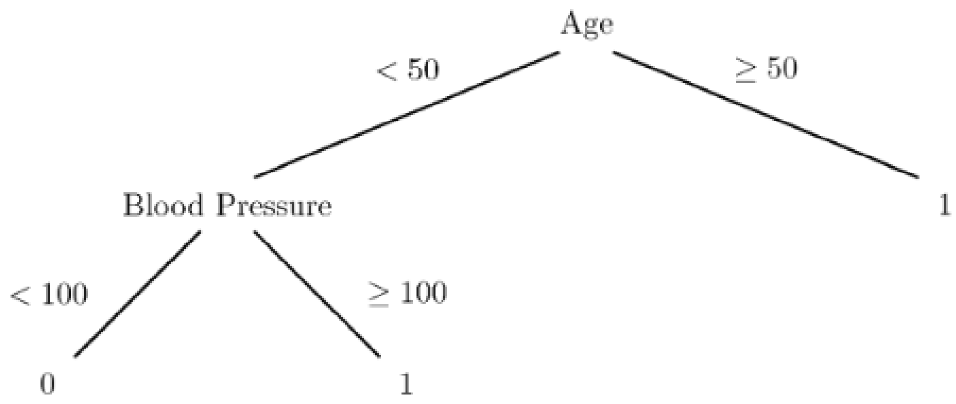

Figure 3: A simple classification tree [Was03]

Definition 7.5 (Ensemble Methods):
Ensemble methods are supervised learning methods which use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.

Definition 7.6 (Random Forest Classifier):
A Random Forest is an ensemble learning method for classification that operates by constructing a multitude of decision trees at training time. The output of the random forest is the class selected by most trees.

7.1.2 Topological Data Analysis

Topological Data Analysis (TDA) is an approach to the analysis of datasets using techniques from the field of algebraic topology. TDA is a data science tool that looks at the *shape* of data. It consists of a range of different approaches with an underlying theme of extracting topological invariants from original data. In our case, we extract topological **features** (7.1.1) from looking at our encrypted bits as **pointclouds**.

7.1.3 The Pipeline

TDA isn't a single technique. Rather, it is a collection of approaches with the common theme of extracting shapes from data. A generic TDA pipeline can be visualised as below.

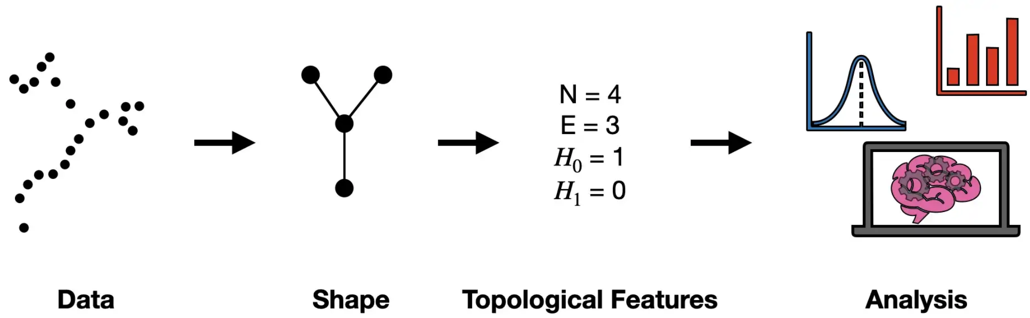

Figure 4: A generic TDA pipeline visualised [Tal22]

