Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Privacy Preserving String Search using Homomorphic Encryption

Master Thesis

Aurel Feer

March 21, 2024

Advisors: Prof. Dr. Kenny Paterson, Dr. Zichen Gui

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Privacy Preserving String Search is a mechanism that allows searching for a short string (the pattern) in a longer string (the text), where both the pattern and text remain private from the party performing the searching computation. Cloud-based computing services are becoming increasingly popular and along with them, cryptographic privacy preserving computation primitives gain greater importance, as they offer tools to outsource computation to the cloud while still maintaining privacy.

In this thesis, we introduce a formal syntax and security notion for Privacy Preserving String Search. We then develop two new Privacy Preserving String Search schemes that use Homomorphic Encryption as building block. We prove both schemes secure according to our new security notion. Using our new schemes, we can privately search for a pattern of any length in a text of length $32'000$ in less than 60ms on an average laptop.

In order to improve the practicality of the schemes, we propose a new compression method for ciphertexts, resulting in lower storage and network complexity.

Acknowledgements

Thank you to Dr. Zichen Gui for supervising this Master Thesis and for continuously providing great new inputs and ideas to enhance the project.

Thank you to Prof. Dr. Kenny Paterson for giving valuable guidance during the project and helping to put it into broader context.

Contents

Acknowledgements **iii**

Contents **v**

1 Introduction **1**

2 Preliminaries **5**
2.1 Notation 5
2.2 String Search 5
2.2.1 Privacy Preserving String Search 7
2.3 Homomorphic Encryption 7
2.3.1 Fully Homomorphic Encryption 8
2.3.2 CKKS 8
2.4 Rotating Vectors 12
2.5 Discrete Fourier Transform 13

3 Related Work **15**
3.1 String Search Based on Searchable Encryption 15
3.2 String Search Based on Homomorphic Encryption 16

4 Constructing Privacy Preserving String Search Schemes **19**
4.1 Syntax for Privacy Preserving String Search Schemes 19
4.2 Security Definition for Privacy Preserving String Search Schemes 20
4.3 Fourier Private Search 21
4.3.1 Applying the Fourier Transform 24
4.3.2 Applying the HE Layer 25
4.4 Randomized Fourier Private Search 26
4.4.1 Variations of RFPS 30
4.5 Features and Limitations 31
4.5.1 Comparison of Schemes 32
4.6 Ciphertext Compression 32

4.7 Error Analysis . . . 34
4.7.1 Encryption Error . . . 35
4.7.2 Compression Error . . . 37
4.7.3 Numerical and Encoding Error . . . 38
4.7.4 Accumulation of Errors . . . 40
4.7.5 False Positives in RFPS . . . 42
4.8 Security Analysis . . . 44
4.8.1 Security Proof for FPS . . . 45
4.8.2 Security Proof for RFPS . . . 47

5 Implementation **51**
5.1 Parameter choice . . . 51
5.2 Results . . . 52

6 Future Work **57**
6.1 Wildcard String Search . . . 57
6.2 Improve Compression . . . 58
6.3 Using other Homomorphic Encryption Primitives . . . 58
6.3.1 Using linearly HE schemes . . . 58
6.3.2 Using boolean HE schemes . . . 60
6.3.3 Using other RLWE-based HE schemes . . . 60
6.4 Privacy Preserving String Search as Building Block . . . 60

7 Conclusion **63**

Bibliography **65**

Chapter 1

Introduction

String Search. The string search problem consists of finding all occurrences of a short string (pattern) in a longer string (text). It has been extensively studied and optimized due to its ubiquity. Use cases might include web search engines, genome sequence search, database search and many more.

Cloud Computing. Cloud storage and cloud computing services have experienced a rise in popularity in recent years. Such services allow a client to outsource storage and computation of data to a cloud service provider (CSP) (i.e. Amazon Web Services, Microsoft Azure, Google Cloud Platform and many more) which is in charge of managing the underlying infrastructure of the data center. This has several advantages for the client:

- The client does not need to acquire and maintain its own physical infrastructure or hire dedicated personnel.
- Scalability is easier to achieve by simply renting more resources on demand.
- In many cases, data availability is better, as backup solutions are provided in case of hardware failures.
- Devices with hardware constraints such as phones or internet of things devices can utilize the computation and storage that the cloud servers provide.

In the case of the string search problem, a client might want to take advantage of cloud infrastructure for string search, while still ensuring (partial) confidentiality of either the pattern, the text, or both! For example, if the client has a lot of texts that do not fit in local device storage, it can store the texts on a server and simply search it privately directly on the server side.

Privacy Preserving String Search. Privacy preserving string search (PPSS) schemes aim to enable string search queries on texts while maintaining specific privacy guarantees. State of the art PPSS schemes vary significantly in their practicality and their privacy guarantees. Privacy is usually measured using leakage functions, whereas practicality can be measured by metrics such as storage cost, network cost and computation cost.

A plethora of constructions that provide string search functionality in a privacy preserving way already exists. Unfortunately, most of them have critical shortcomings that prevent them from being used in the real world. String search schemes based on searchable encryption (SE) exhibit complicated leakage functions. This kind of leakage is unpredictable and the actual information that is exposed is difficult to quantify. This makes SE-based schemes prone to attacks that recover information about the plaintexts that was originally intended to remain private. On the other end of the spectrum, there are string search schemes based on Homomorphic encryption (HE). It is not uncommon for HE-based string search schemes to leak only the length of the searched pattern as well as the length of the text. It is a lot easier to reason about the things an adversary might do with this kind of limited information. Unfortunately, current HE comes at high computational and storage cost and this cost is inherited by string search schemes that use HE. Ciphertexts are usually many times larger than their underlying plaintext and searching for a pattern in a text can take hours. We aim to keep the excellent privacy properties of HE-based schemes, but improve on computation and storage cost.

New Schemes. In this thesis we propose two new PPSS schemes, called *Fourier Private Search* (FPS) and *Randomized Fourier Private Search* (RFPS). Both schemes have very minimal leakage, while improving on practicality considerations when compared to the state of the art. We hope to make a step forward towards the viability of privacy preserving string search in real world applications. These schemes could also be used as subroutines to build more complex protocols. For example, the practicality gains could open the doors for completely new cryptographic primitives such as Substring Private Information Retrieval.

On a high level, FPS is modelled after a well-known plaintext string search algorithm that uses the Discrete Fourier Transform to accelerate computation. The algorithm lends itself well to Single Instruction Multiple Data (SIMD) computation. We use the SIMD-capability of the CKKS fully homomorphic encryption scheme to efficiently compute the plaintext algorithm in a privacy preserving way.

RFPS is a randomized adaptation of FPS. The homomorphic computation is significantly more efficient than the deterministic FPS, but as a trade-off, it

might identify false positive matches.

PPSS Syntax and Security. We introduce a formal syntax for PPSS schemes and define the semantic security under a chosen text and patterns attack (SS-CTPA) notion. We then prove that FPS and RFPS are SS-CTPA-secure.

In [23], Mainardi et. al. introduce syntax and security notions for protocols called Privacy Preserving Substring Search (also PPSS for short). We point out that our definition of PPSS is more general than the one in [23] and FPS or RFPS would not fit the syntax outlined by [23].

Compression. FPS and RFPS achieve the privacy targets we set ourselves, but they still leave potential for improvement when it comes to practicality. In particular, the ciphertext sizes and therefore the communication complexity is quite large. Ciphertexts are many times larger than the underlying plaintext was. This ciphertext expansion also applies to the query response, which contains the matching results for the client. In terms of storage size, the response is larger than the original plaintext.

To combat this, we devise a lossy compression method for CKKS ciphertexts where we throw away the least significant bits of the polynomial coefficients. Depending on encryption parameters, we manage to compress the matching response ciphertext by roughly $20-50\%$ while still maintaining correctness.

Error Bounds. At multiple steps in the FPS and RFPS pipelines, there are errors introduced, affecting the underlying raw data. The errors stem from different sources such as the lossy ciphertext compression or the encryption noise that is required for maintaining security. We analyze these error sources and bound their impact on correctness.

Implementation. To verify the practicality of FPS and RFPS we implement them in C++ and test performance. Searching for a pattern in a text of length $32'000$ merely takes 60ms in FPS and 40ms in RFPS. Both schemes involve very little preprocessing during setup. Setup takes 40ms in FPS and 25ms in RFPS for a text of the same length.

Chapter 2

Preliminaries

2.1 Notation

Table 2.1 provides and overview of symbols and notation used in this document. We explain each notation in more detail as it is introduced.

Logarithm. $\log(\cdot)$ denotes the 2nd logarithm.

Vectors. The i-th element of a n-dimensional vector x is denoted x_i. The first element of the vector has index $i = 0$, the last element has index $i = n - 1$. We allow negative values and values greater than $n - 1$ in the subscript. In that case, we may omit the $(\mod n)$ operation in the interest of legibility.

$$x_i = x_{i \mod n}$$

Vector Slices. For two indices i and j and a vector $x = (x_0, \ldots, x_{n-1})$, we denote the $(j - i + 1)$-dimensional vector slice as $x_{i:j}$.

$$x_{i:j} = (x_i, x_{i+1} \ldots, x_j)$$

2.2 String Search

In the string search problem, given a text and a pattern, we try to find all the indices where the pattern occurs in the text.

More formally, we are given an alphabet Σ, a text T of length n and a pattern P of length m.

$$\begin{aligned}
\Sigma &= \{0, 1, \ldots, |\Sigma| - 1\} \\
T &= (T_0, T_1, \ldots, T_{n-1}) \in \Sigma^n \\
P &= (P_0, P_1, \ldots, P_{m-1}) \in \Sigma^m
\end{aligned}$$

Symbol	Description	Notes
Σ	Alphabet	$\Sigma = \{0, 1, \ldots, \lvert\Sigma\rvert - 1\}$
n	Text length	
m	Pattern length	
T	Searchable Text	$T = (T_0, \ldots, T_{n-1})$
P	Pattern to be searched	$P = (P_0, \ldots, P_{m-1})$
R	Random vector	$R = (R_0, \ldots, R_{m-1})$
χ_R	RFPS Distribution	
M	Matching vector	$M = (M_0, \ldots, M_{n-m+1})$
$\mathcal{I}$	Indices where P occurs in T	
t	Matching Threshold	
$\mathcal{F}(\cdot)$	Discrete Fourier Transform	
$\mathrm{Enc}_k(\cdot)$	Encryption	
$\mathrm{Dec}_k(\cdot)$	Decryption	
$\mathrm{Ecd}(\cdot)$	Encoding	
$\mathrm{Dcd}(\cdot)$	Decoding	
$\mathrm{E}_k(\cdot)$	Encode, then encrypt	$\mathrm{E}_k(\cdot) = \mathrm{Enc}_k(\mathrm{Ecd}(\cdot))$
$\mathrm{D}_k(\cdot)$	Decrypt, then decode	$\mathrm{D}_k(\cdot) = \mathrm{Dcd}(\mathrm{Dec}_k(\cdot))$
$\mathcal{DN}(\sigma^2)$	Discrete normal distribution	
$\mathcal{T}$	Ternary distribution	$\mathcal{T} \sim \text{uniform}\{-1, 0, 1\}$
N	Polynomial modulus degree	
$\mathcal{R}$	Integer Polynomial Ring	$\mathcal{R} = \mathbb{Z}[X]/(\Phi_{2N}(X))$
$\sigma(\cdot)$	Canonical Embedding	$\sigma : \mathcal{R}_q \mapsto \mathbb{C}^N$
$\lVert\cdot\rVert_\infty^{\text{can}}$	Canonical infinity norm	$\lVert a\rVert_\infty^{\text{can}} = \lVert\sigma(a)\rVert_\infty$
$\pi(\cdot)$	Embedding projection	
$\odot$	Element wise product	
$*$	Circular convolution	
$\mathcal{S}$	PPSS Scheme	
ε	HE Scheme	

Table 2.1: Notation used in this document

The alphabet may originally be any set of size $|\Sigma|$ containing letters, symbols, digits and more. We use an encoding function to represent the alphabet as integers. For example, the UTF-8 encoding maps characters to the integer set $\{0, 1, \ldots, 255\}$.

The text T is a vector of n characters, all of which are an element of the alphabet Σ. Analogously, the pattern P is a vector of m characters, all of which are an element of Σ. We assume that $n \geq m$.

We say that P *occurs* in T at index i if and only if $T_{i:i+m-1} = P$. The output of the string search is the set $\mathcal{I}$ which contains all indices where P occurs in T. There are $n - m + 1$ possible indices where P could occur in T.

$$\mathcal{I} = \{i : P \text{ occurs in } T \text{ at index } i\}$$

In this document, we may also say that there is a *match* at index i if P occurs in T at index i. We use `Search()` to denote a plain text algorithm that solves

the string search problem.

$$\mathcal{I} = \texttt{Search}(T, P)$$

In literature, other terms such as *pattern matching*, *substring search* or *substring matching* may be used to refer to the string search problem or variations thereof.

2.2.1 Privacy Preserving String Search

The schemes presented in this thesis attempt to efficiently solve the Privacy Preserving String Search (PPSS) problem. Intuitively, it is similar to the conventional version of string search, but the owner of the text and pattern wants to keep as much as possible about them secret while outsourcing computation and storage to a second party.

Leakage. As is common with many multi-party privacy preserving computation schemes, we use leakage functions, denoted $\mathcal{L}$, to identify what each party learns about the input. In this case, we are mainly interested in the leakage to the server, as the client owns all input data and thus trivially knows everything about it.

2.3 Homomorphic Encryption

Homomorphic Encryption (HE) describes encryption primitives designed to enable computation on encrypted data without knowing the secret key. HE schemes differ widely in their capabilities and practicality. Here we categorize them and briefly give a few examples.

- **Linearly and Multiplicative HE.** Schemes falling within category encrypt numbers and permit a limited set of operations to be executed on the ciphertexts. Often, the depth and type of functions that can be evaluated is heavily constrained. The most common types of operation are addition and multiplication.
  - RSA [25]: Arbitrary number of modular multiplications.
  - BGN [4]: Arbitrary number of additions, but only a single multiplication.
- **Boolean HE** Schemes in this category operate over boolean plaintexts.
  - Goldwasser–Micali [15]: Arbitrary number of exclusive-or operations.
- **LWE and RLWE-based HE** Schemes in this category are relatively recent. They rely on the hardness of the Learning With Errors (LWE)

[24] or the Ring Learning With Errors (RLWE) [22] problems, which are both considered secure against attacks by quantum computers. Encryption involves applying some kind of random error to the original plaintext. LWE- and RLWE-based schemes enable the user to encrypt entire vectors rather than single scalar values. The user can then perform operations on these ciphertexts in a Single Instruction Multiple Data (SIMD) manner. For example, it can add two n-dimensional integer vectors rather than performing n additions of integers.

– BFV [13, 6]: Arithmetic operations over vectors of integers.

– CKKS [10]: Arithmetic operations over vectors of complex numbers. More details can be found in subsection 2.3.2.

2.3.1 Fully Homomorphic Encryption

Fully Homorphic Encryption (FHE) describes a more powerful version HE. The crucial difference here is, that while HE schemes only allow the user to evaluate functions of a certain fixed depth (i.e. just one multiplication), FHE schemes enable the user to evaluate functions of arbitrary depth. This is commonly achieved using a technique called bootstrapping, proposed by [14].

Computational Cost. While bootstrapping certainly makes FHE schemes a lot more powerful than HE schemes, it comes at a high computational cost. This unfortunately makes most current FHE schemes impractical to use in real world applications. Nevertheless, since the original introduction of FHE in 2009, schemes have improved drastically to the point, that people are using FHE to develop complex applications such as privacy preserving machine learning algorithms [27, 19].

2.3.2 CKKS

CKKS is a FHE scheme named after the authors Cheon et. al. who introduced the scheme in [10]. It provides homomorphic arithmetic operations over vectors of complex numbers. The two new string search schemes presented in this document were designed with the capabilities of CKKS in mind. Here we will highlight a few features and explain the parts of CKKS' construction that are especially relevant in more detail.

Parameters. CKKS relies on the hardness assumption of the *Ring Learning With Errors* (RLWE) [22] problem and is (mostly) parameterized by a polynomial modulus degree N and a coefficient modulus chain $(q_L, q_{L-1}, \dots, q_0)$. N must be a power of 2.

Security. CKKS achieves indistinguishability under chosen plaintext attacks (IND-CPA) security. It is worth noting however, that in [21], Li et. al. demonstrate a passive key recovery attack on CKKS. The authors argue, that in the case of approximate HE schemes (such as CKKS), the traditional IND-CPA security notion is insufficient to capture security. They propose a new, stricter security notion that covers security against this attack.

Standardization. First of all, we remark that CKKS is not yet standardized. Hence, we generally follow the suggestions made by the authors of the original paper [10], as well as the choices made by the authors of the Microsoft SEAL library [26].

The Homomorphic Encryption Standard [2] provides baseline security estimations for the RLWE problem. In particular, it recommends RLWE parameters that should be used in order to achieve a specific security level (i.e. 128 bit security). The Microsoft SEAL library chooses parameters based on this standard.

Spaces. CKKS operates using a Message Space, a Plaintext Space and a Ciphertext Space and defines encoding and encryption functions to map between them.

- **Message Space.** The message space is the set of $\frac{N}{2}$-dimensional complex vectors $\mathbb{C}^{\frac{N}{2}}$. It is exposed to the user as the set of messages it can homomorphically encrypt and perform operations on.
- **Plaintext Space.** The plaintext space $\mathcal{R}$ is the set of polynomials with integer coefficients modulo the $(2N)$-th cyclotomic polynomial $\Phi_{2N}(X) = X^N + 1$. We denote it $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. It is used as internal representation of plaintext messages.
- **Ciphertext Space.** The ciphertext space $\mathcal{R}_q^2$ is the set of polynomial pairs $\mathcal{R}^2$, but with the integer coefficients reduced modulo q. We denote it $\mathcal{R}_q^2 = (\mathbb{Z}_q[X]/(X^N + 1))^2$. It is used as representation of encrypted messages.

