석 사 학 위 논 문
Master's Thesis

RLWE 문제 기반 3라운드 정적 그룹키 교환 프로토콜의 구현 연구

Implementation of Static and 3-round Group Key Agreement with RLWE Assumption

2020

한 성 호 (韓 盛 淏 Han, Seongho)

한 국 과 학 기 술 원

Korea Advanced Institute of Science and Technology

석 사 학 위 논 문

RLWE 문제 기반 3라운드 정적 그룹키 교환 프로토콜의 구현 연구

2020

한 성 호

한 국 과 학 기 술 원

전산학부 (정보보호대학원)

RLWE 문제 기반 3라운드 정적 그룹키 교환 프로토콜의 구현 연구

한 성 호

위 논문은 한국과학기술원 석사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2019년 12월 16일

심사위원장 김 광 조 (인)

심 사 위 원 신 인 식 (인)

심 사 위 원 이 주 영 (인)

Implementation of Static and 3-round Group Key Agreement with RLWE Assumption

Seongho Han

Advisor: Kwangjo Kim

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Master of Science in Computer Science (Information Security)

Daejeon, Korea
December 16, 2019

Approved by

Kwangjo Kim
Professor of Computer Science

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

MIS
20174478

한성호. RLWE 문제 기반 3라운드 정적 그룹키 교환 프로토콜의 구현 연구 . 전산학부 (정보보호대학원) . 2020년. 51+iv 쪽. 지도교수: 김광조. (영문 논문)
Seongho Han. Implementation of Static and 3-round Group Key Agreement with RLWE Assumption . School of Computing (Graduate School of Information Security) . 2020. 51+iv pages. Advisor: Kwangjo Kim. (Text in English)

초 록

통신 기술이 발달하면서 한 그룹에서 동일한 키를 공유하는 프로토콜이 점차 중요해지고 있다. 그러나 현재 대부분의 그룹키 교환 방식은 이산대수 문제에 기반하기 때문에 양자 컴퓨터 공격에 취약하다. 이를 해결하기 위해 다양한 양자내성암호가 제안되었다. RLWE문제는 유망한 양자내성암호 중 하나인 격자기반암호에서 사용되는 문제이다. RLWE 문제를 활용하여 여러 그룹키 교환 방식이 제안되었으나, 알려진 바에 의하면 RLWE 문제 기반 그룹키 교환 프로토콜을 구현하고 검증한 적은 없다. 본 석사 논문에서는 RLWE 문제 기반 3라운드 정적 그룹키 교환 프로토콜을 구체화하고, 중재자 기반 네트워크 환경에서 구현한다. 또한 구현된 프로토콜의 성능을 정량적으로 분석한다.

핵 심 낱 말 양자내성암호, RLWE, 그룹키 교환, 분산형

Abstract

With the development of communication technology, the demand for group key agreement protocols is growing. However, most of the group key agreement protocols are based on discrete logarithm problem, which is vulnerable to a quantum adversary. To solve this problem, various quantum-resistant cryptosystems have been proposed. RLWE problem is a kind of problems used in lattice-based cryptography, one of the promising post quantum cryptography. Several group key agreement protocols have been proposed based on RLWE problem. To the best of our knowledge, RLWE-based group key agreement protocol has never been implemented. In this thesis, we instantiate a static and 3-round group key agreement with RLWE assumption and implement the protocol on the arbiter-aided network. We also quantitatively analyze the performance of the implemented protocol.

Keywords Post-quantum cryptosystem, RLWE, Group key agreement, Contributory

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Overview

D-Wave Systems announced the first commercial quantum computer 'D-Wave One' operating on a 128-qubit chipset using quantum annealing in 2011. Recently, Google introduced 72 qubit superconducting quantum chip 'Bristlecone' in 2018 and announced Sycamore with 53 qubit in October 2019 [3]. In the meantime, IBM developed IBM Q 20 Austin with 20 qubits in 2018 and IBM Q 53 with 53 qubits in October 2019. Thus, the era of quantum computing is expected to come within the short term.

Most public key cryptosystems used today are based on the hardness of discrete logarithm problem (DLP) or the difficulty of integer factorization problem (IFP). Breaking the cryptosystems is very hard since both problems are resolved in a computationally infeasible time with classical computers. However, IFP and DLP can be solved within the polynomial-time by Shor's algorithm using a quantum computer [4]. On the other hand, symmetric key cryptosystems such as Advanced Encryption Standards (AES) can be solved in a feasible time using Grover's algorithm [5], which can be used in the data search problem. To be specific, the adversary takes $O(2^n)$ time with a classical computer but takes $O(\sqrt{2^n})$ time with a quantum computer to solve the data search problem. Therefore, researchers are actively studying post-quantum cryptography (PQC) against quantum computer attacks.

There are five categories on PQC: lattice-based, code-based, polynomial-based, hash-based, and isogeny-based. NIST has initiated a process to standardize one or more quantum-resistant public-key cryptographic algorithms in February 2016. The round 2 candidates were announced in January 2019. NIST plans to begin round 3 in 2020 or 2021.

1.2 Motivation

Key exchange protocol between two parties became essential to establish a secure channel that prevents the leak of information after Diffie and Hellman [6] proposed the breakthrough using public key cryptosystem. With the development of communication technologies, the importance of sharing a common group key among multiple parties is growing. Group key agreement (GKA) protocol is a key exchange protocol among multiple parties in which a shared secret is derived from each group member. Several group key exchange protocols [7, 8, 9, 10, 11, 12] have been proposed. Each group member equally contributes to deriving a shared secret in GKA. Every group member has to interact in order to compute the group key, and no entity can predetermine the resulting value. GKA protocol does not require the existence of secure channels between its participants since no secure transfer takes place during processing. However, most of the existing GKA protocols are vulnerable to a quantum adversary because they are based on the difficulty of DLP.

Many researchers have actively been researching quantum-resistant cryptosystem that is secure against a quantum adversary. Regev [13] proposed a lattice-based cryptosystem based on Learning with Error (LWE) problem. As LWE-based cryptosystem has low efficiency, protocols based on ring learning with error (RLWE) have been proposed by Lyubashevsky *et al.* [14]. Then Alkim *et al.* [15] (hereafter referred to as **Newhope**) and Bos *et al.* [16] (hereafter referred to as **BCNS**) implemented RLWE-based 2-party key exchange protocols.

Extending 2-party RLWE-based key exchange protocols, several RLWE-based GKA protocols have been proposed. Ding *et al.* [17] and Apon *et al.* [18] proposed RLWE-based GKA protocols. Choi *et al.* [19] is now in preparation on submitting the paper of dynamic constant-round GKA from RLWE assumption improving Apon *et al.*'s protocol. To the best of our knowledge, there is no practical implementation of RLWE-based GKA with specified parameters. In this thesis, we instantiate Choi *et al.*'s RLWE-based static and 3-round GKA protocol and implement the protocol on the arbiter-aided network. Also, we quantitatively analyze the performance of our implementation.

1.3 Organization

The rest of this thesis is organized as follows: Chapter 2 describes notations and definitions as preliminaries. The related work on the implementation of RLWE-based 2-party key exchange protocols and the algorithms of RLWE-based GKAs is introduced in Chapter 3. In Chapter 4 and Chapter 5, we describe an instantiation of Choi *et al.*'s generic static and 3-round GKA and implementation of the protocol in detail, respectively. Performance evaluation is presented in Chapter 6. Finally, concluding remark and future work are discussed in Chapter 7, respectively.

Chapter 2. Background

2.1 Notations

The following notations are used in this thesis. Table 2.1 describes the notations.

Table 2.1: Notations and Variables

Variables	Description
N	number of peers
P_i	ith peer of a protocol
n	dimension of polynomial ring
$R = \mathbb{Z}[x]/(x^n + 1)$	the ring of integers with dimension n
q	integer modulus
$R_q = \mathbb{Z}_q[x]/(x^n + 1)$	quotient ring of R
ρ	statistical security parameter
λ	computational security parameter
χ	distribution over R
χ_σ	discrete Gaussian distribution with standard deviation σ
$H()$	cryptographic hash function

Statistical security parameter ρ determines the correctness of the protocol, i.e. all peers can share the same session key with probability $1 - 2^{\rho+1}$. We describe the details in Chapter 4.1.

Computational security parameter λ determines the security level of the protocol, i.e. the protocol ensures λ-bit security. We also describe the details in Chapter 4.1.

2.2 Definitions

In this section, we introduce the basic background for the lattice problems, which were introduced in [13, 14], and present the necessary notations.

2.2.1 On the lattice problems

Learning with Errors (LWE) problem

Since Regev [13] proposed LWE, a number of cryptographic schemes based on LWE problem have been introduced. LWE problem is a computational hard problem that is secure against quantum computer attacks. There are two versions of LWE problem: search-LWE and decisional-LWE. As decisional-LWE is used as the primitives for many cryptosystems, we focus on decisional-LWE problem.

Definition 2.2.1. (Decisional-LWE) LWE distribution is defined as follows: For a secret vector $s \in \mathbb{Z}_q^n$, LWE distribution $A_{s,\chi}$ over $\mathbb{Z}_q^n \times \mathbb{Z}_q$ is sampled by choosing $a \in \mathbb{Z}_q^n$ uniformly at random and $e \leftarrow \chi$, and outputting $(a, b = \langle s, a \rangle + e \ mod \ q)$

Given m independent samples $(a_i, b_i) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$ where every sample is distributed from either:

(1) $A_{s,\chi}$ for a uniformly random $s \in \mathbb{Z}_q^n$

(2) The uniform distribution

Distinguish whether samples are from (1) LWE distribution or (2) uniform distribution (with non-negligible advantage)

The difficulty of the decisional-LWE problem is based on the worst-case quantum hardness of two main computational problems on lattices: the decisional version of the shortest vector problem (GapSVP) and the shortest independent vectors problem (SIVP) [13].