7.2 Implementation

We have implemented the scheme as described in Section 4.4 using `python`. Our implementation is provided below. It, along with the code in 7.3, is also made available at https://codeberg.org/aaruni96/master-thesis-code .

```python
# importing stuff
import numpy as np
import math

# initialise the RNG system
rng = np.random.default_rng()

# define a dictionary for when parameters need to change


def floorceiling(a):
    '''
    Returns the closest integer to input
    '''
    # as is standard, exact halfway point is rounded to the nearest even
    # so, 0.5 is rounded *DOWN* to 0, and -0.5 is rounded *UP* to 0.
    return np.round(a).astype(int)


def bitdecomp(a, q):
    '''
    Returns bit decomposition of input (l least significant digits)
    '''
    bda = []
    l = math.floor(math.log(q, 2)) + 1
    for i in range(0, len(a)):
        ai = int(a[i])
        assert ai >= 0, f'ai must always be positive, or overflow. ai = {ai}'
        bitstring = format(ai, f'0{l}b')[::-1]  # reversed bitdecomp of a[i]
        for j in range(0, l):
```

```
            bda.append(int(bitstring[j]))
    return bda


def mbitdecomp(a, q):
    '''
    Bit decomposition method, but for matrices
    '''
    bda = []
    for i in range(np.shape(a)[0]):
        bda.append(bitdecomp(a[i], q))
    return bda


def abitdecomp(a, q):
    '''
    Inverse bitdecomposition method
    '''
    abda = []
    j = 0
    t = 0
    l = math.floor(math.log(q, 2)) + 1
    for i in range(0, len(a)):
        t += a[i] * (2**j)
        j = (j + 1) % l
        if (j == 0):
            abda.append(t % q)
            t = 0
    return abda


def flatten(a, q):
    '''
    Computes a flattened vector of the input.
    The flattened vector is restricted to entries of only {0,1},
    but has the same product as input vector with Powersof2(b)
    '''
    return bitdecomp(abitdecomp(a, q), q)


def mflatten(a, q):
    '''
    Flatten but for matrices
    '''
    a = np.array(a)
    a = a.astype(int)
    f = []
    for i in range(0, np.shape(a)[0]):
        f.append(flatten(a[i], q))
    return f


def powersof2(a, q):
    '''
    Returns a vector composed of each input vector entry
    multiplied with upto l powers of 2
    '''
    r = []
    l = math.floor(math.log(q, 2)) + 1
    for i in range(0, len(a)):
        p = 1
        for j in range(0, l):
```

```
            r.append((a[i] * p) % q)
            p = p * 2
    return r


def setup(n):
    '''
    Setup function
    '''
    # at least 2**3 = 8 for encryption/decryption
    # at least 2**4 = 16 for homomorphic operations
    # something higher for reasonable security
    q = 2**22
    m = 2 * n * math.floor(math.log(q, 2)) + 1
    params = {'q': q, 'n': n, 'm': m}
    return params


def secretkeygen(params):
    '''
    Function to generate secret key
    '''
    sk = [1]
    for i in range(0, params['n']):
        sk.append(-1 * rng.integers(0, params['q'], endpoint=False))
    return sk


def LWE(m, q):
    '''
    Draws from ZZq such that the sum of all choices is
    less than or equal to q/16
    '''
    stdev = 1.0
    e = rng.normal(q / 16 / m, stdev, m)
    print(f'N_d(mu={q/16/m}, stddev={stdev}')
    for n in e:
        assert n >= 0, f'n must be positive, reduce your variance.\nstddev={stdev}, q={q}, , m={m}, mu={q/16/m}\nsum(e) = {sum(e)}'
    e = np.array([k % q for k in e.astype(int)])
    return e


def publickeygen(params, sk):
    '''
    Function to generate public key matrix
    '''
    B = rng.integers(
        0, params['q'],
        (params['m'], params['n']),
        endpoint=False
    )
    t = sk[1:-1]
    t.append(sk[-1])
    for i in range(0, len(t)):
        t[i] = -t[i]
    e = LWE(params['m'], params['q'])
    b = np.matmul(B, t) + e
    A = []
    for i in range(0, len(B)):
        A.append([])
        A[i].append(b[i] % params['q'])
```

```
        for j in range(0, len(B[i])):
            A[i].append(B[i][j])
    A = np.rint(A)
    A = A.astype(int)
    return A


def enc(params, pk, mu, N):
    '''
    Encryption function
    '''
    R = rng.integers(0, 1, (N, params['m']), endpoint=True)
    I_n = np.identity(N)
    A = pk
    C = mflatten(mu * I_n + mbitdecomp(np.matmul(R, A), params['q']), params['
    q'])
    return C


def dec(params, sk, C):
    '''
    Decryption function, given plaintext mu is from a small space (say {0,1})
    '''
    q = params['q']
    v = powersof2(sk, q)
    i = int(math.log(q, 2) - 1)
    x = np.inner(C[i], v) % q
    return int(floorceiling(x / v[i]))


def mpdec(params, sk, C):
    '''
    Decryption function for a general mu, given q is a power of 2
    '''
    l = math.floor(math.log(params['q'], 2)) + 1
    v = powersof2(sk, params['q'])
    C = np.array(C)
    v = np.array(v).astype(int)
    Cv = (np.inner(C, v)).astype(int)
    #print(Cv%params['q'])
    retval = 0
    bits = np.zeros(l - 1).astype(int)
    for i in range(0, l - 1):
        #4 bits of 16
        numerator = Cv[l - 2 - i] % params['q']
        #starting with mu_1
        myrealexpr = numerator
        myexpr = f'(c_{i + 1}'
        #print("calculation for bit ", i+1)
        #print("================================")
        #print("numerator is ",numerator)
        #print("bits so far are", bits)
        for j in range(0, i):
            myexpr += f' - {int(2**(l-2)/2**(j+1))}mu_{i - j}'
            myrealexpr -= int(2**(l - 2) / 2**(j + 1)) * bits[i - j - 1] %
    params['q']
        myexpr += f')/{2**(l - 2)}'
        #print(myrealexpr)
        #print((myrealexpr%q) / 2**(l-2))
        myrealexpr = floorceiling((myrealexpr % params['q']) / 2**(l - 2))
        #print(myexpr)
        #print(myrealexpr)
```

```
        #print("================================")
        bits[i] = myrealexpr
        retval = retval + 2**i * bits[i]
        retval = retval % params['q']
    return retval


def hadd(C1, C2, q):
    '''
    Homomorphic addition function
    '''
    C1 = np.array(C1)
    C2 = np.array(C2)
    C = mflatten(C1 + C2, q)
    return C


def multconst(C, alpha, q):
    return flatten(np.matmul(flatten(alpha * np.identity(np.shape(C)[0]), q),
    C), q)
```