Canonical Embedding. The encoding and decoding processes make heavy use of the canonical embedding of $\mathcal{R}$ in $\mathbb{C}^N$, denoted σ.

$$\sigma : \mathcal{R} \mapsto \mathbb{C}^N$$
$$\sigma(p(x)) = (p(\zeta_N^1), p(\zeta_N^3), \dots, p(\zeta_N^{2N-1}))$$

ζ_N denotes the N-th root of unity $e^{\frac{2\pi i}{N}}$. In other words, the canonical embedding of a polynomial $p(x)$ evaluates $p(x)$ at N different points, where the

j-th point is the $(2j+1)$-th power of the root of unity ζ_N. Given an embedding $z \in \mathbb{C}^N$, the inverse σ^{-1} of the canonical embedding finds a polynomial $p'(x)$ such that $p'(\zeta_N^{2j+1}) = z_j$.

We use the canonical embedding to define a canonical infinity norm $\|\cdot\|_\infty^{\text{can}}$ over $\mathcal{R}$.

$$\|p(x)\|_\infty^{\text{can}} = \|\sigma(p(x))\|_\infty$$

This will prove useful later as we estimate error bounds.

Projecting Embeddings. For any $p(x) \in \mathcal{R}$, the canonical embedding $z = \sigma(p(x))$ is conjugate symmetric, meaning $z_k = \overline{z_{-k}}$, where $\bar{\cdot}$ denotes the complex conjugate. Therefore, the embedding is uniquely defined by $\frac{N}{2}$ complex numbers. $\pi(z)$ projects z to $\mathbb{C}^{\frac{N}{2}}$ and $\pi^{-1}(m)$ expands a message to $\mathbb{C}^N$.

Encoding and Decoding. The encoding and decoding functions $\text{Ecd}(\cdot)$ and $\text{Dcd}(\cdot)$ map elements from the message space to the plaintext space and vice versa. During the encoding process, messages are multiplied by a scale Δ.

- $\text{Dcd}(p(x), \Delta)$: Given a plaintext $p(x) \in \mathcal{R}$ and a scale $\Delta \in \mathbb{N}$, return $\pi(\sigma(\Delta^{-1} \cdot p(x)))$.
- $\text{Ecd}(m, \Delta)$: Given a message $m \in \mathbb{C}^{\frac{N}{2}}$ and a scale $\Delta \in \mathbb{N}$, return $\sigma^{-1}(\lfloor \Delta \cdot \pi^{-1}(m) \rceil_{\sigma(\mathcal{R})})$.

Key Generation. CKKS uses a public key k_p for encryption and a secret key k_s for decryption. Further keys contained in the evaluation key k_{ev} are needed to perform certain homomorphic operations, but we do not explain these in detail. k_s and k_p are generated as follows.

- k_s: Sample s from χ_s. Return $k_s = (1, s)$.
- k_p: Sample polynomial a from χ_a. Sample an error polynomial e from χ_e. Return $k_p = (-a \cdot s + e, a)$.

Encryption and Decryption.

- $\text{Enc}_{k_p}(p)$: Sample v from χ_v. Sample two error polynomials e_0, e_1 both from χ_e. Return $(c_0, c_1) = k_p \cdot v + (p + e_0, e_1) \mod q_L$.
- $\text{Dec}_{k_s}((c_0, c_1))$: Return $\langle (c_0, c_1), k_s \rangle = c_0 + c_1 \cdot s \mod q_l$.

The security of CKKS relies in part on the error polynomials that are added to the ciphertexts during encryption. We can now also explain why the encoding function multiplies the messages by a scale Δ. By scaling, the magnitude of the message relative to the error increases. During decoding, the message magnitude is normalized again.

Figure 2.1 [1] provides a overview of the three spaces and the functions that map between them. Encoding and encryption are often chained together.

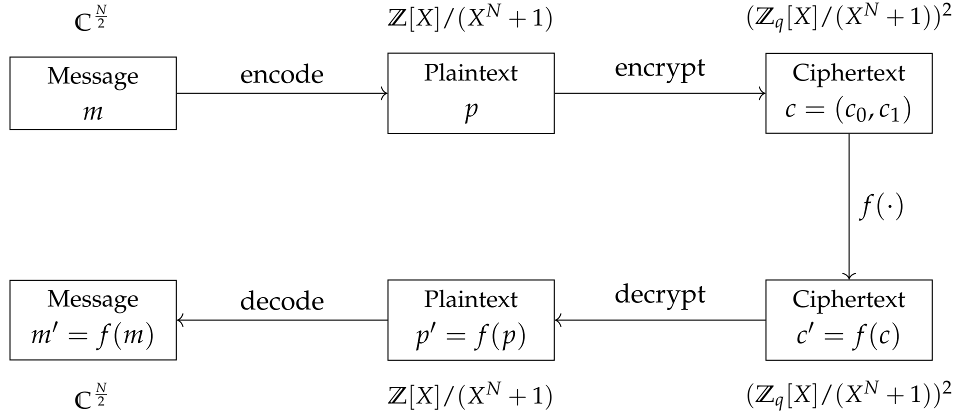

Figure 2.1: Overview over the CKKS pipeline for computing a function $f(\cdot)$

As a shorthand, we use $\mathrm{E}_{k_p}(\cdot)$ to denote the encoding function followed by the encryption function. Analogously, $\mathrm{D}_{k_s}(\cdot)$ is a shorthand for decryption followed by decoding.

$$\mathrm{E}_{k_p}(m) = \mathrm{Enc}_{k_p}(\mathrm{Ecd}(m))$$
$$\mathrm{D}_{k_s}(c) = \mathrm{Dcd}(\mathrm{Dec}_{k_s}(c))$$

Distributions An important part of CKKS standardization is choosing the distributions $\chi_e, \chi_a, \chi_s, \chi_v$ for key generation and encryption. For the rest of this document we use the following distributions over $\mathcal{R}$.

- χ_e draws each coefficient independently at random from the discrete Gaussian distribution $\mathcal{DN}(\sigma^2)$. We use $\sigma = 3.2$ as standard deviation.
- χ_a draws each coefficient uniformly and independently at random from $\mathbb{Z}_{q_L}$.
- χ_s draws each coefficient independently at random from the ternary distribution $\mathcal{T} \sim \mathrm{uniform}\{-1, 0, 1\}$.
- χ_v draws each coefficient independently at random from the ternary distribution $\mathcal{T} \sim \mathrm{uniform}\{-1, 0, 1\}$.

This choice affects the amount of encryption noise as well as for the security of CKKS.

[1] Source: https://blog.openmined.org/ckks-explained-part-1-simple-encoding-and-decoding/, Accessed on: 20.03.2024

Ciphertext Tags. For analysis purposes, we inherit the notation for tagged ciphertexts that was introduced by Chen et. al. in [10]. The idea is, that for each ciphertext c, we also keep track of publicly known meta-information about this ciphertext. We write these tags as a tuple C.

$$C = (c, l, v, B)$$

$c \in \mathcal{R}_{q_l}$ encrypts a plaintext p. $l \in \{0, \ldots, L\}$ denotes the level of the ciphertext. $v \in \mathbb{R}$ provides an upper bound on the encrypted plaintext. $v \geq \|p\|_\infty^{\text{can}}$. Similarly $B \in \mathbb{R}$ provides an upper bound on the error that is present in the ciphertext. $B \geq \|e\|_\infty^{\text{can}}$ where e is the error that has accumulated on c during encryptions and calculations.

Homomorphic Operations. The original paper by Chen et. al. introduced a set of different operations that could be performed on ciphertexts without knowing the secret key. This set has since been expanded by a few new operations. The following operations are relevant for this document.

- **Homomorphic Addition and Subtraction.** Given two ciphertexts c_a and c_b, that encrypt vectors a and b respectively, we can compute a ciphertext c_{a+b} that encrypts the sum $a + b$. Analogously, we can compute c_{a-b}.

- **Homomorphic Multiplication.** Given two ciphertexts c_a and c_b, that encrypt vectors a and b respectively, we can compute a ciphertext $c_{a \odot b}$ that encrypts the element-wise product $a \odot b$.

- **Plaintext Multiplication.** If we are given a publicly known plaintext $p \in \mathcal{R}$, we can create a valid transparent ciphertext $c_{\text{plain}} = (p, 0) \in \mathcal{R}_q^2$ that encrypts p without knowing either the secret or public key. Knowing this, we can use c_{plain} as a generic ciphertext for multiplication.

2.4 Rotating Vectors

The rotation operation $\text{rot}(x, k)$ rotates the elements in a vector x by k dimensions. x may be from an arbitrary vector space $\mathbb{V}^N$.

$$\begin{aligned} \text{rot} : \mathbb{V}^N \times \mathbb{Z} &\mapsto \mathbb{V}^N \\ \text{rot}((x_0, \ldots, x_{N-1}), k) &= (x'_0, \ldots, x'_{N-1}) \\ x'_i &= x_{i-k \mod N} \end{aligned}$$

2.5 Discrete Fourier Transform

The string search schemes introduced in this thesis make use of the Discrete Fourier Transform (DFT), denoted $\mathcal{F}$. It is defined as follows:

$$\mathcal{F} : \mathbb{C}^N \mapsto \mathbb{C}^N$$
$$\mathcal{F}((x_0, x_1, \ldots, x_{N-1})) = (X_0, X_1, \ldots, X_{N-1})$$
$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i \frac{k}{N} n}$$

The Discrete Fourier Transform has an inverse (iDFT), denoted $\mathcal{F}^{-1}$.

$$\mathcal{F}^{-1}((X_0, X_1, \ldots, X_{N-1})) = (x_0, x_1, \ldots, x_{N-1})$$
$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n \cdot e^{2\pi i \frac{k}{N} n}$$

In particular, the schemes use the following well known properties.

- **Linearity** For any two vectors $x, y \in \mathbb{C}^N$ and any two numbers $a, b \in \mathbb{C}$, it holds that
$$\mathcal{F}(ax + by) = a\mathcal{F}(x) + b\mathcal{F}(y)$$

- **Rotation Property** Let $x \in \mathbb{C}^N$ and m be an integer. Given $\mathcal{F}(x)$, we can compute $\mathcal{F}(\mathrm{rot}(m, x))$ as follows:
$$\mathcal{F}(\mathrm{rot}(x, m))_k = \mathcal{F}(x)_k \cdot e^{-2\pi i \frac{k}{N} m}$$
In other words, if we construct a rotation vector $r^{(m)} \in \mathbb{C}^N$ with $r^{(m)}_k = e^{-2\pi i \frac{k}{N} m}$. Then:
$$\mathcal{F}(\mathrm{rot}(x, m)) = \mathcal{F}(x) \odot r^{(m)}$$

- **Convolution Property** The circular convolution of two vectors $x, y \in \mathbb{C}^N$, denoted $x * y \in \mathbb{C}^N$ is defined as follows:
$$(x * y)_k = \sum_{i=0}^{N-1} x_i y_{(k-i) \mod N}$$
Using $\mathcal{F}$, the circular convolution can be computed using the element wise product.
$$\mathcal{F}(x * y) = \mathcal{F}(x) \odot \mathcal{F}(y)$$

- **Symmetry Property** For a real valued input $x \in \mathbb{R}^N$, the Discrete Fourier Transform of x is symmetric.
$$\mathcal{F}(x)_k = \overline{\mathcal{F}(x)_{-k \mod N}}$$

where $\bar{\cdot}$ denotes the complex conjugate.

We observe that because of the symmetry property, a input of N real values is uniquely determined by the first $\lfloor \frac{N}{2} \rfloor + 1$ values of its transform. In this document we will therefore also use the shorthand

$$\mathcal{F} : \mathbb{R}^N \mapsto \mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1}$$
$$\mathcal{F}^{-1} : \mathbb{C}^{\lfloor \frac{N}{2} \rfloor + 1} \mapsto \mathbb{R}^N$$

Chapter 3

Related Work

Privacy preserving string search has naturally been a popular topic of study in recent years. We identify two main approaches to the problem.

- **Searchable Encryption** This approach aims to use conventional (usually symmetric) encryption schemes as building blocks to construct new Searchable Encryption (SE) schemes. It is sometimes also known as *Structured Encryption* or *Searchable Symmetric Encryption*.
- **Homomorphic Encryption** This approach aims to use the relatively new (F)HE schemes as building blocks to construct schemes that enable string search.

3.1 String Search Based on Searchable Encryption

Design Philosophy. The Searchable Encryption (SE) approach uses conventional encryption primitives to build string search schemes [9, 20], or even more capable ones that allow queries such as range queries on the text [12]. Characteristically, SE-based PPSS schemes perform a large amount of preprocessing of the text during the setup phase to construct an encrypted indexing data structure that can be searched. For example, the schemes in [23, 20] employ the Burrows Wheeler Transform (BWT) [8] as preprocessing step whereas the scheme in [9] uses encrypted suffix trees to quickly find substring matches.

Efficiency. The most efficient SE-based constructions use well established and optimized symmetric schemes as building blocks. As a result, SE-based PPSS schemes tend to be computationally efficient. In [12], Faber et. al. implement a working prototype that can perform queries on a Terabyte-scale database in the matter of seconds. In [17], Hahn et al. search a database containing 10'000 indexed emails in 98.3ms. In [23], preprocessing a 40MB

genome string takes approximately 20 seconds. It then takes approximately 5 minutes to search for a pattern of length 6 in the genome.

High Leakage. While the SE-based string search schemes can be very efficient in terms of storage overhead and computation speed, their leakage profiles tend to be quite substantial. This has led to theoretical and practical attacks that manage to recover parts of the plaintext or the queries [16]. For example, [17] presents an attack that manages to recover between 1% and 15% of the plaintext. Furthermore, the complicated nature of the leakage functions for SE-based schemes may be undesirable, as it is not immediately obvious to what extent they can be exploited. New attacks might surface in the future, that exploit the leakage profiles in unforeseen ways.

3.2 String Search Based on Homomorphic Encryption

Schemes in this category make use of some form of Homomorphic Encryption (HE) in order to perform computations directly on ciphertexts. HE opens up new possibilities that would not be available using symmetric schemes. The type of HE used can vary. For example, in [28], the authors devise their own Somewhat Homomorphic Encryption scheme that is specifically designed to compute the Hamming distance between two bit-vectors. This approach potentially enables additional use in applications where the Hamming distance between text and pattern is relevant, rather than exact matches. In [11], the authors merely require a linearly homomorphic encryption scheme as a building block.

The authors of [5] propose a randomized string search scheme using a RLWE-based HE scheme. They construct a HE circuit where the multiplicative depth does not depend on the input text or pattern, but rather on public encryption parameters. Searching for a pattern of length 100 in a UTF-32 text of length 1′080 takes 629 seconds and has a false positive probability of just 2^{-65}. Their randomized construction has inspired the randomized version of the PPSS scheme that we introduce in this document.

Leakage. HE-based PPSS(-like) schemes tend to have very minimal leakage functions. In many cases the information leaked to the server is limited to the text length n and the pattern length m [28, 18, 5]. In [11], the authors take special care to not leak the length of the pattern, as this kind of leakage is especially problematic in the setting of human genome search.

Computation and Storage cost. State of the art HE schemes are inefficient when compared to conventional encryption schemes, impacting both computation efficiency as well as ciphertext sizes. Consequently, PPSS(-like)

schemes that use these HE primitives also suffer from these inefficiencies to some degree. Computation cost for encryption, processing and decryption results should be carefully considered when constructing a PPSS scheme. Remember, that the goal is to outsource the string search computation to a server, so the encryption and decryption cost for a client should ideally be cheaper than performing the string search computation locally. Ciphertext size is important because it directly translates to networking and storage cost.

In [11], an encrypted human genome string of length $3 \cdot 10^9$ takes up 100GB of storage in encrypted form using EC-ElGamal. Furthermore, encrypting such a genome string takes 115 hours. It is noteworthy, that once all preprocessing and data transfer is done, searching for a pattern of length $1'000$ in the entire genome string takes just 0.68ms.

In [5] encrypting a $1'080$ character text results in a ciphertext of size 930 KB. This corresponds to an storage cost of 882 Bytes per UTF-32 character.

Chapter 4

Constructing Privacy Preserving String Search Schemes

In this chapter, we introduce a new syntax and security notion for PPSS, then we introduce our constructions for two new PPSS schemes called FPS and RFPS. Based on the characteristics of these new schemes, we propose a ciphertext compression method that reduces network complexity. We then identify and quantify the error sources that might affect correctness and prove that FPS and RFPS are secure according to the new security notion.

4.1 Syntax for Privacy Preserving String Search Schemes

A PPSS scheme $\mathcal{S}$ with security parameter 1^λ and public parameters $\textsc{pub}$ is defined by the algorithms $\mathcal{S} = (\texttt{KGen}, \texttt{Setup}, \texttt{Query})$.

- $k_s, k_p, k_{\text{ev}} \leftarrow \texttt{KGen}(1^\lambda, \textsc{pub})$ is a probabilistic algorithm run by the client. It takes a security parameter 1^λ, some public parameters $\textsc{pub}$ and returns a secret key k_s, a public key k_p and a evaluation key k_{ev}.
- $(\perp, \textsc{tdata}) \leftarrow [\texttt{Setup}_{\textsf{Clt}}(k_s, k_p, T, \textsc{pub}), \texttt{Setup}_{\textsf{Srv}}(k_{\text{ev}}, \textsc{pub})]$ is an interactive algorithm between the client and the server. $\texttt{Setup}_{\textsf{Clt}}$ takes as input a secret key k_s, a public key k_p, a text T and public parameters $\textsc{pub}$. $\texttt{Setup}_{\textsf{Srv}}$ takes as input a evaluation key k_{ev} and public parameters $\textsc{pub}$. After the algorithm, the server outputs some encrypted text data $\textsc{tdata}$.
- $(\mathcal{I}, \perp) \leftarrow [\texttt{Query}_{\textsf{Clt}}(k_s, k_p, P, \textsc{pub}), \texttt{Query}_{\textsf{Srv}}(k_{\text{ev}}, \textsc{tdata}, \textsc{pub})]$ is an interactive algorithm between the client and the server. $\texttt{Query}_{\textsf{Clt}}$ takes as input a secret key k_s, a public key k_{ev}, a pattern P and public parameters $\textsc{pub}$. $\texttt{Query}_{\textsf{Srv}}$ takes as input a evaluation key k_{ev}, some text data $\textsc{tdata}$ and public parameters $\textsc{pub}$. After the algorithm, the client outputs the matching indices $\mathcal{I}$.