Ring Learning with Errors (RLWE) problem

As the cryptographic schemes based on LWE problem are not enough efficient for practical applications, [14] introduced an algebraic variant of LWE called RLWE. There are also two versions of RLWE problem: search-RLWE and decisional-RLWE. We focus on decisional-RLWE, which is widely used for cryptographic schemes.

Definition 2.2.2. (Decisional-RLWE) Let $R = \mathbb{Z}[x]/(x^n + 1)$ be the ring of integers with dimension n and q be an integer modulus. Define $R_q = R/qR \cong \mathbb{Z}_q[x]/(x^n + 1)$ with $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$. Let χ be a distribution over R, and let $s \leftarrow \chi$. Define distribution $O_{\chi,s}$ over $R_q \times R_q$ is sampled by choosing a from uniform distribution on R_q and $e \leftarrow \chi$, and outputting $(a, as + e) \in R_q \times R_q$

Distinguish whether samples are from (1) distribution $O_{\chi,s}$ (2) uniform distribution on $R_q \times R_q$ (with non-negligible advantage)

In definition 2.2.2, the secret s is chosen from the error distribution χ instead of the uniform distribution over R_q as originally defined in [14]. This problem is as hard as the one in which s is chosen uniformly at random [20].

The hardness of the decisional-LWE problem is guaranteed by quantum reduction from worst-case ideal lattice problems to decisional RLWE problem [21].

2.2.2 Reconciliation method

Two parties in RLWE-based key exchange protocol compute very close values in $\mathbb{Z}_q$, not the same value if using a naive approach as shown in the below procedure. To be specific, we can verify that $k_A \neq k_B$ since $k_A = b's + e''' = (as' + e')s + e''' = ass' + e's + e'''$ and $k_B = bs' + e'' = (as + e)s' + e'' = ass' + es' + e''$. There are two ways to resolve this problem: the reconciliation method and error correction code. We focus on the reconciliation method, which is used for group key agreement protocols with RLWE assumption.

Naive RLWE-based Key Exchange		
Alice		Bob
$s, e \xleftarrow{\$} \chi$		$s', e' \xleftarrow{\$} \chi$
$b \leftarrow as + e \in \mathcal{R}_q$	$\xrightarrow{b}$	$b' \leftarrow as' + e' \in \mathcal{R}_q$
$e''' \xleftarrow{\$} \chi$	$\xleftarrow{b'}$	$e'' \xleftarrow{\$} \chi$
$k_A \leftarrow b's + e'''$		$k_B \leftarrow bs' + e''$

There are two major reconciliation methods on RLWE-based key exchange protocols [2]: Ding *et al.*'s method [17] and Peikert's method [22]. We will describe each method in detail.

Ding *et al.*'s method

Ding *et al.* [17] proposed a first reconciliation method in 2012. Ding *et al.* use a robust extractor to ensure the correctness of the protocol. The following is the definition of robust extractor.

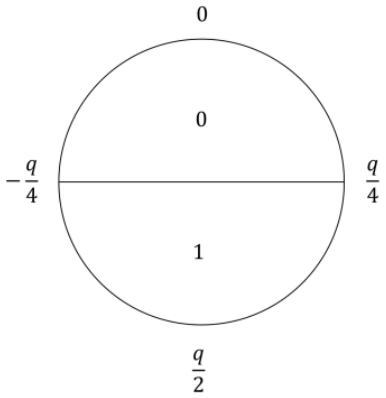

Figure 2.1: Ding *et al.*'s signal function [2]

Definition 2.2.3 (Robust Extractors)**.** An algorithm E is a robust extractor on $\mathbb{Z}_q$ with error tolerance δ with respect to a hint functions **Cha**, if the following holds:

- The deterministic algorithm E takes an input an $x \in \mathbb{Z}_q$ and a signal $w \in \{0, 1\}$, outputs $k = E(x, w) \in \{0, 1\}$.

- The hint algorithm **Cha** takes as input a $y \in \mathbb{Z}_q$ and outputs a signal $w \leftarrow \mathbf{Cha}(y) \in \{0,1\}$.
- For any $x, y \in \mathbb{Z}_q$ such that $x - y$ is even and $|x - y| \leq \delta$, then it holds that $E(x, w) = E(y, w)$, where $w \leftarrow \mathbf{Cha}(y)$.

The errors of x, y in the definition can be set to be multiple of t, where t is a small integer.

The hint function **Cha** which is also called "signal function" is defined in the following.

Definition 2.2.4 (Signal Functions)**.** There are two signal functions used in the robust extractor. For prime $q > 2$, define $S_0(x)$, $S_1(x)$ from $\mathbb{Z}_q$ to $\{0,1\}$.

$$S_0(x) = \begin{cases} 0 & \text{if } x \in [-\lfloor \frac{q}{4} \rfloor, \lfloor \frac{q}{4} \rfloor] \\ 1 & \text{otherwise} \end{cases}; \quad S_1(x) = \begin{cases} 0 & \text{if } x \in [-\lfloor \frac{q}{4} \rfloor + 1, \lfloor \frac{q}{4} \rfloor + 1] \\ 1 & \text{otherwise} \end{cases}$$

For any $y \in \mathbb{Z}_q$, $\mathbf{Cha}(y) = S_b(y)$, where $b \xleftarrow{\$} \{0,1\}$. Figure 2.1 gives intuition on signal functions.

Thus the robust extractor E is defined as: $E(x, w) = (x + w \cdot \frac{q-1}{2} mod\ q)\ mod\ 2$. For any odd $q > 2$, if x is uniformly random in $\mathbb{Z}_q$, then $E(x)$ is uniformly random conditioned on w, where $w \leftarrow Cha(x)$ [17]. Both parties in the protocol compute the shared key using the robust extractor E.

Peikert's method

Peikert [22] gives a variant of error reconciliation method of Ding *et al.*'s protocol. Peikert observed that the agreed-upon bit produced by Ding *et al.*'s protocol is inevitably biased, not uniform. Even one applies post-processing on Ding et al.'s protocol, it reduces the length of the available key. To solve this problem, Peikert directly produces an unbiased key via rounding function, cross-rounding function, and doubling function. Figure 2.2 provides intuition on these functions. We focus on Peikert's method for the reconciliation method in this thesis.

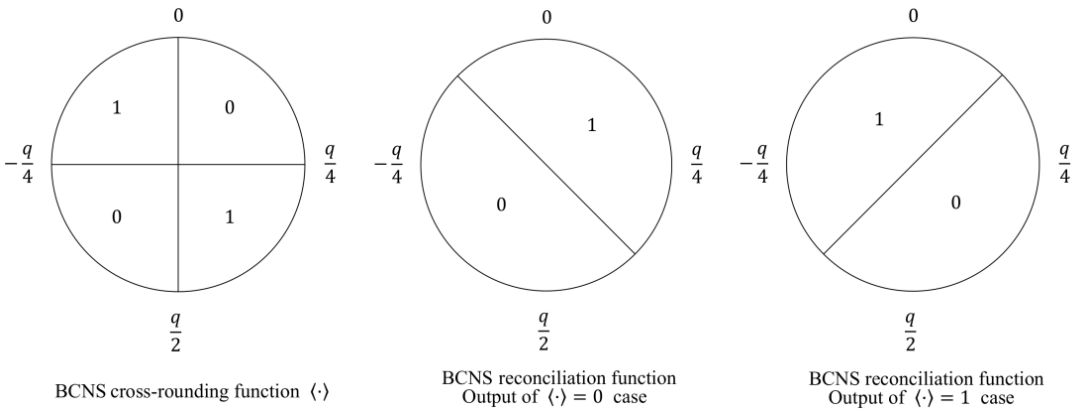

Figure 2.2: BCNS reconciliation function [2]

The followings are notation and concepts required for Peikert's reconciliation function. Let $\lfloor \cdot \rceil : \mathbf{R} \leftarrow \mathbb{Z}$ be the general round function, i.e. $\lfloor x \rceil = z$ for $z \in \mathbb{Z}$ and $x \in [z - 1/2, z + 1/2)$.

Definition 2.2.5. Let q be a positive integer. Define the modular rounding function

$$\lfloor \cdot \rceil_{q,2} : \mathbb{Z}_q \to \mathbb{Z}_2, \quad x \mapsto \lfloor x \rceil_{q,2} = \left\lfloor \frac{2}{q} x \right\rceil \mod 2$$

and the cross-rounding function

$$\langle \cdot \rangle_{q,2} : \mathbb{Z}_q \to \mathbb{Z}_2, \quad x \mapsto \langle \cdot \rangle_{q,2} = \left\lfloor \frac{4}{q} x \right\rceil \mod 2$$

Both functions are extended to elements of quotient ring R_q coefficient-wise: for $f = f_{n-1}X^{n-1} + \cdots + f_1 X + f_0 \in R_q$, define

$$\lfloor f \rceil_{q,2} : (\lfloor f_{n-1} \rceil_{q,2}, \lfloor f_{n-2} \rceil_{q,2}, ..., \lfloor f_0 \rceil_{q,2}),$$
$$\langle f \rangle_{q,2} : (\langle f_{n-1} \rangle_{q,2}, \langle f_{n-2} \rangle_{q,2}, ..., \langle f_0 \rangle_{q,2}).$$

If the modulus q is odd, we have to operate in $\mathbb{Z}_{2q}$ instead of $\mathbb{Z}_q$ to avoid bias in a shared key. As we choose odd q in this thesis, we have to define a randomized doubling function in [22, 16]. Let $\mathsf{dbl}()$: $\mathbb{Z}_q \to \mathbb{Z}_{2q}$, $x \longmapsto \mathsf{dbl}(x) = 2x - e$, where e is sampled from $\{-1, 0, 1\}$ with probabilities $p_{-1} = p_1 = \frac{1}{4}$ and $p_0 = \frac{1}{2}$. The rounding of $dbl(v) \in \mathbb{Z}_{2q}$ for a uniform element $v \in \mathbb{Z}_q$ is uniformly random in $\mathbb{Z}_{2q}$ given its cross-rounding [22].