7.3 Statistical Analysis Code

For the sake of completeness, we include the statistical analysis we ran, implemented in `python`.

```
# lets import things
# raw imports
import os
import math

# aliased imports
import pandas as pd
import numpy as np

# gtda imports
from gtda.homology import CubicalPersistence
from gtda.diagrams import Amplitude, Scaler, PersistenceEntropy
from gtda.images import RadialFiltration, HeightFiltration

# sklearn imports
from sklearn.ensemble import RandomForestClassifier
from sklearn.pipeline import make_pipeline, make_union
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

# set switches for which things to compute
do_rf = True
do_ttsplit = True
# using multiple jobs uses much more memory,
# and spends a lot of CPU time just shovelling data around
dj = 1

# parameter
num = 768

if (do_ttsplit):

    # read some data
    print("Reading data.....")
    path = './'
```

```python
    allfiles = os.listdir(path)
    allpaths = []
    enc = []
    dgm = []
    for f in allfiles:
        if f.startswith(f'enc_{num}_'):
            allpaths.append(os.path.join(f'{path}{f}'))

    print("Making data frames....")
    for f in allpaths:
        print(f"Reading {f}...")
        limbo = np.loadtxt(f, dtype=np.bool_)
        print(
            f'Size of {f} in memory is '
            f'{limbo.nbytes / 1024 /1024 /1024} GiB'
        )
        enc.append(pd.DataFrame(np.reshape(
                    limbo, (limbo.shape[0] // limbo.shape[1], limbo.shape[1]**2)
                    )))
    del limbo   # desperate attempt to free up some memory
    print("Data frames done!")

else:
    print(
        "do_ttsplit is false!"
        "Skipping reading data, and will load it from file in a bit!"
    )

# random forest and DT classifier
if (do_rf):
    # add target to PD
    print("Doing RF stuff!")
    if (do_ttsplit):
        for i in [0, 1]:
            limbo = int(allpaths[i].split('_')[2])
            enc[i] = enc[i].assign(target=np.full(enc[i].shape[0], limbo))
        del limbo   # desperate attempt to free up some memory

    # combine 0s and 1s into a single DF
        data = pd.concat([enc[0], enc[1]], ignore_index=True)
        feature_names = [c for c in data.columns if c not in ["target"]]
        X, y = np.array(data[feature_names]), np.array(data["target"])

    # setup training
        X = X.reshape((
            -1,
            int(math.sqrt(enc[0].shape[1])),
            int(math.sqrt(enc[0].shape[1]))
        ))

        train_size, test_size = int(0.70*X.shape[0]), int(0.30*X.shape[0])  # 70:30 split, as methodology mandates

    # we never use enc after this
        del enc # desperate attempt to save some memory
        X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                            test_size=test_size,
                                                            train_size=train_size,
                                                            random_state=666,
```

```
                                                              stratify=y
                                                              )
        del X, y    # desperate attempt to save memory

        # save to disk
        np.save(f'{num}_y_test', y_test)
        np.save(f'{num}_y_train', y_train)

        # tda pipeline
        print("Starting with the TDA pipeline...")

        steps = [
            ("filtration", RadialFiltration(center=np.array([20, 6]))),
            ("diagram", CubicalPersistence()),
            ("rescaling", Scaler()),
            ("amplitude", Amplitude(
                metric="heat",
                metric_params={'sigma': 0.15, 'n_bins': 60}
            ))
        ]

        print("Done enumerating steps!")

        direction_list = [[-1, 1]]
        center_list = [[13, 13]]

# Creating a list of all filtration transformer, we will be applying
        filtration_list = (
            [
                HeightFiltration(direction=np.array(direction), n_jobs=dj)
                for direction in direction_list
            ] + [
                RadialFiltration(center=np.array(center), n_jobs=dj)
                for center in center_list
            ]
        )
        print("Done creating filteration list!")

# Creating the diagram generation pipeline
        diagram_steps = [
            [
                filtration,
                CubicalPersistence(n_jobs=dj),
                Scaler(n_jobs=dj),
            ]
            for filtration in filtration_list
        ]
        print("Done setting up diagram steps!")

# Listing all metrics we want to use to extract diagram amplitudes
        metric_list = [
            {
                "metric": "bottleneck",
                "metric_params": {}
            },
            {
                "metric": "wasserstein",
                "metric_params": {"p": 1}
            },
            {
                "metric": "wasserstein",
                "metric_params": {"p": 2}
```

```
            },
            {
                "metric": "landscape",
                "metric_params": {"p": 1, "n_layers": 1, "n_bins": 100}
            },
            {
                "metric": "landscape",
                "metric_params": {"p": 1, "n_layers": 2, "n_bins": 100}
            },
            {
                "metric": "landscape",
                "metric_params": {"p": 2, "n_layers": 1, "n_bins": 100}
            },
            {
                "metric": "landscape",
                "metric_params": {"p": 2, "n_layers": 2, "n_bins": 100}
            },
            {
                "metric": "betti",
                "metric_params": {"p": 1, "n_bins": 100}
            },
            {
                "metric": "betti",
                "metric_params": {"p": 2, "n_bins": 100}
            },
            {
                "metric": "heat",
                "metric_params": {"p": 1, "sigma": 1.6, "n_bins": 100}
            },
            {
                "metric": "heat",
                "metric_params": {"p": 1, "sigma": 3.2, "n_bins": 100}
            },
            {
                "metric": "heat",
                "metric_params": {"p": 2, "sigma": 1.6, "n_bins": 100}
            },
            {
                "metric": "heat",
                "metric_params": {"p": 2, "sigma": 3.2, "n_bins": 100}
            },
        ]

        print("Done enumerating metrics!")

        feature_union = make_union(
            *[PersistenceEntropy(nan_fill_value=-1)] + [
                Amplitude(**metric, n_jobs=dj) for metric in metric_list
            ]
        )

        print("Made feature_union!")

        tda_union = make_union(
            *[make_pipeline(*diagram_step, feature_union)
              for diagram_step in diagram_steps],
            n_jobs=dj
        )
        del feature_union, metric_list, diagram_steps

        print("Done tda_union")