The separation of keys into k_s, k_p, k_{ev} is common practice with HE schemes, but might not make sense in a PPSS scheme that does not rely on a HE primitive. In that case, k_p and k_{ev} could be omitted.

Correctness. We say a PPSS scheme $\mathcal{S}$ is (n, m)-correct, if given security parameter $\lambda \in \mathbb{N}$ and public parameters PUB, for all keys k_s, k_p, k_{ev} output by $\texttt{KGen}(1^\lambda, \text{PUB})$, for all texts T of length at most n, for all TDATA output by the server after running $[\texttt{Setup}_{\textbf{Clt}}(k_s, k_p, T, \text{PUB}), \texttt{Setup}_{\textbf{Srv}}(k_{\text{ev}}, \text{PUB})]$, for all patterns P of length at most m and for all $\mathcal{I}'$ output by the client after running $[\texttt{Query}_{\textbf{Clt}}(k_s, k_p, P, \text{PUB}), \texttt{Query}_{\textbf{Srv}}(k_{\text{ev}}, \text{TDATA}, \text{PUB})]$, we have that $\mathcal{I}' = \texttt{Search}(T, P)$.

Analogously we say that $\mathcal{S}$ is *partially* (n, m)-correct if the output $\mathcal{I}'$ is a super set of $\mathcal{I} = \texttt{Search}(T, P)$. In other words, for partial correctness, we allow false positive indices in $\mathcal{I}'$, but no false negative indices.

4.2 Security Definition for Privacy Preserving String Search Schemes

For this model, we consider the server to be an honest-but-curious adversary. This means that the server correctly follows the protocol laid out by the scheme. However, it tries to learn more about the underlying text and pattern than what is allowed by the leakage function. We assume the client to be completely honest, as it knows everything about P and T to begin with.

The adversary is allowed to choose a text T and make as many adaptive queries for patterns $P^{(i)}$ as it likes. In the end, the adversary tries to distinguish whether or not it has been interacting with a real client, or an ideal simulator. We call this security notion *semantic security under a chosen text and patterns attack*, or adaptive SS-CTPA for short.

Ideal and Real Security Games. Algorithm 1 and Algorithm 2 define two security games. $\texttt{Setup}_{\textbf{Srv}}$ and $\texttt{Query}_{\textbf{Srv}}$ are executed by the adversary and hence it learns the corresponding inputs and communications. $\mathcal{L}$ denotes the leakage function of the PPSS scheme. It is made up of two parts $\mathcal{L} = (\mathcal{L}_{\texttt{Setup}}, \mathcal{L}_{\texttt{Query}})$.

- $\mathcal{L}_{\texttt{Setup}} : \Sigma^* \mapsto \{0,1\}^*$ defines the information about the text that the server is allowed to learn during the setup algorithm.
- $\mathcal{L}_{\texttt{Query}} : \Sigma^* \times \Sigma^* \mapsto \{0,1\}^*$ defines the information about the text and pattern that the server is allowed to learn during the query algorithm.

We say a PPSS scheme $\mathcal{S}$ with security parameter 1^λ is adaptively SS-CTPA secure with leakage $\mathcal{L}$ if for every probabilistic polynomial time adversary

Algorithm 1: $\mathrm{Real}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A}}(1^\lambda, \textsc{pub})$

$T \leftarrow \mathcal{A}(1^\lambda, \textsc{pub})$
$k_s, k_p, k_{\mathrm{ev}} \leftarrow \mathtt{KGen}(1^\lambda, \textsc{pub})$
$(\bot, \textsc{tdata}) \leftarrow [\mathtt{Setup}_{\mathbf{Clt}}(k_s, k_p, T, \textsc{pub}), \mathtt{Setup}_{\mathbf{Srv}}(k_{\mathrm{ev}}, \textsc{pub})]$
$i \leftarrow 0$
while $P^{(i)} \leftarrow \mathcal{A}()$ **do**
  $(\mathcal{I}_i, \bot) \leftarrow [\mathtt{Query}_{\mathbf{Clt}}(k_s, k_p, P^{(i)}, \textsc{pub}), \mathtt{Query}_{\mathbf{Srv}}(k_{\mathrm{ev}}, \textsc{tdata}, \textsc{pub})]$
  $i \leftarrow i + 1$
end
$b \leftarrow \mathcal{A}()$
return b

Algorithm 2: $\mathrm{Ideal}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub})$

$T \leftarrow \mathcal{A}(1^\lambda, \textsc{pub})$
$k_s, k_p, k_{\mathrm{ev}} \leftarrow \mathtt{KGen}(1^\lambda, \textsc{pub})$
$(\bot, \textsc{tdata}) \leftarrow [\mathrm{Sim}_{\mathtt{Setup}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\mathtt{Setup}}(T)), \mathtt{Setup}_{\mathbf{Srv}}(k_{\mathrm{ev}}, \textsc{pub})]$
$i \leftarrow 0$
while $P^{(i)} \leftarrow \mathcal{A}()$ **do**
  $(\mathcal{I}_i, \bot) \leftarrow$
   $[\mathrm{Sim}_{\mathtt{Query}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\mathtt{Query}}(T, P^{(i)})), \mathtt{Query}_{\mathbf{Srv}}(k_{\mathrm{ev}}, \textsc{tdata}, \textsc{pub})]$
  $i \leftarrow i + 1$
end
$b \leftarrow \mathcal{A}()$
return b

$\mathcal{A}$ there exists a simulator Sim such that $\mathcal{A}$s advantage $\mathrm{Adv}^{\text{SS-CTPA}}_{\mathcal{S}}(\mathcal{A})$ is negligible.

$$\mathrm{Adv}^{\text{SS-CTPA}}_{\mathcal{S}}(\mathcal{A}) = \left| \Pr[\mathrm{Real}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A}}(1^\lambda, \textsc{pub}) = 1] - \Pr[\mathrm{Ideal}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1] \right| \leq \mathrm{negl}(\lambda)$$

4.3 Fourier Private Search

With security and syntax for PPSS schemes formalized, we present our first new scheme, called Fourier Private Search (FPS). FPS is modelled after a well-known plaintext string search algorithm that uses DFT to accelerate computation. The core idea is to homomorphically compute a matching vector M, that contains all the information about matches.

Underlying HE primitive. We use CKKS as the underlying HE primitive in FPS. In particular, we require the following properties of CKKS.

- **Message Space.** Let N be power of 2. The message space is a set of $\frac{N}{2}$-dimensional complex vectors, $\mathbb{C}^{\frac{N}{2}}$. Ciphertext size is constant, regardless of the encrypted vector.
- **Homomorphic Operations.**
  - **Addition and Subtraction.** Given two ciphertexts c_a and c_b, that encrypt vectors a and b respectively, we can compute a ciphertext c_{a+b} that encrypts the sum $a + b$. Analogously, we can compute c_{a-b}.
  - **Element-wise Multiplication.** Given two ciphertexts c_a and c_b, that encrypt vectors a and b respectively, we can compute a ciphertext $c_{a \odot b}$ that encrypts the element-wise product $a \odot b$.
  - **Plaintext Multiplication.** Given a ciphertext c_a and a vector b, we can compute a ciphertext $c_{a \odot b}$ that encrypts the element-wise product $a \odot b$.

Defining the Matching Vector. In FPS, given a text T and pattern P, to compute a matching vector M.

$$M = (M_0, \dots, M_{n-m}) \tag{4.1}$$

$$M_i = \sum_{j=0}^{m-1} (t_{i+j} - p_j)^2 \tag{4.2}$$

We think of M_i as the matching value for T and P at index i. This definition has the very nice property, that M_i evaluates to 0, if and only if there is a occurrence of P in T at index i.

$$M_i \begin{cases} = 0 & \text{if } P \text{ occurs in } T \text{ at index } i \\ > 0 & \text{otherwise} \end{cases}$$

Thus, we can identify all occurrences of P in T by merely looking at the indices of the 0-values in M.

Computing the Matching Vector. Now, we just need an efficient way to compute the matching vector, given the homomorphic operations that CKKS provides. There is of course the naive way of directly computing each element of M using its definition, but doing so would result in a very inefficient circuit with non-constant additive depth.

Using DFT and a bit of preprocessing, we devise a much more elegant algorithm. If we rearrange equation (4.2), we can split it up into three terms S1, S2 and S3 that can be independently computed.

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j)^2 = \underbrace{\sum_{j=0}^{m-1} T_{i+j}^2}_{\text{S1}} + \underbrace{\sum_{j=0}^{m-1} P_j^2}_{\text{S2}} \underbrace{-2 \sum_{j=0}^{m-1} T_{i+j} P_j}_{\text{S3}} \tag{4.3}$$

Remember, that we want to compute not just a single element M_i, but rather an entire vector $M = (M_0, \dots, M_{n-m})$. Therefore, if we can obtain three vectors corresponding to terms S1, S2 and S3 respectively, we can subsequently determine M by summing these three vectors.

Computing S2. We start with S2, as it is the simplest. Notice, that S2 is completely independent of i and T and can be computed given P alone. Since the client knows P when it makes a query, it computes the value of S2 in plaintext and encrypts it before sending it as part of the query.

$$P^* = (P_0^*, ..., P_n^*)$$
$$P_i^* = \underbrace{\sum_{j=0}^{m} P_j^2}_{\text{S2}}$$

Computing S1. S1 is a bit more tricky, as it depends on i as well as T. The idea is to precompute a version of T called T^*, that significantly simplifies the computation of S1.

$$T^* = (T_0^*, T_1^*, \dots, T_n^*)$$
$$T_i^* = \sum_{j=0}^{i} T_j^2$$
$$T_n^* = 0$$

We can calculate S1 using elements of T^*

$$\underbrace{\sum_{j=0}^{m-1} T_{i+j}^2}_{\text{S1}} = \sum_{j=0}^{i+m-1} T_j^2 - \sum_{j=0}^{i-1} T_j^2 = T_{i+m-1}^* - T_{i-1}^*$$

Remember, that we need M_i for each $i \in \{0, \dots, n - m + 1\}$, so naturally, we also need S1 for each i. Given T^* and a pattern length m, we compute S1

for each i using a single vector rotation (see section 2.4) and a single vector subtraction.

$$T^{*(m)} = T^* - \mathrm{rot}(T^*, m)$$

By definition of T^* and $\mathrm{rot}(\cdot)$, we have the following.

$$T^{*(m)}_{i+m-1} = T^*_{i+m-1} - T^*_{i-1} = \underbrace{\sum_{j=0}^{m-1} T^2_{i+j}}_{\text{S1}}$$

So, $T^{*(m)}_{i+m-1}$ is exactly the S1 term in equation (4.3).

Computing S3. Looking at the S3 term $\sum_{j=0}^{m-1} T_{i+j}P_j$, we observe, that this sum looks quite a bit like a convolution. In fact, if we derive the n-dimensional vector $\widehat{P}$ from P,

$$\widehat{P} = (P_{m-1}, P_{m-2}, \ldots, P_1, P_0, 0, \ldots, 0) \in \Sigma^{n+1}$$

then the convolution $T * \widehat{P}$ is exactly what we want.

$$\begin{aligned}(T * \widehat{P})_{i+m-1} &= \sum_{j=0}^{n-1} T_j \widehat{P}_{i+m-1-j} \\ &= \sum_{j=0}^{n-1} T_{i+j} \widehat{P}_{m-1-j} \\ &= \underbrace{\sum_{j=0}^{m-1} T_{i+j} P_j}_{\text{S3}}\end{aligned}$$

4.3.1 Applying the Fourier Transform

We have seen how to define three $(n+1)$-dimensional vectors S1, S2 and S3 such that M can be calculated by simple vector addition.

$$M = (\underbrace{T^{*(m)}}_{\text{S1}} + \underbrace{P^*}_{\text{S2}} - 2(\underbrace{T * \widehat{P}}_{\text{S3}}))_{m-1:n-1} \tag{4.4}$$

The task is now to compute these vectors as efficiently as possible, given the tools that CKKS provides. The good thing about S1 and S2 is, that they depend only on T, m and P respectively. Hence, S1 and S2 can (partially) be precomputed before encryption. The tricky part, then, is to compute S3 because it depends on both, T and P. As we have seen, computing the S3 values can be done using the convolution of T and $\widehat{P}$. Unfortunately, CKKS has no operation to directly compute the convolution of two vectors. It does,

however have the capability to efficiently compute the element-wise product of two encrypted vectors.

This is where the DFT comes in. Using the convolution property of the DFT (section 2.5), we transform the convolution operation into an element wise multiplication. All we need is the Fourier transformed versions of T and $\widehat{P}$.

$$\mathcal{F}(T) \odot \mathcal{F}(\widehat{P}) = \mathcal{F}(T * \widehat{P})$$

We have seen how to obtain $\mathcal{F}(T * \widehat{P})$ in a single multiplication rather than trying to obtain $T * \widehat{P}$ naively using a non-constant depth circuit. The problem is, that we have the S3 vector in Fourier space, whereas the S1 and S2 vectors are not. The key insight here is to bring S1 and S2 to Fourier space as well and exploit the linearity of the DFT.

$$\begin{aligned}
\mathcal{F}(M) &= \mathcal{F}(T^{*(m)} + P^* - 2(T * \widehat{P})) \\
&= \mathcal{F}(T^{*(m)}) + \mathcal{F}(P^*) - 2\mathcal{F}(T * \widehat{P}) \\
&= \mathcal{F}(T^{*(m)}) + \mathcal{F}(P^*) - 2\mathcal{F}(T) \odot \mathcal{F}(\widehat{P}) \\
&= \mathcal{F}(T^* - \mathrm{rot}(T^*, m)) + \mathcal{F}(P^*) - 2\mathcal{F}(T) \odot \mathcal{F}(\widehat{P}) \\
&= \mathcal{F}(T^*) - \mathcal{F}(\mathrm{rot}(T^*, m)) + \mathcal{F}(P^*) - 2\mathcal{F}(T) \odot \mathcal{F}(\widehat{P}) \\
&= \underbrace{\mathcal{F}(T^*) - \mathcal{F}(T^*) \odot r^{(m)}}_{\text{S1}} + \underbrace{\mathcal{F}(P^*)}_{\text{S2}} - 2\underbrace{\mathcal{F}(T) \odot \mathcal{F}(\widehat{P})}_{\text{S3}}
\end{aligned} \tag{4.5}$$

On line 6 of equation (4.5), we used the rotation property of the DFT (see section 2.5).

4.3.2 Applying the HE Layer

In equation (4.5) we have derived an equation for $\mathcal{F}(M)$ that uses addition, subtraction, element-wise multiplication and plaintext element-wise multiplication. These are all operations that are supported by CKKS.

$$\begin{aligned}
\mathrm{E}_{k_p}(\mathcal{F}(M)) = &\underbrace{\mathrm{E}_{k_p}(\mathcal{F}(T^*)) - \mathrm{E}_{k_p}(\mathcal{F}(T^*)) \odot r^{(m)}}_{\text{S1}} \\
&+ \underbrace{\mathrm{E}_{k_p}(\mathcal{F}(P^*))}_{\text{S2}} - \underbrace{2\,\mathrm{E}_{k_p}(\mathcal{F}(T)) \odot \mathrm{E}_{k_p}(\mathcal{F}(\widehat{P}))}_{\text{S3}}
\end{aligned} \tag{4.6}$$

We left the scalar multiplication by 2 in the equation for clarity, even though it is technically not a CKKS-supported operation. However, there are many ways to emulate this doubling operation, for example using addition.

The last step is now, to put it all together into a protocol. Figure 4.1 provides an overview over the complete FPS scheme. Algorithm 3 shows a pseudocode implementation of the FPS scheme.

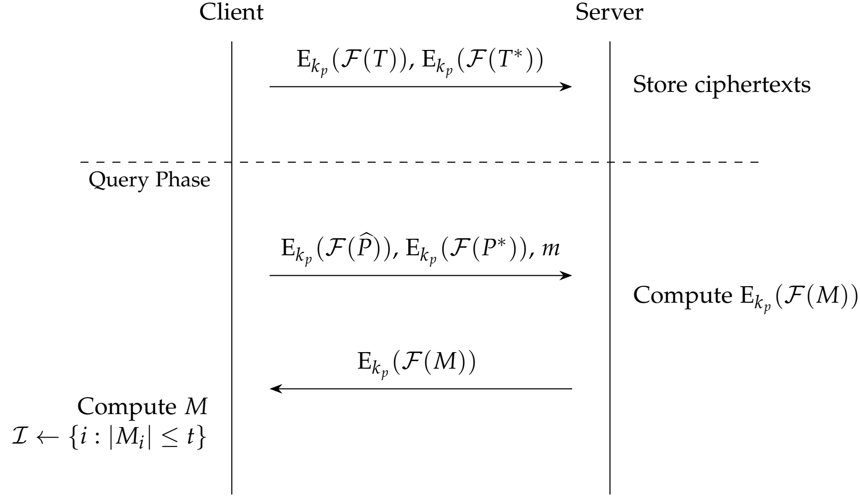

Figure 4.1: High level overview over the FPS scheme

- Given T, the client precomputes T^* and applies $\mathcal{F}$ to T and T^* to obtain $\mathcal{F}(T)$ and $\mathcal{F}(T^*)$ respectively. It then encrypts $\mathcal{F}(T)$ and $\mathcal{F}(T^*)$ and sends c_T, c_{T^*} to the server.
- Given P, the client precomputes $\widehat{P}$ and P^*. It then applies $\mathcal{F}$ to obtain $\mathcal{F}(\widehat{P})$ and $\mathcal{F}(P^*)$. It then encrypts $\mathcal{F}(\widehat{P})$ and $\mathcal{F}(P^*)$ and sends $c_{\widehat{P}}, c_{P^*}, m$ to the server.
- Given $c_T, c_{T^*}, c_{\widehat{P}}, c_{P^*}, m$, the server computes $c_{T^{*(m)}}$ and $c_{T*\widehat{P}}$ as intermediate results. It then computes c_M according to equation (4.6) and sends it to the client.
- Given c_M, the client decrypts c_M to obtain $\mathcal{F}(M)$. It then applies $\mathcal{F}^{-1}$ to obtain M. From M it then extracts the set of matching indices $\mathcal{I}$ by including all i in $\mathcal{I}$ where $|M_i| \leq t$ for some threshold $t \in \mathbb{R}$.