Define the sets $I_0 = \{-, 1, \cdots, \lfloor \frac{2}{q} \rceil - 1\}$ and $I_0 = \{-\lfloor \frac{q}{2} \rfloor, \cdots, -1\}$. Let $E = [-\frac{q}{4}, \frac{q}{4})$ the reconciliation function $\mathsf{rec}()$ function as follows:

$$\mathsf{rec}(w, b) = \begin{cases} 0 & \text{if } w \in I_b + E \mod 2q \\ 1 & \text{otherwise} \end{cases}$$

From reconciliation function, one can recover the rounding $\lfloor dbl(v) \rceil_{2q,2}$ of a random element $v \in \mathbb{Z}_q$ from an element $w \in \mathbb{Z}_q$ that is close to v and the cross-rounding $\langle dbl(v) \rangle_{2q,2}$. Note that reconciliation of a polynomial in R_q is computed coefficient-wise using the reconciliation function on $\mathbb{Z}_{2q} \times \mathbb{Z}_2$.

As Peikert's method has better performance and security than Ding *et al.*'s method, we use Peikert's method as the reconciliation method.

2.2.3 Sampling from distribution χ

The distribution χ denotes a discrete Gaussian on R_q. As we use $n = 1,024$ being a power of 2 for our implementation, sampling from a discrete Gaussian can be done coefficient-wise using a 1-dimensional discrete Gaussian $D_{\mathbb{Z},\sigma}$ with parameter σ [14]. For each $x \in \mathbb{Z}$, $D_{\mathbb{Z},\sigma}(x) = \frac{1}{S}e^{-x^2/(2\sigma^2)}$ where $S = 1 + 2\sum_{k=1}^{\infty} e^{-k^2/(2\sigma^2)}$.

Since implementing a true discrete Gaussian distribution is impossible in the real world, many researchers have tried to approximate the true discrete Gaussian. There are two basic methods to implement a true discrete Gaussian distribution closely: rejection sampling [23], and the inversion method [24].

Rejection sampling from a set S is done by sample $x \in S$ from some easy distribution such as uniform distribution and then accepting the sample with probability proportional $Pr(x)$. Rejection sampling is used in Zhang *et al.*'s protocol [25]. An averaged M times repetition on sampling is required to compute a shared session key with probability $1 - \frac{1}{M}$. Then rejection sampling is inefficient if we want to guarantee a low failure rate, which is required to guarantee the correctness of RLWE-based GKA. This method is suitable for constrained devices.

On the other hand, **BCNS** uses the inversion method to implement discrete Gaussian distribution. To sample with the inversion method, a table that translates sampling from the discrete Gaussian distribution into sampling form a uniform distribution on a different set is required. The inversion method requires a large precomputed table to store values of the cumulative distribution function of a discrete Gaussian distribution. We can enormously improve the performance if using a precomputed table compared to using rejection sampling under the same failure probability.

Meanwhile, **Newhope** tried to substitute a binomial distribution for discrete Gaussian. **Newhope** provides proof that sampling from a centered binomial distribution accommodates for the little loss in security when $\sigma = \sqrt{8}$. However, using the binomial distribution for a large standard deviation σ is not practical [26]. In this thesis, $\sigma_2 = 2,509,945/\sqrt{2\pi}$ is used for random sampling as described in Chapter 4.1. Thus we do not use a centered binomial distribution.

Consequently we choose to implement the inversion method to sample errors based on **BCNS** source code[1] for fast computation.

[1] https://github.com/dstebila/rlwekex

Chapter 3. Related Work

In this chapter, we introduce the implementation of 2-party RLWE-based key exchange protocols. Then we describe the algorithms of group key agreement protocols from RLWE assumption.

3.1 Implementation of 2-party RLWE-based key exchange

Several implementations of 2-party RLWE-based key exchange protocols have been published. There exist two categories implementing RLWE-based key exchange protocols: based on the reconciliation method or based on the error correction code. Implementation of Ding *et al.*'s two-party RLWE-based key exchange protocol, **BCNS**, and **Newhope** are typical examples for the former. On the other hand, **Round5** [27] and **LAC** [28] use error correction code to agree on a shared session key. As we use reconciliation method for our implementation, we focus on Ding *et al.*'s 2-party protocol, **BCNS**, and **Newhope**. We will describe each protocol in detail.

3.1.1 Ding *et al.*'s 2-party protocol

Ding *et al.* [17] proposed a first 2-party RLWE key exchange protocol (hereafter referred to as **Ding Key Exchange**) in 2012. Then Gao *et al.* presents [2] two protocols called **P30** and **P14** implementing **Ding Key Exchange** in 2017. Protocol 1 describes the algorithms of **P30** and **P14**.

Protocol 1: Ding Key Exchange

Alice		Bob
$s, e \xleftarrow{\$} \chi$		$s', e' \xleftarrow{\$} \chi$
$b \leftarrow as + 2e \in \mathcal{R}_q$	$\xrightarrow{b}$	$b' \leftarrow as' + 2e' \in \mathcal{R}_q$
		$e'' \xleftarrow{\$} \chi$
		$k_B \leftarrow bs' + 2e''$
$e''' \xleftarrow{\$} \chi$	$\xleftarrow{b',w}$	$w \leftarrow \mathsf{Cha}(k_B) \in \{0,1\}^n$
$k_A \leftarrow b's + 2e''' \in \{0,1\}^n$		
$\sigma_A \leftarrow E(k_A, w) \in \{0,1\}^n$		$\sigma_B \leftarrow E(k_B, w) \in \{0,1\}^n$

A signal function **Cha()** and a robust extractor E is described in Chapter 2.2.2.

The parameter choice of **P30** is very close to **BCNS** and that of **P14** is exact same as **Newhope**. Parameters of **P30** are $n = 1,024$, $q = 1,073,479,681$, $\sigma = \frac{8}{2\pi} \approx 3.192$. **P30** uses an inversion method

to sample errors. Meanwhile, parameters of **P14** are $n = 1,024$, $q = 12,289$. **P14** utilize a centered binomial distribution Ψ_k with parameter $k = 16$ for error sampling.

3.1.2 BCNS

Bos *et al.* [16] implemented 2-party RLWE key exchange protocol **BCNS** based on [22] in 2015. Protocol 2 describes the algorithms of **BCNS**.

Protocol 2: BCNS		
Alice		Bob
$s, e \xleftarrow{\$} \chi$		$s', e' \xleftarrow{\$} \chi$
$b \leftarrow as + e \in \mathcal{R}_q$	$\xrightarrow{b}$	$b' \leftarrow as' + e' \in \mathcal{R}_q$
		$e'' \xleftarrow{\$} \chi$
		$v \leftarrow bs' + e'' \in \mathcal{R}_q$
		$\overline{v} \xleftarrow{\$} \mathsf{dbl}(v) \in \mathcal{R}_{2q}$
	$\xleftarrow{b',c}$	$c \leftarrow \langle \overline{v} \rangle_{2q,2} \in \{0,1\}^n$
$k_A \leftarrow \mathsf{rec}(2b's, c) \in \{0,1\}^n$		$k_B \leftarrow \lfloor \overline{v} \rceil_{2q,2} \in \{0,1\}^n$

We describe rounding function, cross-round function, doubling function, and reconciliation function in Chapter 2.2.2.

BCNS use n = 1,024, $q = 2^{32} - 1$, $\sigma = \frac{8}{2\pi} \approx 3.192$ for parameters. They use an inversion method to sample from a discrete Gaussian distribution.

3.1.3 Newhope

Alkim *et al.* [15] suggested 2-party RLWE key exchange protocol **Newhope** in 2016 and implemented the protocol in 2016. Protocol 3 describes the algorithms of **Newhope**.

Protocol 3: Newhope		
Alice		Bob
$\mathsf{seed} \xleftarrow{\$} \{0,1\}^{256}$		
$a \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(\mathsf{seed}))$		
$s, e, \xleftarrow{\$} \Psi_{16}^n$		$s', e', e'' \xleftarrow{\$} \Psi_{16}^n$
	$\xrightarrow{(b,\mathsf{seed})}$	$a \leftarrow \mathsf{Parse}(\mathsf{SHAKE\text{-}128}(\mathsf{seed}))$
		$u \leftarrow as' + e'$
		$v \leftarrow bs' + e''$
$v' \leftarrow us$	$\xleftarrow{(u,r)}$	$r \xleftarrow{\$} \mathsf{HelpRec}(v)$
$\nu \leftarrow \mathsf{Rec}(v', r)$		$\nu \leftarrow \mathsf{Rec}(v, r)$
$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$		$\mu \leftarrow \mathsf{SHA3\text{-}256}(\nu)$

Note that HelpRec() and Rec() functions are defined in the protocol **Newhope**. The followings are detailed description of HelpRec() and Rec(). Let $\mathsf{CVP}_{\hat{D}_4}(\mathbf{x} \in \mathbb{R}^4)$ is that an integer vector $\mathbf{z}$ such that is a closest vector to $\mathbf{x} : \mathbf{x} - \mathbf{Bz} \in \mathcal{V}$. $\mathsf{HelpRec}(\mathbf{x}; b)$ is defined as follows:

$$\mathsf{HelpRec}(\mathbf{x}; b) = \mathsf{CVP}_{\hat{D}_4}\Big(\frac{2^r}{q}(\mathbf{x} + b\mathbf{g})\Big) \mod 2^r$$

where $b \in \{0, 1\}$ is uniformly chosen random bit.

$\mathsf{Decode}(\mathbf{x} \in \mathbb{R}^4/\mathbb{Z}^4)$ is that a bit k such that $k\mathbf{g}$ is a closest vector to $\mathbf{x} + \mathbb{Z}^4 : \mathbf{x} - k\mathbf{g} \in \mathcal{V} + \mathbb{Z}^4$. $\mathsf{Rec}(\mathbf{x}, \mathbf{r})$ is defined as follows:

$$\mathsf{Rec}(\mathbf{x}, \mathbf{r}) := \mathsf{Decode}\Big(\frac{1}{q}\mathbf{x} - \frac{q}{2^r}\mathbf{Br}\Big)$$

Reconciliation method of **Newhope** is highly optimized version of **BCNS**.