```

```
        print(" Will begin tda fit next!")
        print('Transforming training data')

        X_train_tda = tda_union.fit_transform(
            np.reshape(
                X_train[0], (1, X_train[0].shape[0], X_train[0].shape[1])
            )
        )
        X_train[0] = 0
        for i in range(1, len(X_train)):
            print(f'{i+1} of {len(X_train)}')
            X_train_tda = np.append(
                X_train_tda,
                tda_union.fit_transform(
                    np.reshape(
                        X_train[i],
                        (1, X_train[i].shape[0], X_train[i].shape[1])
                    )
                ),
                axis=0
            )
            X_train[i] = 0
        np.save(f'{num}_train_data', X_train_tda)
        print("Done fit_transform!")
        print("Transforming testing data")
        X_test_tda = tda_union.transform(np.reshape(
            X_test[0],
            (1, X_test[0].shape[0], X_test[0].shape[1])
        ))
        X_test[0] = 0

        for i in range(1, len(X_test)):
            print(f'{i+1} of {len(X_test)}')
            X_test_tda = np.append(
                X_test_tda,
                tda_union.transform(np.reshape(
                    X_test[i],
                    (1, X_test[i].shape[0], X_test[i].shape[1])
                )),
                axis=0
            )
            X_test[i] = 0
        np.save(f'{num}_test_data', X_test_tda)
        print("Done transform!")
        print("TDA pipeline done!")

    else:
        print("Loading test-train data pair from disk...")
        X_train_tda = np.load(f'{num}_train_data.npy')
        X_test_tda = np.load(f'{num}_test_data.npy')
        y_train = np.load(f'{num}_y_train.npy')
        y_test = np.load(f'{num}_y_test.npy')
        print("Done loading data!")

    # random forest
    # grid search for parameters
    print("Fitting for random forest....")
    maxrfsc = 0
    for md in range(2, 10):
        print(f'{md}')
        for ne in range(2, 50):
            for msp in range(2, 50):
```

```
                rf = RandomForestClassifier(
                    max_depth=md,
                    n_estimators=ne,
                    min_samples_split=msp,
                    random_state=666
                )
                rf.fit(X_train_tda, y_train)
                rfscore = rf.score(X_test_tda, y_test)
                if rfscore > maxrfsc:
                    maxrfsc = rfscore
                    rfretup = (maxrfsc, md, ne, msp)
                if (rfscore > 0.8):
                    print(rfscore, md, ne, msp)

    print("Random forest done!")

    # decision tree
    # grid search for parameters
    print("Fitting for DT....")
    maxdtsc = 0
    for md in range(2, 100):
        for msp in range(2, 50):
            rf = DecisionTreeClassifier(
                max_depth=md,
                min_samples_split=msp,
                random_state=666
            )
            rf.fit(X_train_tda, y_train)
            dtscore = rf.score(X_test_tda, y_test)
            if (dtscore > maxdtsc):
                maxdtsc = dtscore
                dtretup = (dtscore, md, msp)
            if (dtscore > 0.8):
                print(dtscore, md, msp)
    print("DecisionTree done!")

    print(f'random forest score is {rfretup}')

    print(f'decision tree score is {dtretup}')
```