Note that we extract the index set $\mathcal{I}$ from M by comparing the M_i values to a threshold t rather than checking if they are equal to 0. This might seem redundant, since we saw in the definition of M that M_i is exactly 0 if there is a match. However, this thresholding process is necessary, as the ciphertext c_M that the client receives from the server, contains an inexact version of M. This is is due to several error sources such as encryption noise and numerical imprecision. For more on this, see section 4.7.

4.4 Randomized Fourier Private Search

We now present Randomized Fourier Private Search (RFPS), which is a probabilistic variant of FPS. RFPS features an even smaller leakage profile and less computation cost than FPS, but as a trade-off, the computation result is

Algorithm 3: FPS Pseudocode

Input : $\textsc{pub} = (N, (q_L, \ldots, q_0), P, h, \chi_e, \chi_a, \chi_s, \chi_v, \Delta, t, \chi_R, \Sigma)$

Fn $\texttt{KGen}(1^\lambda, \textsc{pub})$:
- $(k_s, k_p, k_{\text{ev}}) \leftarrow \texttt{KGen}^{\texttt{CKKS}}(1^\lambda, \textsc{pub})$
- Send k_{ev} to the server.

Fn $\texttt{Setup}_{\mathit{Clt}}(k_s, k_p, T, \textsc{pub})$:
- $c_T \leftarrow \text{E}_{k_p}(\mathcal{F}(T), \textsc{pub})$
- $c_{T^*} \leftarrow \text{E}_{k_p}(\mathcal{F}(T^*), \textsc{pub})$
- Send (c_T, c_{T^*}) to the server.

Fn $\texttt{Setup}_{\mathit{Srv}}(k_{ev}, \textsc{pub})$:
- Store (c_T, c_{T^*}) for later use.

Fn $\texttt{Query}_{\mathit{Clt}}(k_s, k_p, P, \textsc{pub})$:
- $c_{\widehat{P}} \leftarrow \text{E}_{k_p}(\mathcal{F}(\widehat{P}), \textsc{pub})$
- $c_{P^*} \leftarrow \text{E}_{k_p}(\mathcal{F}(P^*), \textsc{pub})$
- Send $(c_{\widehat{P}}, c_{P^*}, m)$ to the server.
- Receive c_M from the server.
- $M \leftarrow \mathcal{F}^{-1}(\text{D}_{k_s}(c_M), \textsc{pub})$
- $\mathcal{I} \leftarrow \{i : |M_i| \leq t\}$

Fn $\texttt{Query}_{\mathit{Srv}}(k_{ev}, \textsc{tdata}, \textsc{pub})$:
- Receive $(c_{\widehat{P}}, c_{P^*}, m)$ from the client.
- $c_{T^{*(m)}} \leftarrow c_{T^*} - (c_{T^*} \odot r^{(m)})$
- $c_{T\widehat{P}} \leftarrow c_T \odot c_{\widehat{P}}$
- $c_M \leftarrow c_{T^{*(m)}} + c_{P^*} - 2c_{T\widehat{P}}$
- Send c_M to the client.

only partially correct, as there can be false positives. We introduce a new vector $R = (R_0, \ldots, R_{m-1})$. R is sampled anew from the random distribution χ_R by the client each time it makes a query.

Defining the Randomized Matching Vector. By modifying the formula in equation 4.2 using R, we obtain a new definition of M.

$$M = (M_0, \ldots, M_{n-m}) \tag{4.7}$$

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j) R_j \tag{4.8}$$

It is easy to see that M_i is zero if there is a match at index i.

Theorem 4.1 *If P occurs in T at index i, then $M_i = 0$.*

Proof

$$\begin{aligned} T_{i:i+m-1} = P &\implies \forall j \in \{0, \ldots, m-1\} : T_{i+j} - P_j = 0 \\ &\implies M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j) * R_j = 0 \end{aligned}$$

□

False Positives We have seen that $M_i = 0$ if there is a match at index i. However, the reverse implication is not necessarily true. We could have an unlucky combination of T, P and R such that $T_{i:i+m-1} \neq P$ but $M_i = 0$. As an example, consider $T = (2, 0, 4)$, $P = (0, 1, 3)$ and $R = (2, 5, 1)$. Obviously, P does not occur in T at all, yet $M_0 = (2-0)2 + (0-1)5 + (4-3)1 = 0$. In this case the algorithm would incorrectly identify 0 as an index where there is a match. Fortunately we can tweak the probability of such a false positive (FP) case by choosing a suitable distribution χ_R from which R is sampled. For a probability estimation, refer to subsection 4.7.5.

Computing the Randomized Matching Vector. The recipe for homomorphically computing M is actually similar to how we computed M in the FPS scheme. We split up the formula for M_i into two terms S1 and S2.

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j) R_j = \underbrace{\sum_{j=0}^{m-1} T_{i+j} R_j}_{\text{S1}} - \underbrace{\sum_{j=0}^{m-1} P_j R_j}_{\text{S2}}$$

Computing S2. S2 exclusively depends on R and P which are both known by the client during the query phase. The client can precompute a vector PR of repeating elements.

$$\begin{aligned} PR &= (PR_0, \ldots, PR_{n-1}) \\ PR_i &= \sum_{j=0}^{m-1} P_j R_j \end{aligned}$$

Computing S1. Computing S1 is analogous to term S3 from the FPS computation. We derive a n-dimensional vector $\widehat{R}$ from R.

$$\widehat{R} = (R_{m-1}, R_{m-2}, \ldots, R_1, R_0, 0, \ldots, 0) \in \Sigma^n$$

The convolution of T with $\widehat{R}$ contains exactly the elements needed for S1.

$$\begin{aligned}(T * \widehat{R})_{i+m-1} &= \sum_{j=0}^{n-1} T_j \widehat{R}_{i+m-1-j} \\ &= \sum_{j=0}^{n-1} T_{i+j} \widehat{R}_{m-1-j} \\ &= \sum_{j=0}^{m-1} T_{i+j} R_j\end{aligned}$$

We have seen how to compute the to terms S1 and S2. Putting them together gives us a formula for M.

$$M = (\underbrace{T * \widehat{R}}_{\text{S1}} - \underbrace{PR}_{\text{S2}})_{m-1:n-1} \tag{4.9}$$

Applying the Fourier Transform and the HE Layer. We use the convolution property of the DFT to convert the convolution in equation 4.9 into a element-wise multiplication.

$$\mathcal{F}(T * \widehat{R} - PR) = \mathcal{F}(T) \odot \mathcal{F}(\widehat{R}) - \mathcal{F}(PR)$$

We wrap this formula into a HE layer.

$$\mathrm{E}_{k_p}(\mathcal{F}(T * \widehat{R} - PR)) = \underbrace{\mathrm{E}_{k_p}(\mathcal{F}(T)) \odot \mathrm{E}_{k_p}(\mathcal{F}(\widehat{R}))}_{\text{S1}} - \underbrace{\mathrm{E}_{k_p}(\mathcal{F}(PR))}_{\text{S2}} \tag{4.10}$$

Using this formula, we define the RFPS protocol. Figure 4.2 provides an overview over the complete FPS scheme. Algorithm 4 shows a pseudocode implementation of the RFPS scheme.

- Given T, the client computes $\mathcal{F}(T)$. It then encrypts $\mathcal{F}(T)$ and sends c_T to the server.
- Given P, the client samples vector R of length m from χ_R. It then precomputes vectors $\widehat{R}$ and PR. It then applies $\mathcal{F}$ to obtain $\mathcal{F}(\widehat{R})$ and $\mathcal{F}(PR)$. It then encrypts $\mathcal{F}(\widehat{R})$ and $\mathcal{F}(PR)$ and sends $c_{\widehat{R}}, c_{PR}$ to the server.
- Given $c_T, c_{\widehat{R}}, c_{PR}$, the server computes c_{TR} as an intermediate result. It then computes c_M according to equation (4.10) and sends it to the client.
- Given c_M, the client decrypts c_M to obtain $\mathcal{F}(M)$. It then applies $\mathcal{F}^{-1}$ to obtain M. From M it then extracts the set of matching indices $\mathcal{I}$ by including all i in $\mathcal{I}$ where $|M_i| \leq t$ for some threshold $t \in \mathbb{R}$.

As is the case with FPS, we use a threshold t to discriminate between matches and non-matches. Any i with an $|M_i| \leq t$ is counted as a match.

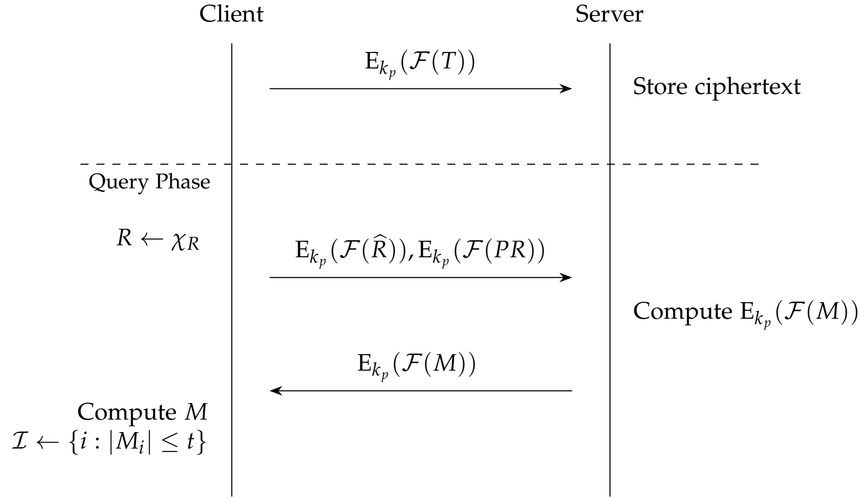

Figure 4.2: High level overview over the RFPS scheme

4.4.1 Variations of RFPS

At this point we want to highlight two possible variations of RFPS that might be of interest.

- **Send R in plain.** In this variant, at the start of the query phase, the client sends R to the server as a plaintext rather than in encrypted form $\mathrm{E}_{k_p}(\mathcal{F}(R))$. The server computes $\mathcal{F}(\widehat{R})$ itself and proceeds with the protocol as normal. This version has the advantage of saving network complexity, as the R plaintext is commonly orders of magnitude smaller than the encrypted ciphertext $c_{\widehat{R}}$. Furthermore, there is less encryption error accumulated in the response ciphertext, as there is no noise from the encryption of R. The downside is, that since R is a vector with the same length as P, the length m is leaked to the server, which is not the case with vanilla RFPS.

- **Don't send P at all.** In this variant, at the start of the query phase, the client sends just the random vector R to the server, but no other data. The server then computes $c_{TR} = \mathrm{E}_{k_p}(\mathcal{F}(T)) \odot \mathcal{F}(R)$ and sends c_{TR} back to the client. Upon receiving c_{TR}, the client unwraps TR and computes $M = (TR - PR)_{m-1:n-1}$ in plain. This is by far the most lightweight version of the RFPS protocol. The network complexity per query is reduced to one plaintext vector and one response ciphertext. Additionally, the server just needs to perform a single homomorphic operation per query. However, one might argue that this variant is so minimalist that it defeats the purpose of the remote computation paradigm, as the server performs such a small portion of the computation.

Algorithm 4: RFPS Pseudocode

Input : $\textsc{pub} = (N, (q_L, \ldots, q_0), P, h, \chi_e, \chi_a, \chi_s, \chi_v, \Delta, t, \chi_R, \Sigma)$

Fn $\texttt{KGen}(1^\lambda, \textsc{pub})$:
    $(k_s, k_p, k_{\text{ev}}) \leftarrow \texttt{CKKS.KGen}(1^\lambda, \textsc{pub})$
    Send k_{ev} to the server.

Fn $\texttt{Setup}_{\mathit{Clt}}(k_s, k_p, T, \textsc{pub})$:
    $c_T \leftarrow \mathrm{E}_{k_p}(\mathcal{F}(T), \textsc{pub})$
    Send c_T to the server.

Fn $\texttt{Setup}_{\mathit{Srv}}(k_{ev}, \textsc{pub})$:
    Store c_T for later use.

Fn $\texttt{Query}_{\mathit{Clt}}(k_s, k_p, P, \textsc{pub})$:
    $R \leftarrow \chi_R$
    $c_{\widehat{R}} \leftarrow \mathrm{E}_{k_p}(\mathcal{F}(\widehat{R}), \textsc{pub})$
    $c_{PR} \leftarrow \mathrm{E}_{k_p}(\mathcal{F}(PR))$
    Send $(c_{\widehat{R}}, c_{PR})$ to the server.

    Receive c_M from the server.
    $M \leftarrow \mathcal{F}^{-1}(\mathrm{D}_{k_s}(c_M), \textsc{pub})$
    $\mathcal{I} \leftarrow \{i : |M_i| \leq t\}$

Fn $\texttt{Query}_{\mathit{Srv}}(k_{ev}, \textsc{tdata}, \textsc{pub})$:
    Receive $(c_{\widehat{R}}, c_{PR})$ from the client.
    $c_M \leftarrow c_T \odot c_{\widehat{R}} - c_{PR}$
    Send c_M to the client.

4.5 Features and Limitations

We highlight a few of the strengths, weaknesses and properties of our newly developed schemes FPS and RFPS.

Privacy. Both schemes have excellent privacy properties. FPS only reveals the pattern length m to the server. RFPS goes even further and does not reveal anything about the text or the queried patterns to the server. In both cases, the matching results are also completely private. Not even the number of matches is revealed.

Text length. Hiding the text and pattern length n, m is only possible because the ciphertext size is exclusively determined by the public parameters

PUB and completely independent from n or m. For example, the encryption of T has a constant size whether T has length $n = 10$ or $n = 1'000$. We cannot store an arbitrarily long T inside a single ciphertext of course.

Remember that CKKS encrypts complex $\frac{N}{2}$-dimensional vectors. Using the DFT symmetry property (section 2.5) and the fact that N is even, we see that if $n \leq N-1$ then the DFT of T has dimension at most $\lfloor \frac{N-1}{2} \rfloor + 1 = \frac{N}{2}$. This shows us that n and m are bounded by the public parameter N in order to ensure correctness.

$$n \leq N-1$$
$$m \leq N-1$$

Efficiency. FPS and RFPS were designed with the capabilities and restrictions of the CKKS HE scheme in mind. The homomorphic circuits have constant multiplicative depth of 1. This is a big advantage as it limits noise growth and enables really efficient computation on the server side. For performance characteristics, see chapter 5.

Ciphertext Size. CKKS ciphertexts are orders of magnitudes larger than the plaintexts they encrypt. The ciphertext expansion is a fundamental property of RLWE based HE schemes (such as CKKS) and is a necessary part of guaranteeing security. This means that a the network cost of the FPS and RFPS protocols is also exceedingly high. In section 4.6 we propose a partial solution to this problem using lossy ciphertext compression.

4.5.1 Comparison of Schemes

Table 4.1 shows a qualitative comparison of FPS, RFPS and the RFPS variants in a selection of disciplines. The RFPS variants *plain R* and *R only* are described in subsection 4.4.1. The number of homomorphic multiplications is a good indicator for computation cost, as ciphertext multiplication is much more expensive than ciphertext addition. The concrete storage size of a CKKS ciphertext depends strongly on the CKKS parameters used (mostly q_L and N). Refer to chapter 5 for real-world storage sizes.

4.6 Ciphertext Compression

FPS and RFPS are efficient constructions in terms of computation cost, but an issue that remains is the size of ciphertexts that need to be sent between the server and the client for a query.

Recall from subsection 2.3.2 that any CKKS ciphertext c consists of a pair of $(N-1)$-degree polynomials with (modular) integer coefficients.

$$c = (c_0, c_1) \in (\mathbb{Z}_q[X]/(X^N+1))^2$$

	FPS	RFPS	RFPS, plain R	RFPS, R only
False Positives	No	Yes	Yes	Yes
Leakage	m	$\perp$	m	m
Storage	2 ciphertexts	1 ciphertext	1 ciphertext	1 ciphertext
Network	3 ciphertexts	3 ciphertexts	2 ciphertexts, 1 vector	1 ciphertext, 1 vector
Hom. Mult.	2	1	1	1

Table 4.1: Qualitative comparison of FPS, RFPS and RFPS variants. *False Positives*: whether or not false positive indices could be returned. *Leakage*: information leaked to the server during the setup and query process combined. *Storage*: amount of information about T the server needs to store long-term in order to answer queries. *Network*: amount of information that needs to be transmitted between the client and server per query. *Hom. Mult.*: number of homomorphic multiplications the server needs to perform in order to answer a query.

In practice, ciphertexts are stored and transmitted as 2 length-N arrays of coefficients plus some metadata. A coefficient a is represented as an integer in $\lceil \log(q) \rceil$ bits. Figure 4.3 shows a schematic representation the bits storing the coefficient a. The rightmost bits are the Least Significant Bits (LSBs).

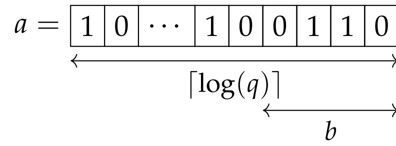

Figure 4.3: Bit representation of an example integer polynomial coefficient a. Every coefficient has a total width of $\lceil \log(q) \rceil$ and b least significant bits are truncated during compression.

The key insight is, that changes to the LSBs of the ciphertext polynomial coefficients have the least amount of impact on the underlying plaintext. Let $m = \mathrm{D}_{k_s}(c) = \mathrm{D}_{k_s}((c_0, c_1))$. For example, if we toggle the least significant bit of every coefficient in c_0 and c_1, then that will only change m very slightly.

Compression Functions. We define the functions `truncate`, `compress` and `decompress`.

- `truncate`(a, b) takes as input a integer coefficient a and a nonnegative integer b. It discards the b LSBs from the binary representation of a and returns the remaining bits.
- `compress`$((c_0, c_1), b_0, b_1)$ takes as input a ciphertext consisting of two polynomials (c_0, c_1) and two nonnegative integers b_0 and b_1. It replaces each coefficient a in c_0 with `truncate`(a, b_0) and each coefficient a in c_1 with `truncate`(a, b_1). It then outputs this compressed version (c'_0, c'_1).