They choose $n = 1,024$ and $q = 12,289$ for parameters. The binomial distribution Ψ_{16}^n is used for error sampling.

3.2 RLWE-based group key agreement

Sharing a common group key is required for secure group communication in group-oriented applications. We can derive a group key in two ways called centralized group key distribution and contributory group key agreement (shorten as GKA) [29]. In the GKD, A trusted third party usually has the duties for choosing a session key and distributing it to the group members. Meanwhile, all members collaboratively derive a session key without the support from any trusted third party in the case of GKA.

GKA protocols have several advantages on the security and performance. First of all, GKA has no central authority since each party equally contributes to establishing a shared group key. Then the risk of corruption of central authority is removed. Second, the risk of communication failure is low as each party in GKA manages the only one group key per each session. To be specific, the probability of loss of a session key is low. Finally, GKA protocols provide perfect forward secrecy, i.e. we can protect past sessions from the leakage of future group keys. Thus we only deal with GKA protocols.

There are hundreds of works on secure and efficient GKA protocols. However, most of the existing GKA protocols are based on the hardness of Diffie-Hellman (DH) problem, which is vulnerable to a

Algorithm 1: DXL-mul$(P[0,1,\cdots,N-1], a, \sigma)$

(Round 1) For each peer P_i for $i = 0$ to $N-1$, do the following in parallel.

1. Computes $z_i = as_i + 2e_i^0$ where $s_i, e_i^0 \leftarrow \chi_\sigma$;
2. Each peer P_i sends z_i to peer P_{i+1};

(Round j (j=2,3,...,$N-1$)) For each peer P_i for $i = 0$ to $N-1$, do the following in parallel.

1. Peer P_{i+j-1} computes $z_i^{j-1} = s_{i+j-1} \cdot z_i^{j-2} + 2e_i^{j-1}$ at j-th Round;
2. Each peer P_{i+j-1} sends z_i^{j-1} to P_{i+j};

(Round N) For peer P_0 only.

1. Samples $e_0^{N-1} \leftarrow \chi_\sigma$ and computes $K_0 = s_0 \cdot z_1^{N-2} + 2e_0^{N-1}$;
2. Computes $w \leftarrow \mathbf{Cha}(K_0)$;
3. Calculates session key $SK_0 = E(k_0, w)$;
4. Broadcasts w;

(Key Computation) For each peer P_i for $i = 1$ to $N-1$, do the following in parallel.

1. Samples $e_i^{N-1} \leftarrow \chi_\sigma$ and computes $K_i = s_i \cdot z_{i+1}^{N-2} + 2e_i^{N-1}$;
2. Computes session key $SK_i = E(K_i, w)$;

quantum adversary. To design quantum-resistant GKA protocols, researchers have tried to extend 2-party quantum-resistant key exchange protocols. For instance, Azarderakhsh *et al.* [30] proposes n-party GKA protocol extending 2-party isogeny-based group key exchange protocol. While Ding *et al.* [17] suggests n-party GKA protocol (hereafter referred to as **DXL-mul**) extending **Ding Key Exchange**. In this thesis, we focus on RLWE-based GKA.

To the best of our knowledge, the only two RLWE-based GKA protocols have been proposed and the one GKA protocol with RLWE assumption is now in preparation to submit the paper: **DXL-mul**, Apon *et al.*'s protocol [18] (hereafter referred to as **ADGK19**), and Choi *et al.*'s protocol [19] (hereafter referred to as **CHK**). We will describe each protocol in detail. From this section to the end of our thesis, we will use the term 'peer' who participated in GKA instead of 'party'. Also, peer index number operation is done in mod N.

3.2.1 Ding *et al.*'s multi-party protocol

DXL-mul is a GKA protocol similar to the protocol in [9] except for the underlying problem. **DXL-mul** is based on the hardness of RLWE problem, but the protocol in [9] is based on the hardness of DH problem. Algorithm 1 describes the GKA procedure of **DXL-mul**.

Algorithm 2: $\mathsf{ADGK19}(P[0,1,\cdots,N-1], a, \mathcal{H}, \sigma_1, \sigma_2)$

(Round 1) For each peer P_i for $i = 0$ to $N-1$, do the following in parallel.

1. Computes $z_i = as_i + e_i$ where $s_i, e_i \leftarrow \chi_{\sigma_1}$;
2. Broadcasts z_i;

(Round 2) For each peer P_i for $i = 0$ to $N-1$, do the following in parallel.

1. Peer P_0 samples $e'_0 \leftarrow \chi_{\sigma_2}$. Each of the other peers P_i samples $e'_i \leftarrow \chi_{\sigma_1}$;
2. Each peer P_i broadcasts $X_i = (z_{i+1} - z_{i-1})\, s_i + e'_i$;

(Round 3) For peer P_{N-1} only.

1. Samples $e''_{N-1} \leftarrow \chi_{\sigma_1}$ and computes $b_{N-1} = z_{N-2}Ns_{N-1} + (N-1)\cdot X_{N-1} + (N-2)\cdot X_0 + \cdots + X_{N-3} + e''_{N-1}$;
2. Computes $(\mathsf{rec}, k_{N-1}) = \mathsf{recMsg}(b_{N-1})$;
3. Broadcasts rec and gets the session key as $\mathsf{sk}_{N-1} = \mathcal{H}(k_{N-1})$;

(Key Computation) For peer P_i $(i \neq N-1)$.

1. Computes $b_i = z_{i-1}Ns_i + (N-1)\cdot X_i + (N-2)\cdot X_{i+1} + \cdots + X_{i+N-2}$;
2. Calculates $k_i = \mathsf{recKey}(b_i, \mathsf{rec})$
3. Gets the session key as $\mathsf{sk}_i = \mathcal{H}(k_i)$;

After **(Key Computation)**, all peers can compute an identical session key with overwhelming probability. We can easily observe that **DXL-mul** has not a constant-round from algorithm 1.

3.2.2 Apon *et al.*'s protocol

ADGK19 is a generic RLWE-based 3-round GKA that is derived from Burmester and Desmedt [7] (hereafter referred to as **BD**) protocol by transforming DH problem into RLWE problem. Algorithm 2 describes the key agreement procedure of **ADGK19**

After **(Key Computation)**, all peers can agree on a shared session key with overwhelming probability. As we can see in algorithm 2, **ADGK19** always performs 3-round procedures to derive a shared group key. Any reconciliation method can be used for **ADGK19**.

3.2.3 Choi *et al.*'s protocol

Choi *et al.* prepares to submit a paper of a generic 3-round GKA called **CHK** based on Dutta and Barua [12] (hereafter referred to as **DB**) protocol. **CHK** has better performance than **ADGK19**. Also, **CHK** provides both static and dynamic group key agreement. In this thesis, we only deal with static GKA in **CHK**. Algorithm 3 describes the static group key agreement procedure of **CHK**.

Algorithm 3: $\mathsf{CHK}(P[0,1,\cdots,N-1], a, \mathcal{H}, \sigma_1, \sigma_2)$

(Round 1) For each peer P_i for $i = 0$ to $N-1$, do the following in parallel.

1. Computes $z_i = as_i + e_i$ where $s_i, e_i \leftarrow \chi_{\sigma_1}$;
2. Broadcasts z_i;

(Round 2) For $i = 0$ to $N-1$, do the following in parallel.

1. If $i = 0$, peer P_0 samples $e'_0 \leftarrow \chi_{\sigma_2}$ and otherwise, peer P_i samples $e'_i \leftarrow \chi_{\sigma_1}$;
2. Each peer P_i broadcasts $X_i = (z_{i+1} - z_{i-1})\, s_i + e'_i$;

(Round 3) For peer P_{N-1} only.

1. Samples $e''_{N-1} \leftarrow \chi_{\sigma_1}$ and computes $Y_{N-1,N-1} = X_{N-1} + z_{N-2}s_{N-1} + e''_{N-1}$;
2. For $j = 1$ to $N-1$, computes $Y_{N-1,(N-1)+j} = X_{(N-1)+j} + Y_{N-1,(N-1)+(j-1)}$;
3. Calculates $b_{N-1} = \sum_{j=0}^{N-1} Y_{N-1,(N-1)+j}$;
4. Runs $\mathsf{recMsg}()$ to output $(\mathsf{rec}, k_{N-1}) = \mathsf{recMsg}(b_{N-1})$;
5. Broadcasts rec and gets the session key as $\mathsf{sk}_{N-1} = \mathcal{H}(k_{N-1})$;

(Key Computation) For peer P_i $(i \neq N-1)$.

1. Computes $Y_{i,i} = X_i + z_{i-1}s_i$;
2. For $j = 1$ to $N-1$, computes $Y_{i,i+j} = X_{i+j} + Y_{i,i+(j-1)}$;
3. $b_i = \sum_{j=0}^{N-1} Y_{i,i+j}$;
4. Runs $\mathsf{recKey}()$ to output $k_i = \mathsf{recKey}(b_i, \mathsf{rec})$ and gets the session key as $\mathsf{sk}_i = \mathcal{H}(k_i)$;

After **(Key Computation)**, all peers can derive a shared session key with overwhelming probability. **CHK** always performs 3-round procedures to share a group key as we can see in algorithm 3. We can use any reconciliation method to instantiate **CHK**.

According to algorithm 1,2,3, **CHK** is more practical to implement than **ADGK19** and **DXL-mul**. First of all, **CHK** is similar to **DB**, which is an efficient DH-based GKA protocol. On contrast, **ADGK19** is similar to **BD**. Thus **CHK** is more efficient than **ADGK19**, as **DB** is more efficient than **BD** [12]. The details are described in Chapter 6. Second, the algorithm of **DXL-mul** is similar to that of Bresson *et al.*'s protocol [9] except for cryptographic primitives. However, **DXL-mul** is not practical due to the large complexity on ring polynomial multiplication and error sampling as we described in Chapter 6. Thus we choose to implement **CHK**.