References

[RAD78] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. “On Data Banks and Privacy Homomorphisms”. In: *Massachusetts Institute of Technology* (1978). URL: http://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf.

[Sin00] Simon Singh. *The Code Book : The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Knopf Doubleday Publishing Group, 2000.

[Was03] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference (Springer Texts in Statistics)*. Dec. 2003. ISBN: 0387402721.

[Gen09] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[Reg09] Oded Regev. “On lattices, learning with errors, random linear codes, and cryptography”. In: *Journal of the ACM (JACM)* 56.6 (2009), pp. 1–40.

[Mic11] Daniele Micciancio. “Shortest Vector Problem”. In: *Encyclopedia of Cryptography and Security*. Ed. by Henk C. A. van Tilborg and Sushil Jajodia. Boston, MA: Springer US, 2011, pp. 1196–1197. ISBN: 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_434. URL: https://doi.org/10.1007/978-1-4419-5906-5_434.

[Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[MP12] Daniele Micciancio and Chris Peikert. “Trapdoors for Lattices: Simpler, Tighter, Faster, Smaller”. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 700–718. ISBN: 978-3-642-29011-4.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: (2013). URL: https://eprint.iacr.org/2013/340.

[DM14] Léo Ducas and Daniele Micciancio. *FHEW: Bootstrapping Homomorphic Encryption in less than a second*. Cryptology ePrint Archive, Paper 2014/816. https://eprint.iacr.org/2014/816. 2014. URL: https://eprint.iacr.org/2014/816.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. *On the concrete hardness of Learning with Errors*. Cryptology ePrint Archive, Paper 2015/046. https://eprint.iacr.org/2015/046. 2015. URL: https://eprint.iacr.org/2015/046.

[Chi+16] Ilaria Chillotti et al. *TFHE: Fast Fully Homomorphic Encryption Library*. 2016. URL: https://tfhe.github.io/tfhe/.

[Che+17] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Cham: Springer International Publishing, 2017, pp. 409–437. ISBN: 978-3-319-70694-8.

[TSB18] Christopher Tralie, Nathaniel Saul, and Rann Bar-On. "Ripser.py: A Lean Persistent Homology Library for Python". In: *The Journal of Open Source Software* 3.29 (Sept. 2018). DOI: 10.21105/joss.00925. URL: https://doi.org/10.21105/joss.00925.

[Tau+20] Guillaume Tauzin et al. *giotto-tda: A Topological Data Analysis Toolkit for Machine Learning and Data Exploration*. 2020. arXiv: 2004.02551 [cs.LG].

[Tal22] Shawhin Talebi. "Topological Data Analysis (TDA) A less mathematical introduction". In: (2022). URL: https://towardsdatascience.com/topological-data-analysis-tda-b7f9b770c951.

[KKP23] Jayati Kaushik, Aaruni Kaushik, and Upasana Parashar. *Using Topological Data Analysis to Classify Encrypted Bits*. 2023. DOI: 10.48550/ARXIV.2301.07393. URL: https://arxiv.org/abs/2301.07393.