- `decompress`$((c'_0, c'_1), b_0, b_1)$ takes as input a compressed ciphertext consisting of two polynomials (c'_0, c'_1) and two nonnegative integers b_0 and b_1. It appends b_0 0-bits to each coefficient in c_0 and appends b_1 0-bits to each coefficient in c_1. It then outputs the decompressed ciphertext $(\widetilde{c_0}, \widetilde{c_1})$.

Compression Factor. Let $\widetilde{c}$ be the compressed version of c. $\widetilde{c} = \texttt{compress}(c, b_0, b_1)$. Let $\widetilde{m}$ be the decryption of $\widetilde{c}$. $\widetilde{m} = \mathrm{D}_{k_s}(\widetilde{c})$. $\widetilde{c}$ takes $N(b_0 + b_1)$ fewer bits to store than c. Ignoring the constant-sized metadata part of a ciphertext, we achieve a compression factor of $\rho(q, b_0, b_1)$.

$$\begin{aligned}
\rho_q(b_0, b_1) &= \frac{\text{size}(\tilde{c})}{\text{size}(c)} \\
&= \frac{N(\lceil \log(q) \rceil - b_0) + N(\lceil \log(q) \rceil - b_1)}{2N\lceil \log(q) \rceil} \\
&= 1 - \frac{b_0 + b_1}{2\lceil \log(q) \rceil}
\end{aligned}$$

We choose b_0 and b_1 carefully, such that we can save as much ciphertext size as possible while keeping the error incurred by the truncation below a certain threshold. In other words, we choose b_0 and b_1 to minimize $\rho_q(b_0, b_1)$ while keeping the compression error incurred by b_0 and b_1 below a tolerable correctness threshold.

In the case of FPS and RFPS, we take advantage of the knowledge, that the response vector M is a integer vector. Concretely, if a compressed message $\tilde{m} \in \mathbb{R}^k$ is close enough to the actual message $m \in \mathbb{Z}^k$ ($\|m - \widetilde{m}\|_\infty < 0.5$), then we can infer m from $\widetilde{m}$. For a analysis on the compression error, refer to subsection 4.7.2.

4.7 Error Analysis

Multiple steps during the execution of FPS and RFPS introduce errors that might break the correctness of the schemes. We identify four distinct sources of errors.

- **Encryption Error** Denoted ϵ_{enc} The encryption error is introduced by the CKKS encryption algorithm. It is necessary to provide the security guarantees of CKKS.
- **Compression Error** Denoted ϵ_{comp}. The compression error stems from the compression operation on the final result-ciphertext.
- **Numerical and Encoding Error** Denoted ϵ_{num}. The numerical error stems from the imprecision of floating point computation during the

DFT computation and the CKKS encoding and decoding algorithms. Additionally, during CKKS encoding, a rounding step is performed, which is a source of error.

Inevitably, FPS and RFPS will need to be able to deal with some amount of error acting on the result vector M in order to function correctly. Ideally, we choose parameters in such a way such that the infinity norm of the final error-perturbed vector $\widetilde{M}$ differs less than 0.5 from the true vector M.

$$\epsilon_{\text{acc}}(\textsc{pub}, b_0, b_1) = \left\| M - \widetilde{M} \right\|_\infty = \|M - (M + e)\|_\infty = \|e\|_\infty < 0.5$$

Higher bounds than 0.5 are acceptable in a scenario where false positives are allowed.

We provide theoretical or experimental bounds for these errors and discuss them in more detail. Then we discuss how these error types accumulate to influence the correctness of the complete schemes.

4.7.1 Encryption Error

The CKKS encryption function adds random error (or ”noise”) polynomials sampled from χ_e to the output ciphertext in order to conceal the contained plaintext (see subsection 2.3.2). As well as ensuring confidentiality, these errors do unfortunately change the underlying values in a small way. Furthermore, performing operations on ciphertexts can accumulate and quickly increase the error magnitude. As a brief informal example, if we multiply two ciphertexts that encrypt the values $m_0 + e_0$ and $m_1 + e_1$, then the result is $(m_0 + e_0)(m_1 + e_1) = m_0 m_1 + m_0 e_1 + m_1 e_0 + e_0 e_1$. The error of the product has grown to $m_0 e_1 + m_1 e_0 + e_0 e_1$.

Chen et. al. provide probabilistic bounds on these errors and their growth. In particular, we refer to Lemma 1, 2, and 3 in [10]. Equation (4.11) shows terms for B_{clean} and B_{mult} as defined in [10]. P is a large prime used for relinearization and h refers to the Hamming weight of the secret polynomial s.

$$\begin{aligned} B_{\text{clean}} &= 8\sqrt{2}\sigma N + 6\sigma\sqrt{N} + 16\sigma\sqrt{hN} \\ B_{\text{mult}}(l) &= P^{-1} q_l B_{\text{ks}} + B_{\text{scale}} \\ B_{\text{ks}} &= \frac{8\sigma N}{\sqrt{3}} \\ B_{\text{scale}} &= \sqrt{\frac{N}{3}}(3 + 8\sqrt{h}) \end{aligned} \tag{4.11}$$

- **Lemma 1** in [10] states that for a freshly encrypted ciphertext, the encryption noise is bounded by B_{clean} with high probability. Recall the notation for tagged ciphertexts in subsection 2.3.2.

- **Lemma 3** in [10] states that given two ciphertexts $C_0 = (c_0, l, \nu_0, B_0)$ and $C_1 = (c_1, l, \nu_1, B_1)$, the sum of C_0 and C_1 has an error bounded by $B_0 + B_1$. The product of C_0 and C_1 has an error bounded by $\nu_0 B_1 + \nu_1 B_0 + B_0 B_1 + B_{\text{mult}}(l)$.

We combine these Lemmas to provide a probabilistic upper bound on the encryption noise accumulated in the FPS and RFPS circuits.

FPS Encryption Error

We start with four fresh ciphertexts. Recall the tagged ciphertext notation from subsection 2.3.2.

$$\begin{aligned} C_T &= (c_T, 0, \nu_T, B_{\text{clean}}) \\ C_{T^*} &= (c_{T^*}, 0, \nu_{T^*}, B_{\text{clean}}) \\ C_{\widehat{P}} &= (c_{\widehat{P}}, 0, \nu_{\widehat{P}}, B_{\text{clean}}) \\ C_{P^*} &= (c_{P^*}, 0, \nu_{P^*}, B_{\text{clean}}) \end{aligned}$$

Now we follow the homomorphic computation outlined by $\texttt{Query}_{\textbf{Srv}}$ in algorithm 3 and keep track of the errors. We model the intermediate ciphertexts using tagged ciphertexts.

$$\begin{aligned} C_{r^{(m)}} &= (c_{r^{(m)}}, 0, 1, 0) \\ C_{T^{*(m)}} &= (c_{T^{*(m)}}, 0, \nu_{T^{*(m)}}, B_{T^{*(m)}}) \\ C_{T*\widehat{P}} &= (c_{T*\widehat{P}}, 0, \nu_{T*\widehat{P}}, B_{T*\widehat{P}}) \\ C_M &= (c_M, 0, \nu_M, B_M) \end{aligned}$$

Through repeated application of Lemma 3, we obtain a value for B_M.

$$\begin{aligned} B_M &= B_{T^{*(m)}} + B_{P^*} + 2B_{T*\widehat{P}} \\ &= 2B_{T^*} + B_{\text{mult}}(0) + B_{P^*} + 2(\nu_T B_{\widehat{P}} + \nu_{\widehat{P}} B_T + B_{\widehat{P}} B_T + B_{\text{mult}}(0)) \\ &= 3B_{\text{clean}} + 2B_{\text{mult}}(0) + 2B_{\text{clean}}(\nu_T + \nu_{\widehat{P}}) + 2B_{\text{clean}}^2 \end{aligned}$$

From the definitions of T and $\widehat{P}$ we determine their corresponding bounds ν_T and $\nu_{\widehat{P}}$.

$$\begin{aligned} \nu_T &= N \cdot |\Sigma| \cdot \Delta \geq \|\mathcal{F}(T)\|_\infty \cdot \Delta \\ \nu_{\widehat{P}} &= N \cdot |\Sigma| \cdot \Delta \geq \left\|\mathcal{F}(\widehat{P})\right\|_\infty \cdot \Delta \end{aligned}$$

This gives a function to bound the size of the encryption error that acts on the result ciphertext M.

$$\epsilon_{\text{enc}}^{\text{FPS}}(\textsc{pub}) = B_M = 4B_{\text{clean}} N |\Sigma| \Delta + 3B_{\text{clean}} + 2B_{\text{mult}}(0) + 2B_{\text{clean}}^2$$

RFPS Encryption Error

We start with three fresh ciphertexts

$$\begin{aligned} C_T &= (c_T, 0, \nu_T, B_{\text{clean}}) \\ C_{\widehat{R}} &= (c_{\widehat{R}}, 0, \nu_R, B_{\text{clean}}) \\ C_{PR} &= (c_{PR}, 0, \nu_{PR}, B_{\text{clean}}) \end{aligned}$$

We follow the homomorphic computation outlined by $\texttt{Query}_{\textbf{Srv}}$ in algorithm 4. The intermediate result $c_{T\widehat{R}} = c_T \odot c_{\widehat{R}}$ is modelled using the tagged ciphertext $C_{T\widehat{R}}$.

$$\begin{aligned} C_{T\widehat{R}} &= (c_{T\widehat{R}}, 0, \nu_{T\widehat{R}}, B_{T\widehat{R}}) \\ C_M &= (c_M, 0, \nu_M, B_M) \end{aligned}$$

Through repeated application of Lemma 3, we obtain a value for B_M.

$$\begin{aligned} B_M &= B_{T\widehat{R}} + B_{PR} \\ &= \nu_T B_{\widehat{R}} + \nu_{\widehat{R}} B_T + B_T B_{\widehat{R}} + B_{\text{mult}}(0) + B_{PR} \\ &= (\nu_T + \nu_{\widehat{R}} + 1) B_{\text{clean}} + B_{\text{clean}}^2 + B_{\text{mult}}(0) \end{aligned}$$

We derive the following upper bounds ν_T and $\nu_{\widehat{R}}$.

$$\begin{aligned} \nu_T &= N \cdot |\Sigma| \cdot \Delta \geq \|\mathcal{F}(T)\|_\infty \cdot \Delta \\ \nu_{\widehat{R}} &= N \cdot R_{\max} \cdot \Delta \geq \left\|\mathcal{F}(\widehat{R})\right\|_\infty \cdot \Delta \end{aligned}$$

$R_{\max}$ denotes the maximal coefficient value that can be sampled from χ_R. We summarize this bound in the $\epsilon_{\text{enc}}^{\text{RFPS}}$ function.

$$\epsilon_{\text{enc}}^{\text{RFPS}}(\textsc{pub}) = B_M = (N\Delta(|\Sigma| + R_{\max}) + 1)B_{\text{clean}} + B_{\text{clean}}^2 + B_{\text{mult}}(0)$$

4.7.2 Compression Error

The compression method introduced in section 4.6 is lossy. Theorem 4.2 provides an upper bound on the compression error.

Theorem 4.2 *Let c be a ciphertext. Let $\tilde{c}$ be the counterpart of c compressed using b_0 and b_1. The compression error is bounded by*

$$\|\text{Dec}_{k_s}(\tilde{c}) - \text{Dec}_{k_s}(c)\|_\infty^{can} \leq 2^{b_0} N + 2^{b_1} N^2$$

Proof Let s be the secret polynomial in $k_s = (1, s)$ where each coefficient s_i is sampled the ternary distribution $\mathcal{T}$. We know that $\|s\|_\infty^{\text{can}} \leq \|s\|_1 \leq N$.

Let $c = (c_0, c_1)$ be the encryption of polynomial p. The CKKS Decryption operation is defined as follows:

$$\mathrm{Dec}_{k_s}(c) = c_0 + c_1 s \mod q_l$$

During compression, each coefficient in c_0 and c_1 is truncated by throwing away the least significant. For a ciphertext coefficient a and its compressed counterpart $\tilde{a} = \texttt{truncate}(a, b)$ it holds that:

$$a - \tilde{a} \mod q_l < 2^b$$

We obtain the compressed ciphertext $\tilde{c} = (\tilde{c_0}, \tilde{c_1})$ by truncating each coefficient in c_0 and c_1.

$$\begin{aligned}
\|\mathrm{Dec}_{k_s}(\tilde{c}) - \mathrm{Dec}_{k_s}(c)\|_\infty &= \|\tilde{c_0} + \tilde{c_1}s - c_0 - c_1 s\|_\infty \\
&\leq \|\tilde{c_0} - c_0\|_\infty + \|(\tilde{c_1} - c_1)s\|_\infty \\
&\leq 2^{b_0} + \|(\tilde{c_1} - c_1)s\|_\infty \\
&\leq 2^{b_0} + N(\|\tilde{c_1} - c_1\|_\infty \cdot \|s\|_\infty) \\
&\leq 2^{b_0} + 2^{b_1}N
\end{aligned}$$

We have omitted the ($\mod q_l$) operation for legibility. From the definition of the canonical embedding σ (see subsection 2.3.2) we derive the bound $\|x\|_\infty^{\mathrm{can}} = \|\sigma(x)\|_\infty \leq N\|x\|_\infty$.

$$\begin{aligned}
\|\mathrm{Dec}_{k_s}(\tilde{c}) - \mathrm{Dec}_{k_s}(c)\|_\infty^{\mathrm{can}} &\leq N\|\mathrm{Dec}_{k_s}(\tilde{c}) - \mathrm{Dec}_{k_s}(c)\|_\infty \\
&\leq 2^{b_0}N + 2^{b_1}N^2
\end{aligned}$$

□

Using Theorem 4.2, we bound the compression error magnitude in the message space as ϵ_{comp}.

$$\epsilon_{\mathrm{comp}}(\textsc{pub}, b_0, b_1) = 2^{b_0}N + 2^{b_1}N^2$$

We measure the compression error $\|\mathrm{Dec}_{k_s}(\tilde{c}) - \mathrm{Dec}_{k_s}(c)\|_\infty^{\mathrm{can}}$ using an example implementation. Figure 4.4 shows empirical measurements of the errors alongside the theoretical bound ϵ_{comp}.

4.7.3 Numerical and Encoding Error

When it comes to actually implementing our PPSS schemes, the real and complex numbers need to be represented as a finite amount of bits. This leads to numerical errors. The size of the numerical and encoding error depends in part on the scale Δ used for encoding and decoding, as well as the specific hardware and software implementation. We use $\tilde{\cdot}$ to refer to the

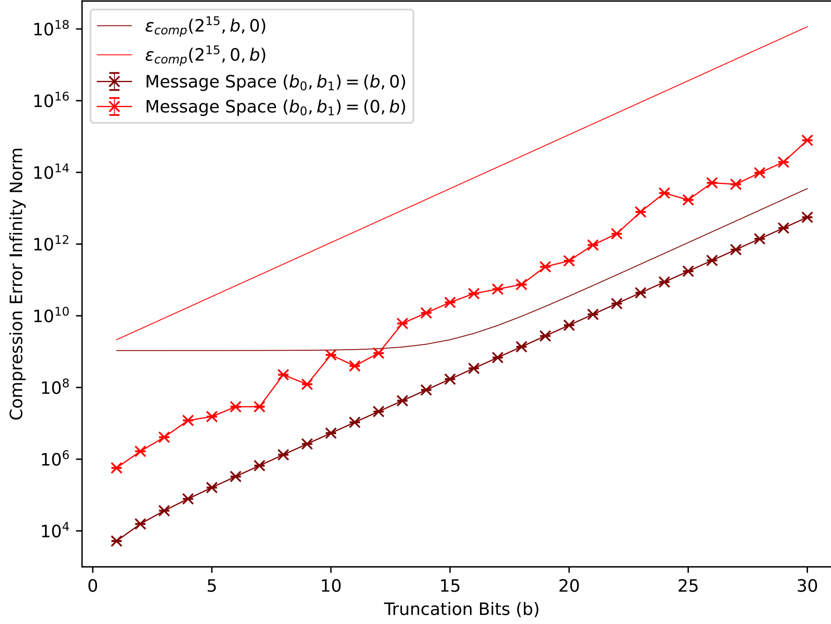

Figure 4.4: Compression error magnitude message space alongside the theoretical bound ϵ_{comp}. b_0 and b_1 is the number of least significant bits truncated in c_0 and c_1 respectively.

imperfect implementations of the underlying mathematical algorithms. For example, $\widetilde{\mathcal{F}}$ refers to a implementation that tries to compute $\mathcal{F}$, but does so imperfectly, with errors.