Chapter 4. Instantiation of Choi *et al.*'s Protocol

Since **CHK** is a generic static 3-round GKA protocol from RLWE assumption, we have to instantiate the protocol with specified parameters. In this chapter, we describe the parameter choices required to implement **CHK**. Then, we evaluate the security under the selected parameters.

4.1 Parameter choice

To instantiate **CHK**, we first consider the restrictions for choosing parameters. We should satisfy Theorem 1 and Theorem 2 for correctness and security. Then the performance is considered for practicality.

Theorem 1. For a fixed ρ, and assume that

$$(N-1)N/2 \cdot \sqrt{n}\rho^{3/2}\sigma_1^2 + (N(N+1)/2 + N)\,\sigma_1 + (N-2)\sigma_2 \leq \beta_{\mathsf{Rec}}.$$

Then all participants in a group have the same key except with probability at most $2^{-\rho+1}$.

Theorem 1 provides the correctness of the GKA protocol. i.e. all peers who participate in **CHK** can agree on a shared group key with probability $1 - 2^{\rho+1}$.

Note that $\beta_{\mathsf{Rec}} = q/8$ since we use Peikert's reconciliation method [16].

Theorem 2. (Simplified version) For basic GKA protocol CHK, $2N\sqrt{n}\lambda^{3/2}\sigma_1^2 + (N-1)\sigma_1 \leq \beta_{\mathsf{Rényi}}$ and $\sigma_2 = \Omega\left(\beta_{\mathsf{Rényi}}\sqrt{n/\log\lambda}\right) \leq q$. Then the protocol has λ-bit security against a classical adversary.

Theorem 2 provides the security level of the GKA protocol. Note that the security model to measure the security level of GKA is proposed by [10]. We can ensure λ-bit security level of the protocol from Theorem 2.

Considering theorem 1 and theorem 2, we first set a statistical security parameter $\rho = 256$ related to correctness and a computational security parameter $\lambda = 64$ related to the security level. We initially tried to implement the protocol with $\lambda = 256$, which is the recommended security level from NIST, but failed when we generate a precomputed table. Thus $\lambda = 64$ is selected for a practical reason. If generating a precomputed table having more than 20,000,000 indices is possible, λ can be set to 128 or over.

Table 4.1: Parameter choice

ρ	λ	N	n	q	σ_1	σ_2	β_{rec}	$\beta_{\text{Rényi}}$
256	64	3	1,024	$2^{32}-1$	$8/\sqrt{2\pi}$	$2,509,945/\sqrt{2\pi}$	$2^{29}-1$	1,001,323

Then we choose adequate values for dimension n and modulus q to implement RLWE-based protocol more efficient. Generally, n can be a power of 2 as cyclotomic rings are used for RLWE. We set $n = 1,024$ which is used for many RLWE-based protocols such as **Newhope**, **BCNS**.

q can be any integers since decisional RLWE is hard over a prime cyclotomic ring with any modulus [21]. Then we can use any modulus q for better performance. According to [27], general choices of the modulus q are (1) A number theoretical transform (NTT) friendly prime number, such as 12,289 in **Newhope** [15], (2) A composite number that fits in a data type for modern computers, such as $2^{32}-1$ in **BCNS** [16], (3) A power of 2 that makes modulo operations and integer multiplications efficient, such as 211 in NTRUEncrypt [31]. q should be sufficiently large due to Theorem 2. Thus we choose $q = 2^{32}-1$, which is used in **BCNS**.

Finally, we calculate σ_2, N, $\beta_{\text{Rényi}}$ under the determined parameters. Note that $\sigma_1 = 8/\sqrt{2\pi}$ since we extend **BCNS** protocol. $N = 3$, $\sigma_2 = 2,509,945/\sqrt{2\pi}$, and $\beta_{\text{Rényi}} = 1,001,323$ are chosen for a practical reason.

Parameters used for our implementation are summarized in Table 4.1.

4.2 Security evaluation

We analyze the security of our implementation from two perspectives. We evaluate the security of **CHK** with our parameter settings based on Theorem 2. From theorem 2, we can observe that **CHK** has $\lambda = 64$ level bit of security.

On the other hand, we consider cryptanalytic attacks on RLWE problem. Even though there are many algorithms to attack RLWE, many of those are inappropriate for our settings [15]. Thus we consider only two BKZ attacks [32, 33], usually referred to as primal and dual attacks. The security level of RLWE is estimated with the approach presented in [34]. According to [15], **BCNS** provides 86-bit classical security and 78-bit quantum security against primal attack. Likewise, **BCNS** provides 86-bit classical security and 78-bit quantum security against dual attack [15]. These security levels are applied to our protocol since we extend **BCNS** protocol.

Therefore, since we assume that an adversary attacks the weakest part of the protocol for conservative estimation, the classical security level of our protocol is 64-bit, and the quantum security level is 64-bit.

Chapter 5. Implementation of Choi *et al.*'s Protocol

In this chapter, we provide details on our implementation of **CHK** with parameters described in Chapter 4.

5.1 Ring polynomial arithmetic

Polynomial arithmetic in the cyclotomic ring $R_q = \mathbb{Z}_q[x]/(\Phi_{2^{k+1}}(x))$ is used for RLWE-based GKA where $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$ and $\Phi_{2^{k+1}}(x) = X^{2^l}+1$ is the 2^{l+1}-th cyclotomic polynomial. As we choose $q = 2^{32}-1$, 2 is invertible in the ring $\mathbb{Z}_q$. Multiplication of two ring polynomials in R_q can be done by computing the discrete Fourier transform via fast Fourier transform (FFT) [35] algorithms, which are used in **BCNS**. Then we use a Nussbaumer's approach [36] as summarized by Knuth [37]. We can efficiently compute the modular reduction from Nussbaumer's approach in case of the degree $n = 2^k$ for some integer k and the modulus $q = 2^{32} - 1$. The strategy we follow for the modular arithmetic is described in [38].

Most of the basic structures on polynomial arithmetic follow the implementation of **BCNS**. We further implement a subtraction algorithm for polynomial arithmetic, which is used in **CHK** round 2.

5.2 Sampling from a discrete Gaussian distribution

The inversion method adopted in **BCNS** is used in our implementation for sampling elements in the ring R_q. We independently sample each of the $n = 1,024$ coefficients of an element in R_q from a one-dimensional discrete Gaussian. For a one-dimensional discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ centered at $\mu = 0$ with standard deviation σ, the probability of sample $x \in \mathbb{Z}$ from a random variable is $D_{\mathbb{Z},\sigma}(x) = \frac{1}{S}e^{-x^2/(2\sigma^2)}$ where $S = \Sigma_{k=-\infty}^{\infty} e^{-k^2/(2\sigma^2)}$. In our parameter settings, we use two σ values: $\sigma_1 = 8/\sqrt{2\pi}$, $\sigma_2 = 2,509,945/\sqrt{2\pi}$. Then we have approximate values $S_1 = 8$ and $S_2 = 2,509,945$.

To efficiently implement the inversion method, we use a precomputed lookup table T_1 and T_2 of size 52 and 7,707,672, respectively. T_1 and T_2 are used to sample from a distribution $D_{\mathbb{Z},\sigma_1}$ and $D_{\mathbb{Z},\sigma_2}$, respectively. The elements in each T_i $(i = 1, 2)$ are set as follows: $T_i[j] = \lfloor 2^{c_i} \cdot (\frac{1}{S_i} + 2\Sigma_{x=1}^{j} D_{\mathbb{Z},\sigma_i}(x)) \rfloor$ where $c_1 = 192$ and $c_2 = 64$. All table elements in T_1, T_2 are integers in $[2^{189}, 2^{192}]$, $[7349461471749, 2^{64}]$, respectively. Note that $T_i[j+1] > T_i[j]$ holds for $i = 1, 2$ and $0 \le j \le maxind_i - 1$ where $maxind_i$ is

the size of T_i.

The elements and the size of T_1, T_2 are derived from the lemmas in [39]. As we set $\sigma_1 = 8/\sqrt{2\pi}$, we take T_1 from **BCNS** implementation. Then we only need to compute a table T_2. The followings are the way to obtain T_2. Let D'' is the distribution on Z^n corresponding to taking n independent samples using a precomputed lookup table and let $D_{\mathbb{Z}^n,\sigma}$ be the true discrete Gaussian distribution on $\mathbb{Z}^n$. From the lemma in [39], the statistical difference $\Delta(D'', D_{\mathbb{Z}^n,\sigma})$ of the two distributions is bounded by $2^{-k} + 2mt\sigma\epsilon$. As we take the parameters from **BCNS** implementation, we obtain $k = 129$, $m = 1024$, $t = 42$, $\sigma_2 = 2,509,945/\sqrt{2\pi}$ and $\epsilon = 2^{-64}$. Note that small $\epsilon = 2^{-64}$ is chosen for a practical reason. We first tried to derive T_2 with $\lambda = 256$ and $\epsilon = 2^{-192}$. We calculated that σ_2 is at least $20,079,439/\sqrt{2\pi}$ and the size of T_2 is at least $122,154,285$ since the size of T_2 is equal to the smallest integer x that satisfies $D_{\mathbb{Z},\sigma}(x) \geq \epsilon$. However, the process of table generation was killed in the middle under these parameter settings. Then we tried to modify ϵ and λ. From several attempts, we find that the size of a precomputed lookup table becomes large when we choose small ϵ or large λ. Thus we can verify the trade-off between the accuracy of error sampling and performance. Finally, we set $\lambda = 64$ and $\epsilon = 64$. In our parameter choice, the statistical difference of sampling distribution using a table T_2 and the theoretical distribution is less than 2^{-27}.

To obtain a precomputed lookup table T_2 in the real world, we should preserve 64 decimal places for each element. To achieve this, we use Taylor series. As a result, we successfully computed T_2.