We provide three separate measurements of error here. Let v be a vector sampled uniformly at random from $\{0,1\}^{2^{15}-1}$. The *DFT error* measures the error when performing one forward and one backward DFT.

$$\frac{\left\| v - \widetilde{\mathcal{F}^{-1}}(\widetilde{\mathcal{F}}(v)) \right\|_\infty}{\|v\|_\infty}$$

The *Encoding error* measures the error when performing one encoding and one decoding step with scale Δ.

$$\frac{\left\| \widetilde{\mathcal{F}}(v) - \widetilde{\text{Dcd}}(\widetilde{\text{Ecd}}(\widetilde{\mathcal{F}}(v))) \right\|_\infty}{\left\| \widetilde{\mathcal{F}}(v) \right\|_\infty}$$

The *DFT and encoding error* measures the error of a full DFT-Encoding pipeline.

$$\frac{\left\| v - \widetilde{\mathcal{F}^{-1}}(\widetilde{\text{Dcd}}(\widetilde{\text{Ecd}}(\widetilde{\mathcal{F}}(v)))) \right\|_\infty}{\|v\|_\infty}$$

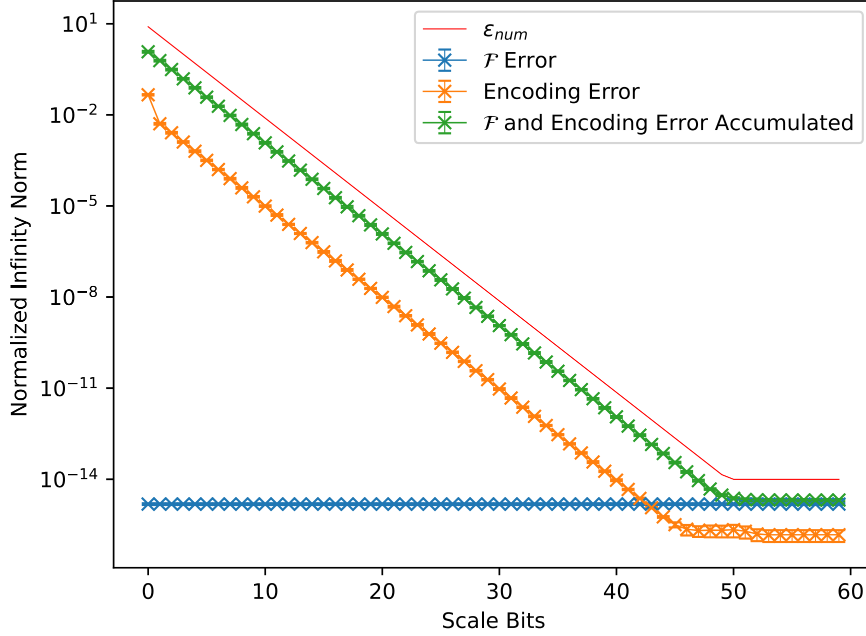

Figure 4.5: Mean numerical errors from DFT and CKKS encoding algorithms, alongside the upper bound $\epsilon_{\text{num}}(\textsc{pub})$. The plaintext polynomial dimension is $N = 2^{15}$. The DFT dimension is $2^{15} - 1$.

Since this kind of error is highly implementation and even hardware dependent, but not necessarily a mathematical property of FPS or RFPS, we provide an experimental bound rather than a theoretical one. Figure 4.5 shows experimental measurements of the error incurred by numerical imprecision in the DFT and Encoding/Decoding steps.

For ϵ_{num}, we use a generous upper bound derived from these empirical measurements.

$$\epsilon_{\text{num}}(\textsc{pub}) = \max(2^{-\log(\Delta)+3}, 10^{-14}) \text{ if } N = 2^{15}$$

4.7.4 Accumulation of Errors

Now that we have defined our three different error functions $\epsilon_{\text{num}}, \epsilon_{\text{enc}}, \epsilon_{\text{comp}}$, we investigate how they interact with each other bound the impact that the accumulation of all these errors has.

We define ϵ_{acc} as the infinity norm of $M - \widetilde{M}$ where M is computed exactly according to its definition, and $\widetilde{M}$ is the output of FPS or RFPS.

$$\epsilon_{\text{acc}}(\textsc{pub}, b_0, b_1) \geq \left\| M - \widetilde{M} \right\|_\infty$$

ϵ_{enc} and ϵ_{comp} both act on the least significant bits of the coefficients in the ciphertext c_M. We can think of them overwriting each other. Thus, only the larger of the two is relevant for the accumulation. Furthermore, as they act on the ciphertext c_M, their magnitude is scaled during the decoding process as well as the inverse DFT. Hence, they are divided by the scale of c_M, denoted Δ_M as well as the dimension of the iDFT, $N-1$.

The numerical error ϵ_{num} scales with the infinity norm of the vector it acts on. Let ν_M denote an upper bound on $\|M\|_\infty$ over all possible values of T, P (and in the case of RFPS: R).

$$\epsilon_{\text{acc}}(\text{PUB}, b_0, b_1) = \epsilon_{\text{num}}(\text{PUB})\nu_M + (1 + \epsilon_{\text{num}}(\text{PUB})) \left(\frac{\max(\epsilon_{\text{enc}}(\text{PUB}), \epsilon_{\text{comp}}(\text{PUB}, b_0, b_1))}{\Delta_M \cdot (N-1)} \right)$$

Choosing a threshold. Both, FPS and RFPS compensate for errors by comparing the computed values $\widetilde{M}_i$ to some threshold t instead of comparing them exactly to 0.

$$\mathcal{I} = \{i : |\widetilde{M}_i| \leq t\}$$

If $M_i = 0$, then $|\widetilde{M}_i| \leq \epsilon_{\text{acc}}(\text{PUB}, b_0, b_1)$. Hence, if we choose $t = \epsilon_{\text{acc}}(\text{PUB}, b_0, b_1)$, then FPS and RFPS will correctly identify all matches.

Typical Error Values. In our experiments we measured empirical values for $\epsilon_{\text{acc}}(\text{PUB}, b_0, b_1)$ using some example parameters. Table 4.2 shows measured errors for some example parameters. We can see that the FPS error is

Scheme	(b_0, b_1)	Mean Error	Std. Deviation
FPS	$(0, 0)$	0.042	0.0002
FPS	$(12, 10)$	0.22	0.012
RFPS	$(0, 0)$	2.53	0.22
RFPS	$(30, 25)$	2.58	0.24

Table 4.2: Experimentally measured error values for example scenarios . $N = 2^{15}$, $n = N - 1$, $m = 10$. *Mean Error:* the mean of the error values $\left\|M - \widetilde{M}\right\|_\infty$ measured over 100 repetitions. (b_0, b_1) denote the compression parameters used. $(0, 0)$ indicates no ciphertext compression.

well below 0.5 in both cases, so we can comfortably choose the matching threshold $t = 0.5$. Even without compression, RFPS already has errors well above 0.5 due to the more conservative public parameters (smaller coefficient modulus q_L, smaller scale Δ). In an application where we tolerate false positives, we choose the matching threshold $t = 4.0$. This choice results in RFPS potentially incorrectly identifying indices as matches where there are

none. For example, for an index i where the true matching value is $|M_i| = 1$, the computed matching value could fit below the threshold $|\widetilde{M}_i| \leq t = 4.0$.

4.7.5 False Positives in RFPS

The RFPS scheme is a probabilistic algorithm that can produce incorrect results. Recall the definition for M_i in equation 4.8:

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j) R_j$$

RFPS identifies index i as a match if $M_i = 0$. From the definition it is easy to see that RFPS will correctly identify all true matches with probability 1. However, there is a possibility that RFPS identifies i as a match even though there is no match at index i. We call this occurrence a False Positive (FP). p_{FP} denotes the probability that such a false positive occurs.

$$p_{\text{FP}}(\chi_T, \chi_P, \chi_R) = \Pr[M_i = 0 | T_{i:i+m-1} = P]$$

Evidently, p_{FP} depends on the distributions of T, P and R called χ_T, χ_P and χ_R respectively. To give a estimation of p_{FP} we make the following assumptions about χ_T and χ_P.

- χ_T, χ_P sample each character independently and uniformly at random from Σ.
- χ_T and χ_P both output vectors of the same length ($n = m$). Hence, the RFPS matching vector M will only contain one element M_0.

Computing p_{FP}. Let J be the sequence of indices where T and P differ.

$$J = (j_0, j_1, \ldots) = (j : T_j \neq P_j)$$

Let the distance $d(T, P)$ be the exact number of characters where T and P differ. $d(T, P) = |J|$. We note that the value of M_0 is exclusively determined by the characters at the indices in J.

$$\begin{aligned} M_0 &= \sum_{j=0}^{m-1} (T_j - P_j) R_j \\ &= \sum_{j \in J} (T_j - P_j) R_j + \sum_{j \notin J} (T_j - P_j) R_j \\ &= \sum_{j \in J} (T_j - P_j) R_j + 0 \\ &= \sum_{j \in J} (T_j - P_j) R_j \end{aligned}$$

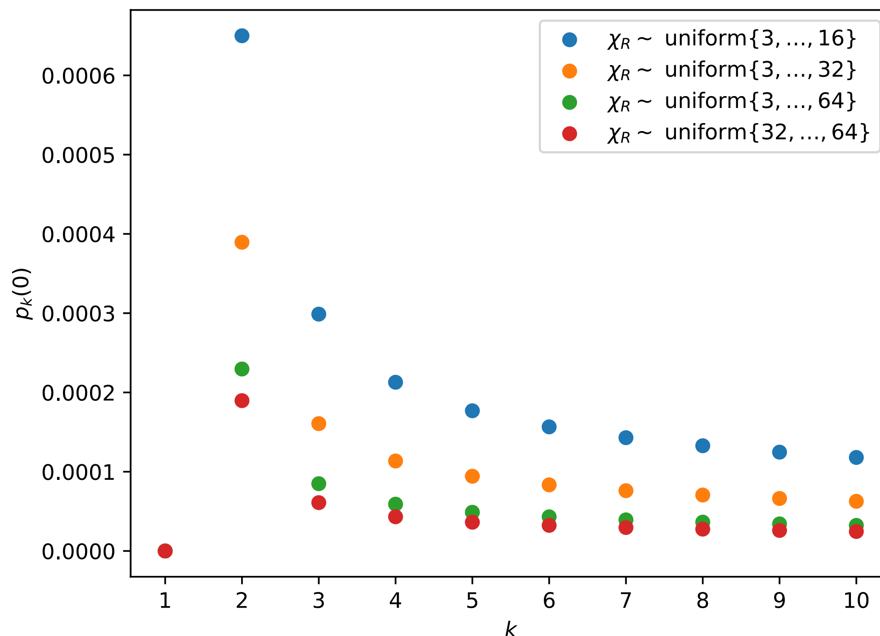

Figure 4.6: Values of $p_k(0)$ for different example χ_R distributions. k is the distance $d(T, P)$. The alphabet is $\Sigma = \{0, \ldots, 255\}$. We assume each character in T and P to be sampled uniformly at random from Σ.

We define $p_k(x)$ to be the probability that $M_0 = x$ given that $d(T, P) = k$.

$$p_k(x) = \Pr[M_0 = x | d(T, P) = k]$$

Obviously, if $T = P$ then $d(T, P) = 0$ and $M_0 = 0$. So we can fix $p_0(x)$.

$$p_0(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$$

For any $k > 0$, $p_k(x)$ is computed using the recursive formula

$$p_k(x) = \sum_{x'=-\infty}^{\infty} p_{k-1}(x') \Pr[(T_{j_{k-1}} - P_{j_{k-1}}) R_{j_{k-1}} = x - x']$$

Figure 4.6 shows values of $p_k(0)$ up to $k = 10$ using some example parameters.

Using $p_k(x)$, we derive an equation for the false positive probability p_{FP}. Recall that we assume P to have the same length as T ($n = m$).

$$\begin{aligned} p_{\text{FP}}(\chi_T, \chi_P, \chi_R) &= \sum_{k=1}^{n} p_k(0) \Pr[d(T, P) = k] \\ &= \sum_{k=1}^{n} p_k(0) \left(1 - \frac{1}{|\Sigma|}\right)^k \left(\frac{1}{|\Sigma|}\right)^{n-k} \binom{n}{k} \end{aligned}$$

Table 4.3 shows the computed false positive probability for some example distributions.

n	χ_R	$p_{\text{FP}}(\chi_T, \chi_P, \chi_R)$
10	uniform$\{3, \ldots, 16\}$	$1.18 \cdot 10^{-4}$
10	uniform$\{3, \ldots, 32\}$	$6.28 \cdot 10^{-5}$
10	uniform$\{3, \ldots, 64\}$	$3.24 \cdot 10^{-5}$
10	uniform$\{32, \ldots, 64\}$	$2.46 \cdot 10^{-5}$

Table 4.3: False positive probabilities for different χ_R distributions. χ_T and χ_P sample characters independently and uniformly from $\Sigma = \{0, \ldots, 255\}$ and output vectors of length $n = m = 10$.

Choosing χ_R. Naturally, we want to minimize the false positive probability p_{FP}. Since χ_T and χ_P are application dependent, our only real dial to tweak p_{FP} is our choice of χ_R. Our analysis has shown that choosing χ_R with large upper bounds is beneficial. Ideally, we would sample R from a distribution that allows arbitrarily large elements, but we are restricted by the CKKS' limitations. If R contains large integers, then this will propagate to large integers in the matching vector M.

CKKS limits the largest plaintext vector element that can be represented. This limit can be increased by tweaking public parameters of course, but doing so also significantly expands the ciphertext size, which is already a concern with HE schemes in general (see section 4.6). The challenge is then to find a reasonable balance between a high-entropy χ_R and limiting the size of the intermediate results in the homomorphic computation. For the experiments in this paper we choose $\chi_R \sim$ uniform$\{3, \ldots, 32\}$.

4.8 Security Analysis

We prove the security of our two schemes assuming that the underlying HE primitive is secure. We use a simulation based proof.

4.8.1 Security Proof for FPS

Theorem 4.3 *The FPS scheme is adaptively SS-CTPA secure with leakage $\mathcal{L}_{\texttt{Setup}}$ and $\mathcal{L}_{\texttt{Query}}$, public parameters* PUB *and security parameter λ. Where $\mathcal{L}_{\texttt{Setup}}$ and $\mathcal{L}_{\texttt{Query}}$ are defined as follows:*

$$\mathcal{L}_{\texttt{Setup}}(T) = \perp$$
$$\mathcal{L}_{\texttt{Query}}(T, P) = m$$

Proof We prove Theorem 4.3 by constructing a simulator Sim for the ideal PPSS-game (Algorithm 2). We then show through a series of game hops, that the ideal game using Sim is computationally indistinguishable from the real PPSS-game (Algorithm 1).

Simulator. The simulator is composed of two functions $\text{Sim}_{\texttt{Setup}}$ and $\text{Sim}_{\texttt{Query}}$.

- $\text{Sim}_{\texttt{Setup}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\texttt{Setup}}(T))$ takes as input the security parameter 1^λ, public parameters PUB and the leakage $\mathcal{L}_{\texttt{Setup}}(T) = \perp$. It sends to the server a pair of ciphertexts $(\text{E}_{k_p}(\mathcal{F}(0)), \text{E}_{k_p}(\mathcal{F}(0)))$.
- $\text{Sim}_{\texttt{Query}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\texttt{Query}}(T, P))$ takes as input the security parameter 1^λ, public parameters PUB and the leakage $\mathcal{L}_{\texttt{Query}}(T, P) = m$. It sends to the server a tuple $(\text{E}_{k_p}(\mathcal{F}(0)), \text{E}_{k_p}(\mathcal{F}(0)), m)$.

Figures 4.7, 4.8 and 4.9 show games G_1, G_2 and G_3 respectively.

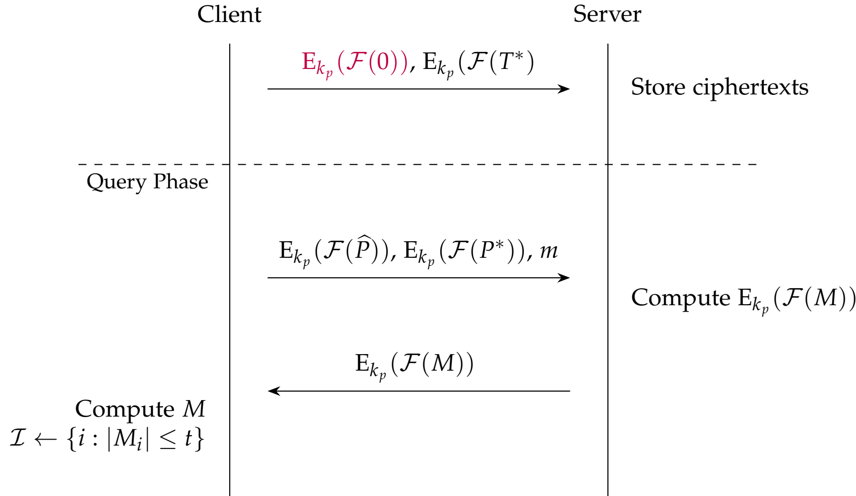

Figure 4.7: Game G_1 for the FPS scheme.

Hop to G_1. Let ε be the HE scheme used by FPS. We show that if adversary $\mathcal{A}$ can distinguish between games Real and G_1, then we can construct an

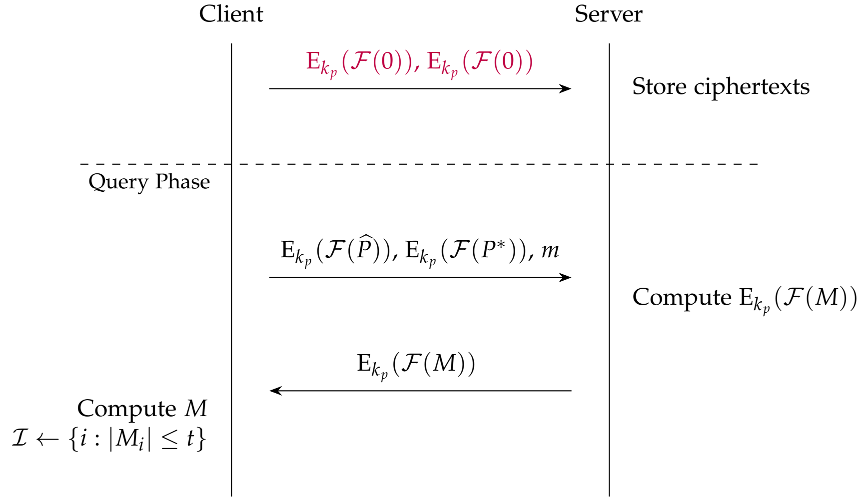

Figure 4.8: Game G_2 for the FPS scheme.