The error sampling e_i from χ_{σ_i} using the precomputed tables is done as follows: Let $e_i = \Sigma_{j=0}^{1023} e_{i,j} x^j$. To sample from χ_{σ_i}, we independently generate a c_i-bit integer $v_{i,j}$ uniformly at random and find the smallest integer index $ind_{i,j} \in [0, maxind_i - 1]$ such that $v_{i,j} < T[ind_{i,j}]$ for each $i = 1, 2$ and $j = 0, ..., 1023$. Then one additional random bit is generated to decide the $sign_{i,j} \in \{-1, 1\}$, and return the j-th coefficient of e_i as $e_{i,j} \leftarrow sign_{i,j} \cdot ind_{i,j}$.

There are two approaches to find the smallest integer index. The first approach increases the index by one until the generated random element becomes smaller than the element in a precomputed table. We denote this approach as *non-constant-time*. On the other hand, *constant-time* approach is implemented in our protocol. We loads every table element and creates a mask based on whether the input is bigger than each accessed element in *constant-time* approach. *Constant-time* approach provides resistance against timing attack such as cache attacks [40] using the information indicating whether or

not the elements in the precomputed table is already loaded.

Note that we need a large amount of random data as every operation of random sampling requires 1,024 random strings of c_i bits. For obtaining a lot of random numbers, OpenSSL's **RAND bytes** function is used to generate a 256-bit seed, then **AES** in counter mode is applied as the PRNG function. For each ring element, we reseed the PRNG.

5.3 Network configuration

We configure the network using socket programming. From our parameter settings, 3 peers participate in the GKA protocol. Each peer broadcasts the computed intermediate values to all other peers in the algorithm of **CHK**. However, no one can act as a bulletin board due to the feature of socket programming. Thus we employ an arbiter-peer network for communication.

The roles of an arbiter are as followings: 1) participates in GKA 2) receive public information from a peer such as z_i or X_i in **CHK** 3) collect received data 4) broadcasts collected data to other peers 5) computes **rec** and broadcasts **rec** to other peers. We set P_2 as an arbiter in our implementation. Note that node number is randomly assigned.

The roles of a peer are as followings: 1) participates in GKA 2) calculate and send public information 3) calculate session key from received data. Peers (P_0, P_1) can calculate session keys with received values from an arbiter.

The entire source code of an arbiter and a peer are described in Appendices A and B, respectively. The full reference source code is available in our github address[2].

[2]https://github.com/hansh17/DRAGKE

Chapter 6. Performance Evaluation

In this chapter, we describe experiments on our implementation. First of all, the experimental setup is introduced. Then we explain four experiments that measure the correctness and the performance of the implementation.

6.1 Experimental setup

The experimental environment is as follows: Intel(R) CPU i5-8250, RAM 8GB, and OS Ubuntu v16.04.5 LTS. As we use a virtual machine, only a partial power of the computer is utilized for performance evaluation. We use 2 processors with 100% execution cap for CPU and 4GB for RAM. gcc v5.4.0 is used as compiler. $-O3$ optimizations are used when compiling.

6.2 Experiments

We performed four experiments on our protocol. We measure the correctness of our protocol in experiment 1. Then the runtime for our implementation is evaluated in experiment 2. In experiment 3, we theoretically compare the complexity of **CHK** with other RLWE-based GKAs. Finally, the comparison between our implementation and DH-based GKA is performed in experiment 4.

6.2.1 Experiment 1

In experiment 1, we measured the correctness of **CHK**. As **CHK** has never been implemented, we have to check whether **CHK** works correctly or not. According to Theorem 1, all peers in GKA should agree on a shared key except $2^{-\rho+1}$. In our parameter settings, the failure probability of a group key agreement is 2^{-257}. We simulated 1,000 times to verify our implementation since 1,000 trials are enough to measure the accuracy of the protocol. The algorithm of checking the correctness is as follows:

1) calculate each P_i's session key for $i = 0, 1, 2$.

2) Check whether {session key of P_i} = {session key of P_j} $(0 \leq i < j \leq 2)$.

The test was performed on a local environment. We did not consider the reliability of the network. If the network environment is unstable, the failure rate will increase.

Table 6.1: Average runtime of single operations

	non-constant-time	constant-time
Operation	runtime(μsec)	runtime(μsec)
Random sampling from χ_{σ_1}	137	589
Random sampling from χ_{σ_2}	816,431	14,146,619
Ring polynomial addition	0	0
Ring polynomial multiplication	200	274

Table 6.2: Average runtime of each function in GKA

	non-constant-time	constant-time
Operation	runtime(μsec)	runtime(μsec)
Compute z_i	524	936
Compute X_0	844,847	11,882,138
Compute X_i	418	675
Compute reconcile (arbiter)	432	678
Compute session key (peer)	236	237

As an experimental result, no failure occurred on all the 1,000 tests. Thus we verified that both the protocol **CHK** and our implementation works correctly.

6.2.2 Experiment 2

We evaluated the runtime of single operations and functions in GKA in experiment 2. The runtime of each operation is measured 200 times and averaged. Table 6.1 shows the performance of single operations. The term *non-constant-time* and *constant-time* is described in Chapter 5.2.

As we can see in Table 6.1, *constant-time* implementation has a much longer runtime than *non-constant-time* implementation. Thus we found the trade-off between the performance and the security.

On the other hand, random sampling from χ_{σ_2} takes much longer time than random sampling from χ_{σ_1}. This is derived from the difference in size of $\sigma_1 = 8/\sqrt{2\pi}$ and $\sigma_2 = 2,509,945/\sqrt{2\pi}$. Ring polynomial addition running time is almost 0, but ring polynomial multiplication running time is 280 μsec. Therefore ring polynomial multiplication and random sampling is an important factor for performance.

We also evaluated the runtime of each function in GKA. We measure the time of computing z_i, X_i, rec, and session key. These values are described in Chapter 3.2. As we can see in Table 6.2, computing X_0 takes much longer time than computing X_i due to sampling from χ_{σ_2}. Note that only P_0 samples from χ_{σ_2}. Therefore random sampling from χ_{σ_2} is the most important factor in measuring performance of **CHK**.

Table 6.3: Time complexity of **CHK**, **ADGK19**, and **DXL-mul**

Time Complexity of operations	**CHK**	**ADGK19**	**DXL-mul**
# of rounds	**3**	**3**	N
# of ring polynomial addition operations	$2N^2 + 3N + 1$	$(N+1)^2$	N^2
# of ring polynomial multiplication operations	**3N**	**3N**	N^2
# of secret samplings	N	N	N
# of error samplings	**2N+1**	**2N+1**	N^2

# denotes the number.
Bold texts denote that less computation is required for each operation than other protocols. We exclude the number of ring polynomial addition operations because of its low importance on computation time.

6.2.3 Experiment 3

We theoretically analyze and compare the performance of **CHK**, **ADGK19**, and **DXL-mul** in experiment 3. As there are no implementations of **ADGK19** and **DXL-mul**, we compare three protocols from the perspective of time complexity of the number of rounds, the number of ring polynomial addition operations, the number of ring polynomial multiplication operations, and the number of secret sampling and error sampling. The number of ring polynomial addition operations is counted whenever peer calculates $+$ or $-$, and the number of ring polynomial multiplication operations is calculated whenever peer calculates $\cdot$. Secret sampling and error sampling are counted once per each sampling. Ring polynomial multiplication and samplings are the most critical factors in measuring the efficiency of the protocol as we described in experiment 2.

Table 6.3 shows the number of each operation of **CHK**, **ADGK19** and **DXL-mul** as functions of the number of peers N. We can observe that the number of multiplications and error samplings of **CHK** and **ADGK19** are less than that of **DXL-mul** as N increases. Thus we can expect that **CHK** and **ADGK19** outperforms **DXL-mul** even though **DXL-mul** performs less polynomial addition operations than other protocols. On the other hand, **CHK** is much more efficient than **ADGK19** since **ADGK19** performs scalar multiplication on the ring polynomial, but **CHK** does not. Note that we do not consider scalar multiplication in Table 6.3.

Therefore, as the number of peers N in GKA increases, **CHK** provides better performance than **ADGK19** and **DXL-mul**.

Table 6.4: Performance evaluation of lattice-based cryptographic schemes [1]

	BCNS	**Newhope**	**Frodo**
Security level (bit)	163	206	142
Runtime(ms)	2.774	0.31	2.6
Payload (bytes)	8,320	3,872	22,673

Table 6.5: Performance evaluation of our protocol and **DB**

	non-constant-time	constant-time	**DB**
Security level (bit)	64	64	86
Total runtime (μsec)	846,039	11,883,989	2,040
Payload (bytes)	24,960	24,960	768
Secure against quantum adversary	O	O	X

6.2.4 Experiment 4

We compared the performance of our implementation and DH-based GKA in experiment 4 to check the utility of RLWE-based GKA. We chose **DB** as DH-based GKA. Note that **CHK** and **DB** has similar procedure except underlying cryptographic problem.

We first surveyed the performance of two-party lattice-based schemes to check whether **BCNS** is the optimized implementation. Then analyze the performance of **BCNS**, **Newhope**, and **Frodo** [1].

We observed that **BCNS** is not an optimized implementation of RLWE from Table 6.4. RLWE-based protocol outperform LWE-based protocol in theoretically, but **BCNS** has similar performance as **Frodo**. Since our implementation extends **BCNS**, we found that our protocol is not optimized. Improving **CHK** via optimization remains an open problem.

To compare the total runtime of **CHK** and **DB**, we implemented **DB** using *openssl* library. We also adopted an arbiter-peer network to implement **DB**. We measured the whole procedure of GKA, i.e., the protocol ends when all peers calculate a shared group key. Since the calculation is performed in parallel, we add the longest time to calculate a public value in each round.

As we can see in Table 6.5, our {non-constant-time, constant-time} implementation takes {$\times$415, $\times$5826} longer than **DB** to share a group key even though our protocol has lower security. Also, the payloads of our protocol are much larger than that of **DB**. RLWE-based GKA is much slower than DH-based GKA despite using the fastest method for error sampling. Consequently, we can infer that trade-off between quantum-resistance and performance exists.