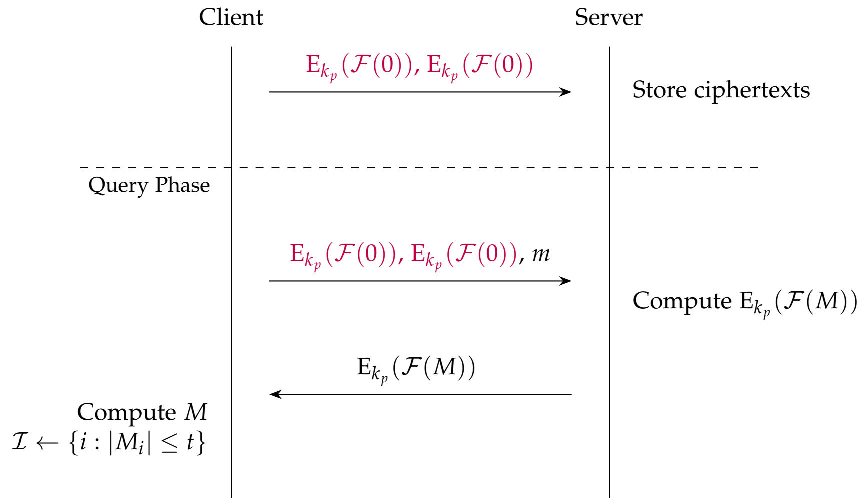

Figure 4.9: Game G_3 for the FPS scheme.

adversary $\mathcal{B}$ that successfully attacks the IND-CPA security of ε.

$$|\Pr[\mathrm{Real}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A}}(1^\lambda, \textsc{pub}) = 1] - \Pr[G^{\mathrm{PPSS}}_{1,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1]| \leq \mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B})$$

$\mathcal{B}$ interacts with a IND-CPA Left-or-Right (LoR) oracle and invokes $\mathcal{A}$ as a subroutine. Upon receiving the text T from $\mathcal{A}$, it sends $(\mathcal{F}(T), \mathcal{F}(0))$ as a query to the oracle and receives back a ciphertext c which is either an encryption of $\mathcal{F}(T)$ or $\mathcal{F}(0)$. $\mathcal{B}$ then proceeds normally through the FPS scheme until it receives the bit b from $\mathcal{A}()$. It directly outputs b. If $\mathcal{A}$ can indeed distinguish between Real and G_1, then $\mathcal{B}$ can use $\mathcal{A}$s output to distinguish between a L-oracle and a R-oracle.

Hop to G_2. The hop from G_1 to G_2 works analogously to the first hop. This time $\mathcal{B}$ is given an adversary $\mathcal{A}$ that can distinguish between G_1 and G_2 as subroutine and uses it to distinguish between the encryptions of $(\mathcal{F}(T^*)$ and $\mathcal{F}(0))$.

$$|\Pr[G^{\mathrm{PPSS}}_{1,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1] - \Pr[G^{\mathrm{PPSS}}_{2,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1]| \\ \leq \mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B})$$

Hop to G_3. Let $\mathcal{A}$ be an adversary that distinguishes between G_2 and G_3 and let n_q the number of queries that $\mathcal{A}$ makes. We construct $\mathcal{B}$ in the same way as we did before, but this time distinguishing the encryptions of $(\mathcal{F}(\widehat{P}), \mathcal{F}(0))$ and $(\mathcal{F}(P^*), \mathcal{F}(0))$. Notice that per query that $\mathcal{A}$ makes, $\mathcal{B}$ makes two queries.

$$|\Pr[G^{\mathrm{PPSS}}_{2,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1] - \Pr[G^{\mathrm{PPSS}}_{3,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1]| \\ \leq 2n_q \,\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B})$$

Game G_3 is the same as the ideal game, so no adversary will ever exist to distinguish the two. For legibility, let p_i be the probability that game G_i returns 1 and $p_{\mathrm{Real}}, p_{\mathrm{Ideal}}$ be the probabilities that games Real and Ideal return 1 respectively. Plugging it all together, we get

$$\begin{aligned} \mathrm{Adv}^{\text{SS-CTPA}}_{\mathrm{FPS}}(\mathcal{A}) &= |p_{\mathrm{Real}} - p_{\mathrm{Ideal}}| \\ &\leq |p_{\mathrm{Real}} - p_1| + |p_1 - p_2| + |p_2 - p_3| + |p_3 - p_{\mathrm{Ideal}}| \\ &\leq 2\,\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B}) + 2n_q\,\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B}) + 0 \\ &= (2 + 2n_q)\,\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B}) \end{aligned}$$

If ε is a IND-CPA secure HE scheme, then $\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B}) \leq \mathrm{negl}(\lambda)$. It follows that

$$\mathrm{Adv}^{\text{SS-CTPA}}_{\mathrm{FPS}}(\mathcal{A}) \leq (2 + 2n_q)\,\mathrm{Adv}^{\text{IND-CPA}}_{\varepsilon}(\mathcal{B}) \leq \mathrm{negl}(\lambda) \qquad \square$$

4.8.2 Security Proof for RFPS

Theorem 4.4 *The RFPS scheme is adaptively SS-CTPA secure with leakage $\mathcal{L}_{\texttt{Setup}}$ and $\mathcal{L}_{\texttt{Query}}$, public parameters* PUB *and security parameter λ. Where $\mathcal{L}_{\texttt{Setup}}$ and $\mathcal{L}_{\texttt{Query}}$ are defined as follows:*

$$\mathcal{L}_{\texttt{Setup}}(T) = \perp \\ \mathcal{L}_{\texttt{Query}}(T, P) = \perp$$

Proof We prove Theorem 4.4 by constructing a simulator Sim for the ideal PPSS-game (Algorithm 2). We then show through a series of game hops, that the ideal game using Sim is computationally indistinguishable from the real PPSS-game (Algorithm 1).

Simulator. The simulator is composed of two functions $\text{Sim}_{\texttt{Setup}}$ and $\text{Sim}_{\texttt{Query}}$.

- $\text{Sim}_{\texttt{Setup}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\texttt{Setup}}(T))$ takes as input the security parameter 1^λ, public parameters $\textsc{pub}$ and the leakage $\mathcal{L}_{\texttt{Setup}}(T) = \perp$. It sends to the server a ciphertext $\text{E}_{k_p}(\mathcal{F}(0))$.
- $\text{Sim}_{\texttt{Query}}(1^\lambda, \textsc{pub}, \mathcal{L}_{\texttt{Query}}(T, P)$ takes as input the security parameter 1^λ, public parameters $\textsc{pub}$ and the leakage $\mathcal{L}_{\texttt{Query}}(T, P) = \perp$. It sends to the server a pair of ciphertexts $(\text{E}_{k_p}(\mathcal{F}(0)), \text{E}_{k_p}(\mathcal{F}(0)))$.

Figures 4.10 and 4.11 show games G_1, G_2 respectively.

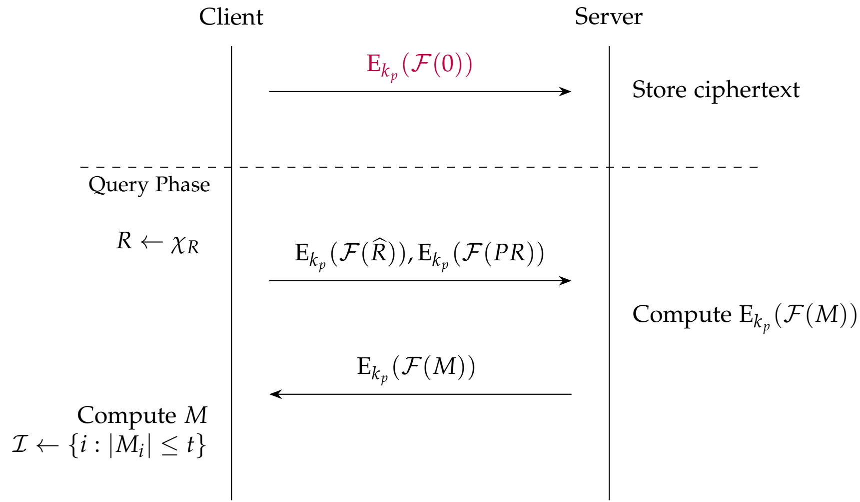

Figure 4.10: Game G_1 for the RFPS scheme.

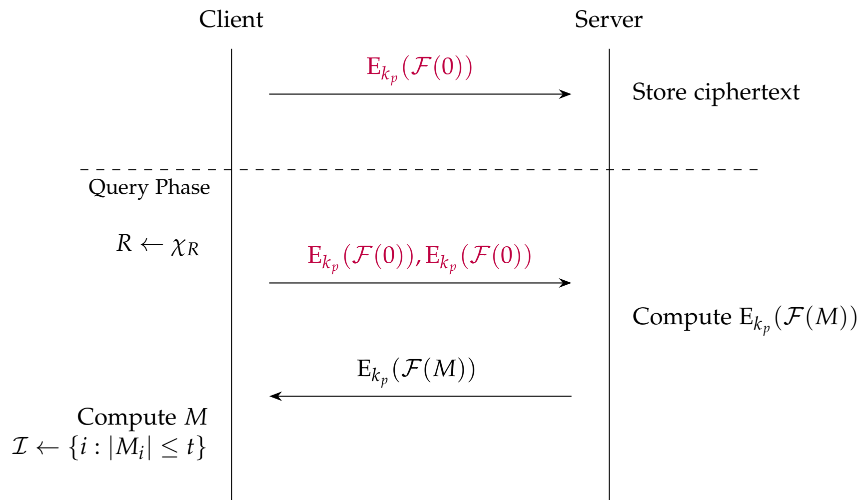

Figure 4.11: Game G_2 for the RFPS scheme.

Hop to G_1. Let ε be the HE scheme used by FPS. We show that if adversary $\mathcal{A}$ can distinguish between games Real and G_1, then we can construct an adversary $\mathcal{B}$ that successfully attacks the IND-CPA security of ε.

$$|\Pr[\mathrm{Real}^{\mathrm{PPSS}}_{\mathcal{S},\mathcal{A}}(1^\lambda, \textsc{pub}) = 1] - \Pr[G^{\mathrm{PPSS}}_{1,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1]| \leq \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B})$$

$\mathcal{B}$ interacts with a LoR oracle and invokes $\mathcal{A}$ as subroutine. Upon receiving T from $\mathcal{A}$, it sends the pair $\mathcal{F}(T), \mathcal{F}(0)$ as a query to the oracle and receives back a ciphertext c, which is either an encryption of $\mathcal{F}(T)$ or $\mathcal{F}(0)$. $\mathcal{B}$ then proceeds normally thorugh the RFPS scheme until it receives the bit b from $\mathcal{A}()$. It directly outputs b.

Hop to G_2. Let $\mathcal{A}$ be an adversary that distinguishes between G_1 and G_2 and let n_q be the number of queries that $\mathcal{A}$ makes. We construct $\mathcal{B}$ in the same way as we did before, but this time distinguishing the encryptions of $(\mathcal{F}(\widehat{R}), \mathcal{F}(0))$ and $(\mathcal{F}(PR), \mathcal{F}(0))$.

$$\begin{aligned}|\Pr[G^{\mathrm{PPSS}}_{1,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1] - \Pr[G^{\mathrm{PPSS}}_{2,\mathcal{S},\mathcal{A},\mathcal{L},\mathrm{Sim}}(1^\lambda, \textsc{pub}) = 1]|& \\ \leq 2n_q \, \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B})&\end{aligned}$$

Game G_2 is the same as the ideal game, so no adversary will ever exist to distinguish the two. For legibility, let p_i be the probability that game G_i returns 1 and $p_{\mathrm{Real}}, p_{\mathrm{Ideal}}$ be the probabilities that games Real and Ideal return 1 respectively. Plugging it all together, we get

$$\begin{aligned}\mathrm{Adv}^{\mathrm{SS\text{-}CTPA}}_{\mathrm{RFPS}}(\mathcal{A}) &= |p_{\mathrm{Real}} - p_{\mathrm{Ideal}}| \\ &\leq |p_{\mathrm{Real}} - p_1| + |p_1 - p_2| + |p_2 - p_{\mathrm{Ideal}}| \\ &\leq \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B}) + 2n_q \, \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B}) + 0 \\ &= (1 + 2n_q) \, \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B})\end{aligned}$$

If ε is a IND-CPA secure HE scheme, then $\mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B}) \leq \mathrm{negl}(\lambda)$. It follows that

$$\mathrm{Adv}^{\mathrm{SS\text{-}CTPA}}_{\mathrm{RFPS}}(\mathcal{A}) \leq (1 + 2n_q) \, \mathrm{Adv}^{\mathrm{IND\text{-}CPA}}_{\varepsilon}(\mathcal{B}) \leq \mathrm{negl}(\lambda) \qquad \square$$

Chapter 5

Implementation

We implemented the FPS and RFPS schemes in C++ using the CKKS implementation in the Microsoft SEAL library [26] and tested the performance as well as correctness.

5.1 Parameter choice

CKKS Parameters. The CKKS polynomial modulus degree N must be a power of 2. We choose $N = 2^{15}$ unless otherwise noted. This is the maximal value for N that is supported by MS SEAL. In FPS, we choose the maximal level to be $L = 1$ and coefficient modulus chain (q_1, q_0) where q_0 is a 60 bit wide prime and q_1 is the product of q_0 and another 60 bit wide prime. In RFPS, we choose the maximal level to be $L = 0$ and the coefficient modulus chain (q_0) where q_0 is a 60 bit wide prime. For P we choose another 60 bit prime. In [10] the parameter h defines the exact Hamming weight of the secret polynomial s. This notion of h is not applicable in this implementation, as MS SEAL samples s from the generic ternary distribution $\mathcal{T}$ and thus the value of h is not a public parameter known prior to sampling the secret key. The distributions $\chi_e, \chi_a, \chi_s, \chi_v$ are instantiated by MS SEAL according to the descriptions in subsection 2.3.2. For the scale Δ that is applied during encoding, we choose 2^{40} for FPS and 2^{20} for RFPS.

According to the recommendations made by the HE standard in 2018 [2], this parameter choice easily fits the requirements for the 128 bit security level in RLWE-based schemes.

String Search Parameters For all tests, we use the UTF-8 alphabet $\Sigma = \{0, \ldots, 255\}$. For T and P, we sample each character uniformly at random from Σ. Unless otherwise noted, T has the maximal possible length $n = N - 1 = 2^{15} - 1$. The pattern has length $m = 100$. Note that homomorphic computation performance and network complexity do not depend on n or

m as long as they are both below the upper bound N. For FPS, we choose a matching threshold of $t = 0.5$. For RFPS, we choose a more generous threshold of $t = 4.0$ and a R distribution $\chi_R \sim \text{uniform}\{3, \ldots, 32\}$.

5.2 Results

Experiment Setup. All experiments were performed on a laptop with a Intel i7-8850H CPU @ 2.60GHz and 16GB of memory.

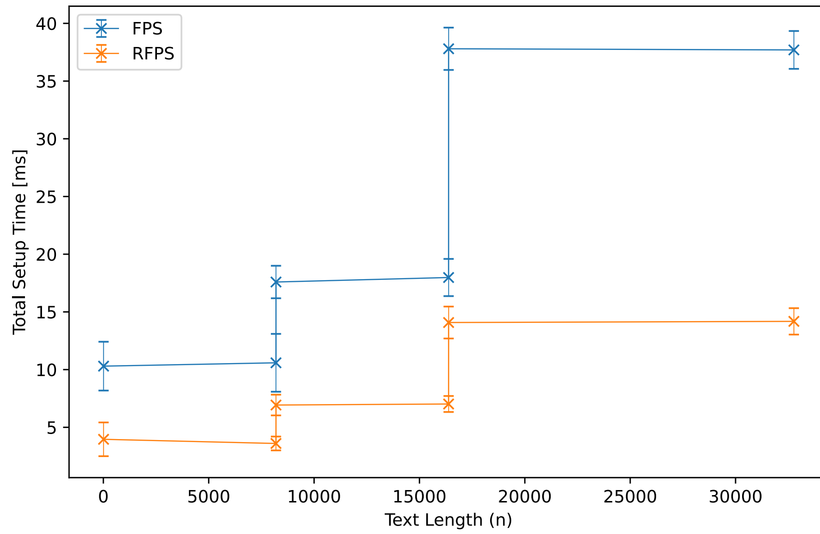

Figure 5.1: Setup times for different text lengths. CKKS public parameter N was chosen such that $n \leq N - 1$. Setup time includes encoding and encryption of the text information.

Computation Cost. Figures 5.1 and 5.2 show the computation times for the setup process and query process respectively. Recall that computation cost scales with the public parameter N, which must be a power of 2. For these measurements, N was chosen to be the smallest power of 2 such that $n \leq N - 1$. This explains the jumps at points $n = 2^{13} = 8'192$ and $n = 2^{14} = 16'384$ in these and all subsequent performance graphs.

Given a text of length $n = 32'000$ and a pattern of any length $m \leq n$, a query in FPS takes less than 60ms. In RFPS, the same query takes less than 40ms. As expected, RFPS is more lightweight than FPS due to the simpler homomorphic computation performed.

Figure 5.3 shows a breakdown of the FPS query computation time into its components. The query processing time is dominated by the encoding and

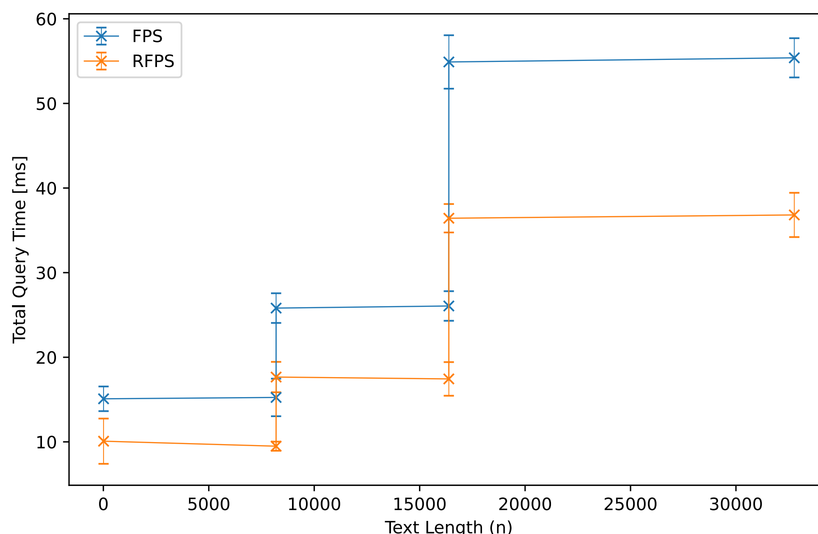

Figure 5.2: Query times for different text lengths. CKKS public parameter N was chosen such that $n \leq N - 1$. Query time includes encoding, encryption, homomorphic computation, decryption and decoding, but no network transmission time.

encryption processes on the client side as well as the homomorphic computations on the server side.

Storage Cost. Figure 5.4 shows ciphertext sizes for FPS. Assuming $N = 2^{15}$, any fresh ciphertext, for example the encryption of $\mathcal{F}(T)$ has a size larger than 1MB. The FPS implementation performs a rescaling operation after the homomorphic computation, which reduces ciphertext sizes significantly. This is why the size of the response ciphertext is less than 600KB. We compress the response further by applying our compression method. All polynomial coefficients in the response ciphertext originally have a width of 60 bits each. We truncate the 10 least significant bits from coefficients in c_0 and 12 least significant bits from coefficients in c_1. The resulting compressed ciphertext is roughly 23% smaller than the original response. In our tests, FPS is still correct with these compression parameters.

RFPS uses the CKKS plaintext space more efficiently. Hence we can choose the CKKS public encryption parameters more conservatively. This results in fresh ciphertexts being much smaller than in FPS. Since the RFPS implementation does not rescale ciphertexts at any point, the response ciphertext is exactly as large as a fresh ciphertext. However, we can be more aggressive with our compression parameters. In c_0 polynomial coefficients, we truncate 30 out of 60 least significant bits and in c_1 polynomial coefficients, we truncate 25 out of 60 least significant bits. The resulting compressed ciphertext

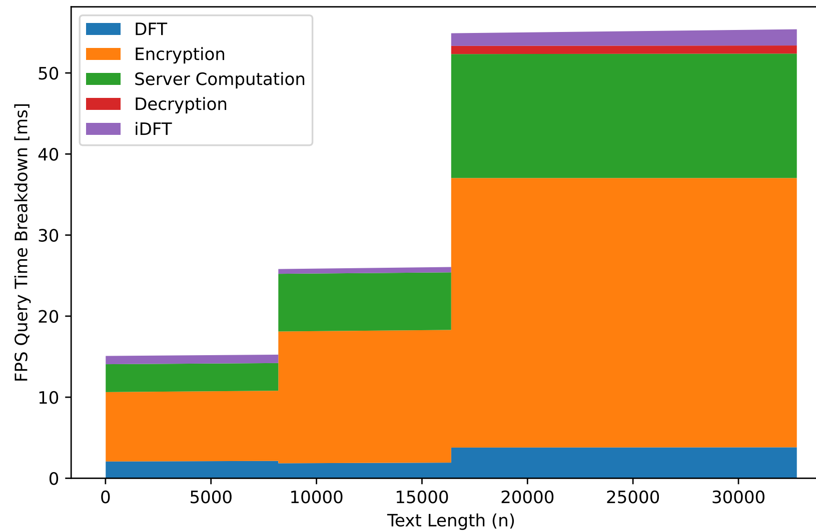

Figure 5.3: FPS Query Time Breakdown for different text lengths. CKKS public parameter N was chosen such that $n \leq N - 1$.

is roughly 49% smaller than the original response. Again, in our tests, these compression parameters did not affect correctness of RFPS.

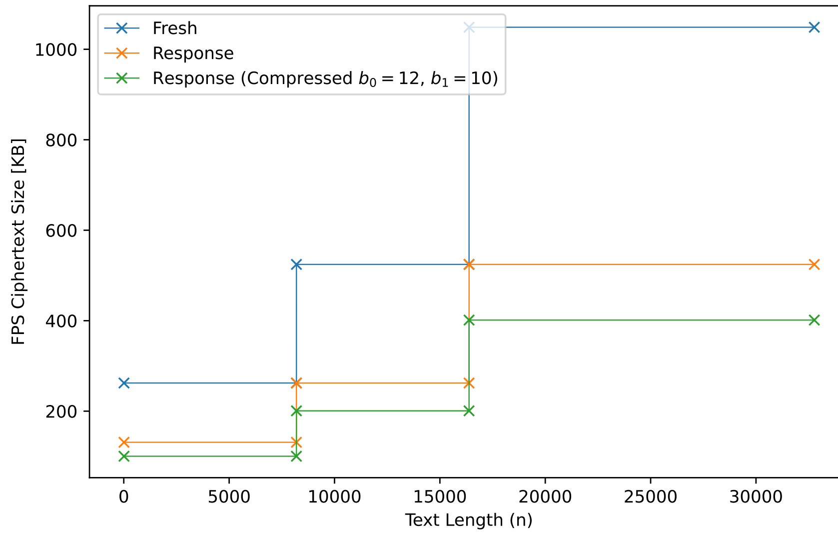

Figure 5.4: FPS Ciphertext sizes for different text lengths. CKKS public parameter N was chosen such that $n \leq N - 1$.

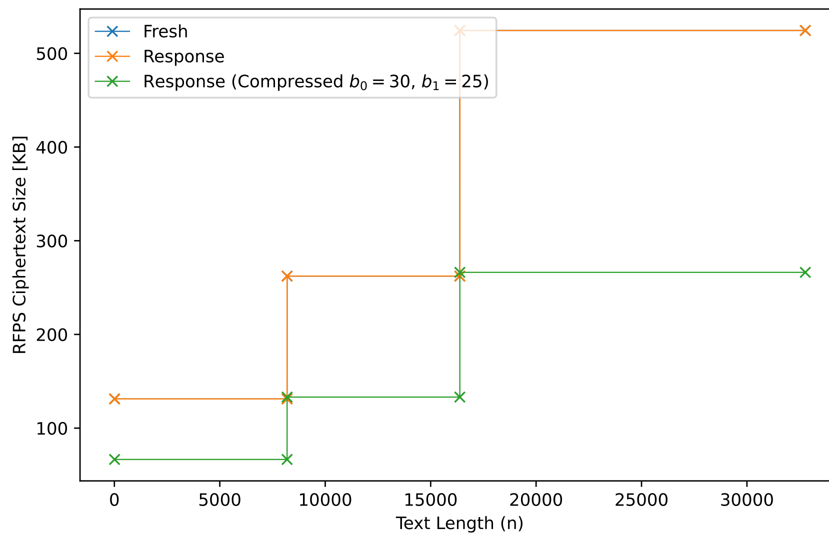

Figure 5.5: RFPS Ciphertext sizes for different text lengths. CKKS public parameter N was chosen such that $n \leq N - 1$.

Chapter 6

Future Work

There are many exciting directions that future research projects could take. Here we highlight some ideas.

6.1 Wildcard String Search

Adding wildcard characters is a natural extension to the string search problem. Informally, the wildcard character, denoted "$*$", is a special new character that matches with every other character in the alphabet. It can be used in either the text or the pattern to make more flexible queries. Consider T and P:

$$T = (0, 1, 0, 2, 0)$$
$$P = (0, *, 0)$$

The wildcard in P matches with characters 1 as well as 2, so the correct result of wildcard string search is $\mathcal{I} = \{0, 2\}$.

Implementing Wildcards in RFPS. In FPS and RFPS, the wildcard functionality is fairly simple to implement by modifying the definitions of M. For example, in RFPS we can implement pattern-wildcards at almost no additional cost. We sample R as usual from χ_R, but then replace R_i with 0 if $P_i = *$. The definition of M remains as before.

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j) R_j$$

Since $R_j = 0$ if $P_j = *$, the j-th term of the sum evaluates to 0 regardless of the values T_{i+j} or P_j. The wildcard version of RFPS is as efficient and secure as vanilla RFPS.

Implementing Wildcards in FPS. To implement pattern-wildcards in FPS, we introduce a new masking vector $H \in \{0,1\}^m$.

$$H_i = \begin{cases} 0 \text{ if } P_i = * \\ 1 \text{ otherwise} \end{cases}$$

We adapt the definition of M to incorporate H.

$$M_i = \sum_{j=0}^{m-1} (T_{i+j} - P_j)^2 H_j$$

The wildcard version of FPS is less efficient than the vanilla version of FPS due to the additional homomorphic multiplication as well an additional ciphertext c_H needing to be transmitted.

6.2 Improve Compression

As we have seen in chapter 5, ciphertext size is a major challenge for FPS, RFPS and HE schemes in general. Encrypting a 30KB UTF-8 text in FPS results in a ciphertext size of more than 1MB without compression. This corresponds to a ciphertext expansion of roughly 34×. The response ciphertext is smaller at 600KB, but still about 20× times larger than the plaintext.

In our compression method (section 4.6), we have exploited the knowledge that our final ciphertext is composed of integers rather than real numbers. However, there might still be a big untapped potential for more compression. Ultimately, in FPS and RFPS, we only care about whether or not a value M_i is zero ($|M_i| \leq t$) or non-zero ($|M_i| > t$). So in theory, we could encode all the important information about M in just $|M| = n - m + 1$ bits.

6.3 Using other Homomorphic Encryption Primitives

While the string search schemes developed in this thesis use CKKS as the underlying HE primitive, there is nothing stopping us from using a different HE scheme to compute the matching formulas. This might yield a new string search scheme that has different practicality characteristics based on the advantages and disadvantages of the HE primitive used.

6.3.1 Using linearly HE schemes

Given the simplicity of the homomorphic circuits in FPS and RFPS, some well established linearly HE schemes might be suitable candidates as HE primitives. In contrast to the vectorized RLWE-based schemes, schemes in this category usually encrypt and operate on scalar values. This gives us a lot more flexibility, but might also be less performant, as we cannot make use of the powerful SIMD operations of RLWE-based HE.

Using BGN. BGN introduced by Boneh et. al. in [4] is a encryption scheme that enables homomorphic operations over integers. We highlight the most relevant properties of BGN.

- **Homomorphic Additions.** The user can perform an arbitrary number of additions of ciphertexts.
- **Homomorphic Multiplications.** The user can perform multiplications of two ciphertexts. However, the result ciphertext of the multiplication cannot be used in further multiplications.
- **Decryption.** Decryption involves solving the discrete logarithm problem for a small set of possible exponents. In [4], the authors use boolean values as plaintexts, so only the two exponents 0 and 1 need to be checked.

Points 1, 2 and 3 make BGN capable of evaluating any multivariate polynomials of degree 2, so long as the set of possible results is small. FPS and RFPS provide definitions for M_i that are just that! M_i is a multivariate polynomial of degree 2 with the characters in T, P (and R) as function parameters. Furthermore, to extract the indices $\mathcal{I}$ from M, we only need to know whether or not M_i is zero or non-zero. Given the BGN encryption of M_i, it is therefore possible to efficiently deduce if M_i is zero. BGN stands out as a possibly viable alternative to CKKS as HE primitive. Several advantages and disadvantages over CKKS should be considered.

- The scalar nature of BGN allows us to encrypt each character in T, P (and R) individually, rather than as a monolithic ciphertext. This makes the ways in which they can be used a lot more flexible. For example, we lose the upper bound on n that CKKS mandated and in theory, T and P could be arbitrarily long.
- CKKS provided element-wise multiplication of vectors as a single efficient SIMD operation. We made great use of this fact in our constructions by applying the convolution property of the DFT. In Fourier space, computing the convolution could be done using n multiplications or one element-wise vector multiplication. As BGN encrypts integers, we unfortunately cannot perform the convolution computation in Fourier space. Instead, we need to compute the convolution naively using $O(n^2)$ homomorphic multiplications.
- The new flexibility of encrypting each character separately opens the door to employing SIMD and parallelization in a more traditional sense. As a primitive example, each M_i ciphertext could be computed on a separate CPU thread and multiple homomorphic operations could be performed at once using vectorized CPU instructions.

- A BGN based implementation of FPS and RFPS is likely to have a substantially lower ciphertext expansion factor than the CKKS version we presented.

Exploring whether the increased flexibility of a BGN-based FPS-like construction is worth the trade offs in practice is an interesting open research question.

6.3.2 Using boolean HE schemes

Boolean HE schemes such as Goldwasser-Micali (GM) [15] are fundamentally different from the other schemes in this list as they offer boolean operations over ciphertexts rather than arithmetic operations. Unfortunately, this makes GM rather unsuitable as HE primitive for FPS which relies heavily on arithmetic to compute M. The boolean operation paradigm might open the door for completely different design approaches to PPSS, though.

6.3.3 Using other RLWE-based HE schemes

Next to CKKS, many other RLWE-based schemes exist and there is active development in this area. BGV [7] and BFV [13, 6] are implemented in the MS SEAL library and have a similar interfaces as CKKS. In particular, they also operate in a SIMD manner, which is one of the key strengths used by FPS and RFPS. In contrast to CKKS however, they operate on (modular) integer vectors instead of complex vectors. This makes them unsuitable as direct drop-in replacements. Working with integers rather than floating point complex numbers is somewhat easier as there is no numerical imprecision. Exploring this possibility may be worth-wile, for example using the Number Theoretic Transform.

Number Theoretic Transform The Number Theoretic Transform (NTT) [1] is a extension of the DFT to finite fields (i.e. integers modulo a prime). It exhibits the same convolution property of the DFT that we rely on for FPS, but does so with *no* numerical imprecision. One could use the NTT combined with a integer vector scheme such as BFV to build a more practical version of FPS or RFPS.

6.4 Privacy Preserving String Search as Building Block

With PPSS syntax and security formalized and implementations being more practical, we can explore the possibility of using PPSS schemes as building blocks for more complex cryptographic primitives.

Substring Private Information Retrieval. Private Information Retrieval (PIR) refers to a mechanism where a client retrieves an item from a database stored on a server without the server knowing what item was retrieved. In some cases, the database is publicly known. This mechanic can of course be extended to the realm of string search. A client might want to retrieve the text segments surrounding all occurrences of a pattern in a large text. This functionality is subtly different from PPSS, but existing PPSS schemes might serve as a stepping stone towards achieving practical substring-PIR.

Encrypted Databases. In [3], Bian et. al. propose a HE-based encrypted Database Management System called HE3DB. HE3DB offers various different comparison and aggregation functionalities that are common in SQL queries. Notably, HE3DB features exact string comparisons but no substring queries. Using PPSS, HE3DB could potentially be expanded to allow for substring queries using the SQL LIKE keyword.

Chapter 7

Conclusion

In this thesis we have developed two new schemes for Privacy Preserving String Search using Homomorphic Encryption as a building block. We designed the constructions in such a way as to minimize to computational complexity of the homomorphic circuits. In particular, the circuits have multiplicative depth 1 and a constant number of additions.

The resulting schemes are really computationally efficient, being able to perform string search queries in a matter of tens of milliseconds. This demonstrates the capability of modern HE schemes to be fast and practical if used with care. On the other hand, network complexity remains a shortcoming of our PPSS schemes despite our compression efforts. Using CKKS within (R-)FPS, we observe ciphertext expansion factors of $20\times$ and upwards. Network complexity seems to be the critical factor keeping HE-based PPSS schemes from being fully practical.

In the case of PPSS, the target communication cost per query should be lower than the size of text T. Otherwise, if the communication is higher, the server could store a symmetrically encrypted ciphertext of T during setup phase and just stream this ciphertext back to the client each time it makes a query.

The phenomenon of large ciphertext expansion seems to extend across all of RLWE-based HE. RLWE-based HE relies on the additional entropy provided by larger ciphertexts for security. It remains a key open question by how much we can reduce ciphertext expansion of FHE-based PPSS solutions, such that they are competitive with other solutions.

Bibliography

[1] R. Agarwal and C. Burrus. Fast convolution using fermat number transforms with applications to digital filtering. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 22(2):87–97, 1974.

[2] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[3] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. He^{3}db: An efficient and elastic encrypted database via arithmetic-and-logic fully homomorphic encryption. Cryptology ePrint Archive, Paper 2023/1446, 2023. https://eprint.iacr.org/2023/1446.

[4] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Charlotte Bonte and Ilia Iliashenko. Homomorphic string search with constant multiplicative depth. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, CCSW'20, page 105–117, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. Cryptology ePrint Archive, Paper 2012/078, 2012. https://eprint.iacr.org/2012/078.

[7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. Fully homomorphic encryption without bootstrapping. Cryptology ePrint Archive, Paper 2011/277, 2011. https://eprint.iacr.org/2011/277.

[8] Michael Burrows, D J Wheeler D I G I T A L, Robert W. Taylor, David J. Wheeler, and David Wheeler. A block-sorting lossless data compression algorithm. 1994.

[9] Melissa Chase and Emily Shen. Substring-searchable symmetric encryption. Cryptology ePrint Archive, Paper 2014/638, 2014. https://eprint.iacr.org/2014/638.

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. Cryptology ePrint Archive, Paper 2016/421, 2016. https://eprint.iacr.org/2016/421.

[11] Emiliano De Cristofaro, Sky Faber, and Gene Tsudik. Secure genomic testing with size- and position-hiding private substring matching. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, page 107–118, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Paper 2015/927, 2015. https://eprint.iacr.org/2015/927.

[13] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Paper 2012/144, 2012. https://eprint.iacr.org/2012/144.

[14] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery.

[16] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. Leakage-abuse attacks against order-revealing encryption. Cryptology ePrint Archive, Paper 2016/895, 2016. https://eprint.iacr.org/2016/895.

[17] Florian Hahn, Nicolas Loza, and Florian Kerschbaum. Practical and secure substring search. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 163–176, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Yu Ishimaki, Hiroki Imabayashi, and Hayato Yamana. Private substring search on homomorphically encrypted data. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, 2017.

[19] Joon-Woo Lee, Hyungchul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, and Jong-Seon No. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[20] Iraklis Leontiadis and Ming Li. Storage efficient substring searchable symmetric encryption. Cryptology ePrint Archive, Paper 2017/153, 2017. https://eprint.iacr.org/2017/153.

[21] Baiyu Li and Daniele Micciancio. On the security of homomorphic encryption on approximate numbers. Cryptology ePrint Archive, Paper 2020/1533, 2020. https://eprint.iacr.org/2020/1533.

[22] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[23] Nicholas Mainardi, Alessandro Barenghi, and Gerardo Pelosi. Privacy preserving substring search protocol with polylogarithmic communication cost. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 297–312, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), sep 2009.

[25] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, feb 1978.

[26] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.

[27] Xiaoqiang Sun, Peng Zhang, Joseph K. Liu, Jianping Yu, and Weixin Xie. Private machine learning classification based on fully homomorphic encryption. *IEEE Transactions on Emerging Topics in Computing*, 8(2):352–364, 2020.

[28] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*, CCSW '13, page 65–76, New York, NY, USA, 2013. Association for Computing Machinery.