Chapter 7. Concluding Remark

This thesis tries the first practical implementation of RLWE-based GKA protocol. We instantiate Choi *et al.*'s generic protocol and implement the protocol on an arbiter-aided network for a practical reason. For fast implementation, we use FFT algorithms for ring-polynomial operations, which are efficient in our parameter settings. Also, Peikert's reconciliation method is used to remove bias from a shared key. Moreover, Error sampling is done by the inversion method, which is suitable for large modulus q. Consequently, we extend **BCNS** protocol to implement **CHK**, then evaluate the security of implemented protocol. Besides, we evaluate the performance of the implemented protocol and compare it with other GKA protocols. We verify that our implementation and **CHK** works correctly. Also, we check the trade-off between the performance and timing attack resistance. We find the fact that the running time of sampling from discrete Gaussian with a large standard deviation is much longer than other operations. Furthermore, we theoretically verify that **CHK** has better performance than **ADGK19** and **DXL-mul**. Finally, we observe that **CHK** is much slower than **DB** to derive a session key.

As future work, we can improve our implementation in several directions. First of all, we will improve the security level from 64 to 128 by generating a larger precomputed table. Second, we can reduce the time required for the error sampling via optimization. Third, we will integrate our protocol into TLS v1.3 instead of TLS v1.2 by upgrading OpenSSL v1.0.2g to v1.1.1d which is the latest released version at this point. We can improve security and speed using TLS v1.3. Finally, we will implement our protocol in dynamic settings. In the real world, group members who participate in GKA protocol changes frequently. Thus this improvement will make our protocol more practical.

Bibliography

[1] J. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the ring! practical, quantum-secure key exchange from LWE," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1006–1018, ACM, 2016.

[2] X. Gao, J. Ding, R. Saraswathy, L. Li, and J. Liu, "Comparison analysis and efficient implementation of reconciliation-based RLWE key exchange protocol.," *IACR Cryptology ePrint Archive*, vol. 2017, p. 1178, 2017.

[3] E. Gibney, "Hello quantum world! Google publishes landmark quantum supremacy claim.," *Nature*, vol. 574, no. 7779, p. 461, 2019.

[4] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings, Annual Symposium on Foundations of Computer Science–FOCS'94*, pp. 124–134, IEEE, 1994.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, ACM, 1996.

[6] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

[7] M. Burmester and Y. Desmedt, "A secure and efficient conference key distribution system," in *Workshop on the Theory and Application of of Cryptographic Techniques*, pp. 275–286, Springer, 1994.

[8] M. Steiner, G. Tsudik, and M. Waidner, "Key agreement in dynamic peer groups," *IEEE Transactions on Parallel and Distributed Systems*, vol. 11, no. 8, pp. 769–780, 2000.

[9] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater, "Provably authenticated group Diffie-Hellman key exchange," in *Proceedings of the 8th ACM conference on Computer and Communications Security*, pp. 255–264, ACM, 2001.

[10] E. Bresson, O. Chevassut, and D. Pointcheval, "Provably authenticated group Diffie-Hellman key exchange—the dynamic case," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 290–309, Springer, 2001.

[11] E. Bresson, O. Chevassut, and D. Pointcheval, "Dynamic group Diffie-Hellman key exchange under standard assumptions," in *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 321–336, Springer, 2002.

[12] R. Dutta and R. Barua, "Constant round dynamic group key agreement," in *International Conference on Information Security*, pp. 74–88, Springer, 2005.

[13] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, p. 34, 2009.

[14] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," *Journal of the ACM (JACM)*, vol. 60, no. 6, p. 43, 2013.

[15] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange—a new hope," in *25th USENIX Security Symposium (USENIX Security 16)*, pp. 327–343, 2016.

[16] J. W. Bos, C. Costello, M. Naehrig, and D. Stebila, "Post-quantum key exchange for the TLS protocol from the ring learning with errors problem," in *2015 IEEE Symposium on Security and Privacy*, pp. 553–570, IEEE, 2015.

[17] J. Ding, X. Xie, and X. Lin, "A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem.," *IACR Cryptology ePrint Archive*, vol. 2012, p. 688, 2012.

[18] D. Apon, D. Dachman-Soled, H. Gong, and J. Katz, "Constant-Round Group Key Exchange from the Ring-LWE Assumption.," *IACR Cryptology ePrint Archive*, vol. 2019, p. 398, 2019.

[19] R. Choi, D. Hong, and K. Kim, "Constant-round Dynamic Group Key Exchange from RLWE Assumption," *private communication*, 2019.

[20] V. Lyubashevsky, C. Peikert, and O. Regev, "A toolkit for ring-LWE cryptography," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 35–54, Springer, 2013.

[21] C. Peikert, O. Regev, and N. Stephens-Davidowitz, "Pseudorandomness of ring-LWE for any ring and modulus," in *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 461–473, ACM, 2017.

[22] C. Peikert, "Lattice cryptography for the internet," in *international workshop on post-quantum cryptography*, pp. 197–219, Springer, 2014.

[23] V. Lyubashevsky, "Lattice signatures without trapdoors," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 738–755, Springer, 2012.

[24] L. Devroye, "Sample-based non-uniform random variate generation," in *Proceedings of the 18th conference on Winter simulation*, pp. 260–265, ACM, 1986.

[25] J. Zhang, Z. Zhang, J. Ding, M. Snook, and Ö. Dagdelen, "Authenticated key exchange from ideal lattices," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 719–751, Springer, 2015.

[26] H. Nejatollahi, N. Dutt, S. Ray, F. Regazzoni, I. Banerjee, and R. Cammarota, "Post-quantum lattice-based cryptography implementations: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, p. 129, 2019.

[27] H. Baan, S. Bhattacharya, S. R. Fluhrer, O. Garcia-Morchon, T. Laarhoven, R. Rietman, M.-J. O. Saarinen, L. Tolhuizen, and Z. Zhang, "Round5: Compact and Fast Post-Quantum Public-Key Encryption.," *IACR Cryptology ePrint Archive*, vol. 2019, p. 90, 2019.

[28] X. Lu, Y. Liu, Z. Zhang, D. Jia, H. Xue, J. He, B. Li, K. Wang, Z. Liu, and H. Yang, "LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus.," *IACR Cryptology ePrint Archive*, vol. 2018, p. 1009, 2018.

[29] H. Xiong, Y. Wu, and Z. Lu, "A Survey of Group Key Agreement Protocols with Constant Rounds," *ACM Computing Surveys (CSUR)*, vol. 52, no. 3, p. 57, 2019.

[30] R. Azarderakhsh, A. Jalali, D. Jao, and V. Soukharev, "Practical Supersingular Isogeny Group Key Agreement," *IACR Cryptology ePrint Archive*, vol. 2019, p. 330, 2019.

[31] J. Hoffstein, J. Pipher, J. M. Schanck, J. H. Silverman, W. Whyte, and Z. Zhang, "Choosing parameters for NTRUEncrypt," in *Cryptographers' Track at the RSA Conference*, pp. 3–18, Springer, 2017.

[32] Y. Chen and P. Q. Nguyen, "BKZ 2.0: Better lattice security estimates," in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 1–20, Springer, 2011.

[33] C.-P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Mathematical programming*, vol. 66, no. 1-3, pp. 181–199, 1994.

[34] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[35] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[36] H. Nussbaumer, "Fast polynomial transform algorithms for digital convolution," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 28, no. 2, pp. 205–215, 1980.

[37] D. E. Knuth, "The art of computer programming. volume 1: Fundamental algorithms. volume 2: Seminumerical algorithms," *Bull. Amer. Math. Soc*, 1997.

[38] J. W. Bos, C. Costello, H. Hisil, and K. Lauter, "Fast cryptography in genus 2," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 194–210, Springer, 2013.

[39] N. C. Dwarakanath and S. D. Galbraith, "Sampling from discrete gaussians for lattice-based cryptography on a constrained device," *Applicable Algebra in Engineering, Communication and Computing*, vol. 25, no. 3, pp. 159–180, 2014.

[40] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," in *Cryptographers' track at the RSA conference*, pp. 1–20, Springer, 2006.

Appendices

A Source code of arbiter

```
#include <arpa/inet.h>
#include <getopt.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <fcntl.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include "fft.h"
#include "rlwe.h"
#include "rlwe_a.h"
#include "rlwe_rand.h"

#define MAX_PEER 6
#define POLY_LEN 1024
#define KEY_LEN  16
#define HASH_LEN 129

bool check_augmented_pub_keys[MAX_PEER];
bool option_check[4][MAX_PEER];
uint32_t sec_keys[MAX_PEER][POLY_LEN];
uint32_t pub_keys[MAX_PEER][POLY_LEN];
uint32_t augmented_pub_keys[MAX_PEER][POLY_LEN];
uint64_t session_keys[MAX_PEER][KEY_LEN];
unsigned char hashed_keys[MAX_PEER][HASH_LEN];
uint64_t reconcile[KEY_LEN];

int calculate_pubkey(int peer, const uint32_t *a, uint32_t s[1024], FFT_CTX *
    ctx);
int calculate_augmented_pubkey(int peer, int num_peer, uint32_t s[1024],
    FFT_CTX *ctx);
int calculate_reconcile(int num_peer, uint32_t s[1024], uint64_t rec[16],
    uint64_t k[16], unsigned char hk[129], FFT_CTX *ctx);
void run_server(int num_peer, int server_port);
```

```
int calculate_pubkey(int peer, const uint32_t *a, uint32_t s[1024], FFT_CTX *
    ctx) // calculate z_i in Round 1 (i=peer)
{
  if (peer < 0 || peer > MAX_PEER)
  {
            printf("peer range error!\n");
            return -1;
      }
  int ret;
  uint32_t e[1024];
  RAND_CTX rand_ctx;
  ret = RAND_CHOICE_init(&rand_ctx); // initialize seed
  if (!ret)
  {
    return ret;
  }

#if CONSTANT_TIME
  rlwe_sample_ct(s, &rand_ctx); // sample s_i (constant)
  rlwe_sample_ct(e, &rand_ctx); // sample e_i (constant)
#else
  rlwe_sample(s, &rand_ctx); // sample s_i (non-constant)
  rlwe_sample(e, &rand_ctx); // sample e_i (non-constant)
#endif

  uint32_t tmp[1024];
  rlwe_key_gen(tmp, a, s, e, ctx);  // compute tmp=as_i+e_i
  for(int t=0; t<1024; t++)
  {
    pub_keys[peer][t]=tmp[t]; // save tmp as pub_keys[peer]
  }

  rlwe_memset_volatile(e, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp, 0, 1024 * sizeof(uint32_t));
  RAND_CHOICE_cleanup(&rand_ctx);
  return ret;
}

int calculate_augmented_pubkey(int peer, int num_peer, uint32_t s[1024],
    FFT_CTX *ctx) // calculate X_i in Round 2 (i=peer)
{
  int ret;
  uint32_t e[1024];
  RAND_CTX rand_ctx;
  ret = RAND_CHOICE_init(&rand_ctx); // initialize seed
  if (!ret)
  {
    return ret;
```

```
  }

  uint32_t result[1024]={0,};
  uint32_t tmp1[1024];
  uint32_t tmp2[1024];
  if (peer==num_peer-1) // if i = N-1
  {
#if CONSTANT_TIME
    rlwe_sample_ct(e, &rand_ctx); // sample e'_{N-1} (constant)
#else
    rlwe_sample(e, &rand_ctx); // sample e'_{N-1} (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[0][t]; // tmp1 = z_0
      tmp2[t]=pub_keys[peer-1][t]; // tmp2 = z_{N-2}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_0 - z_{N-2}
    FFT_mul(result, result, s, ctx);  // result = (z_0 - z_{N-2}) * s_{N-1}
    FFT_add(result, result, e); // result = (z_0 - z_{N-2}) * s_{N-1} + e'_{N
   -1}
  }
  else if (peer==0) // if i = 0
  {
#if CONSTANT_TIME
    rlwe_sample2_ct(e, &rand_ctx); // sample e'_0 from sigma2 (constant)
#else
    rlwe_sample2(e, &rand_ctx); // sample e'_0 from sigma2 (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[peer+1][t]; // tmp1 = z_1
      tmp2[t]=pub_keys[num_peer-1][t]; // tmp2 = z_{N-1}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_1 - z_{N-1}
    FFT_mul(result, result, s, ctx); // result = (z_1 - z_{N-1}) * s_0
    FFT_add(result, result, e); // result = (z_1 - z_{N-1}) * s_0 + e'_0
  }
  else // if 1<= i <= N-2
  {
#if CONSTANT_TIME
    rlwe_sample_ct(e, &rand_ctx); // sample e'_i (constant)
#else
    rlwe_sample(e, &rand_ctx); // sample e'_i (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[peer+1][t]; // tmp1= z_{i+1}
```

```
      tmp2[t]=pub_keys[peer-1][t]; // tmp2 = z_{i-1}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_{i+1} - z_{i-1}
    FFT_mul(result, result, s, ctx); // result = (z_{i+1} - z_{i-1}) * s_i
    FFT_add(result, result, e); // result = (z_{i+1} - z_{i-1}) * s_i + e'_i
  }

  for(int t=0; t<1024; t++)
  {
    augmented_pub_keys[peer][t]=result[t]; // save result as augmented_pub_keys
    [peer]
  }

  rlwe_memset_volatile(result, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp1, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp2, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(e, 0, 1024 * sizeof(uint32_t));
  RAND_CHOICE_cleanup(&rand_ctx);
  return ret;
}

void sha512_session_key(uint64_t *in, char outputBuffer[129]) // calculate hash
     value of session key (SHA-512)
{
    unsigned char hash[SHA512_DIGEST_LENGTH]; // SHA512_DIGEST_LENGTH=64
    SHA512_CTX sha512;
    SHA512_Init(&sha512);
    SHA512_Update(&sha512, in, 8*16);
    SHA512_Final(hash, &sha512);
    int i = 0;
    for(i = 0; i < SHA512_DIGEST_LENGTH; i++)
    {
        sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
    }
    outputBuffer[128]=0;
}

int calculate_reconcile(int num_peer, uint32_t s[1024], uint64_t rec[16],
    uint64_t k[16], unsigned char hk[129], FFT_CTX *ctx){ // calculate reconcile
  int ret;
  uint32_t e[1024];
  RAND_CTX rand_ctx;
  ret = RAND_CHOICE_init(&rand_ctx); // initialize seed
  if (!ret)
  {
    return ret;
  }
```

```
#if CONSTANT_TIME
  rlwe_sample_ct(e, &rand_ctx); // sample e''_{N-1} (constant)
#else
  rlwe_sample(e, &rand_ctx); // sample e''_{N-1} (non-constant)
#endif

  uint32_t Y[MAX_PEER][POLY_LEN];
  uint32_t tmp[1024];
  uint32_t tmp2[1024];
  for(int t=0; t<1024; t++){
    tmp[t]=pub_keys[num_peer-2][t]; // tmp = z_{N-2}
    tmp2[t]=augmented_pub_keys[num_peer-1][t]; // tmp2 = X_{N-1}
  }

  FFT_mul(tmp, tmp, s, ctx); // tmp = z_{N-2} * s_{N-1}
  FFT_add(tmp, tmp, tmp2); // tmp = z_{N-2} * s_{N-1} + X_{N-1}
  FFT_add(tmp, tmp, e); // tmp = z_{N-2} * s_{N-1} + X_{N-1} + e''_{N-1}

  for(int k=0; k<1024; k++){
    Y[num_peer-1][k]=tmp[k]; // save tmp as Y_{N-1}
    tmp2[k]=augmented_pub_keys[0][k]; // tmp2 = X_0
  }

  FFT_add(tmp, tmp, tmp2); // tmp = Y_{N-1} + X_0
  for(int k=0; k<1024; k++){
    Y[0][k]=tmp[k]; // save tmp as Y_0
    tmp2[k]=augmented_pub_keys[1][k]; // tmp2 = X_1
  }
  for (int j=1; j<num_peer-1; j++){
    FFT_add(tmp, tmp, tmp2); // tmp = Y_{j-1} + X_j
    for(int k=0; k<1024; k++){
      Y[j][k]=tmp[k]; // save tmp as Y_j
      tmp2[k]=augmented_pub_keys[j+1][k]; // tmp2 = X_{j+1}
    }
  }

  uint32_t result[1024]={0,};
      for (int i = 0; i < num_peer; i++) // compute b_{N-1}
      {
    for(int k=0; k<1024; k++)
    {
      tmp[k]=Y[i][k]; // tmp = Y_i
    }
            FFT_add(result, result, tmp); // result = result + Y_i
      }

#if CONSTANT_TIME
  rlwe_crossround2_ct(rec, result, &rand_ctx); // compute rec (constant)
```

```
  rlwe_round2_ct(k, result); // compute key k_{N-1} (constant)
#else
  rlwe_crossround2(rec, result, &rand_ctx); // compute rec (non-constant)
  rlwe_round2(k, result); // compute key k_{N-1} (non-constant)
#endif
  sha512_session_key(k, hk); // compute hash value of k_{N-1} and save as hk_{N
    -1}

  rlwe_memset_volatile(result, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(e, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(Y, 0, 1024 * MAX_PEER * sizeof(uint32_t));
  rlwe_memset_volatile(tmp, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp2, 0, 1024 * sizeof(uint32_t));
  RAND_CHOICE_cleanup(&rand_ctx);
  return ret;
}

int next_option(int option, int num_peer) // To check whether (step i) finish
    or not
{
    bool check = true;
    for (int i = 0; i < num_peer - 1; i++)
    {
        check = check && option_check[option][i];
    }
    if (check)
        return option + 1;
    return option;
}

void run_server(int num_peer, int server_port) // Communication between peers
    and arbiter
{
    int server_socket;
    server_socket = socket(PF_INET, SOCK_STREAM, 0);
    if (server_socket == -1)
    {
        printf("socket() error!\n");
        exit(1);
    }

    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family      = AF_INET;
    server_addr.sin_port        = htons(server_port);
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);

    if (bind(server_socket, (struct sockaddr *)&server_addr, sizeof(server_addr
```

```
)) == -1)
{
    printf("bind() error!\n");
    exit(1);
}
if (listen(server_socket, 5) == -1)
{
    printf("listen() error!\n");
    exit(1);
}

struct sockaddr_in client_addr;
socklen_t client_addr_size;
client_addr_size = sizeof(client_addr);
int client_socket[MAX_PEER];
for (int i = 0; i < num_peer; i++)
{
    client_socket[i] = 0;
}
fd_set readfds;
int sd, max_sd;
int activity;
int new_socket;
int peer;
int option = 0;
bool first_process;
uint32_t result[POLY_LEN];

memset(check_augmented_pub_keys, false, sizeof(check_augmented_pub_keys));
memset(option_check, false, sizeof(option_check));
bool reconcile_calculated = false;
FFT_CTX ctx;
FFT_CTX_init(&ctx);
calculate_pubkey(num_peer - 1, rlwe_a, sec_keys[num_peer - 1], &ctx); //
calculate z_{N-1}

while (option < 4)
{
    FD_ZERO(&readfds);
    FD_SET(server_socket, &readfds);
    max_sd = server_socket;

    for (int i = 0; i < num_peer-1; i++)
    {
        sd = client_socket[i];
        if (sd > 0)
            FD_SET(sd, &readfds);
        if (sd > max_sd)
```

```
            max_sd = sd;
    }
    activity = select(max_sd + 1, &readfds, NULL, NULL, NULL);

    if (FD_ISSET(server_socket, &readfds)) // peer and arbiter connect
    {
        new_socket = accept(server_socket, (struct sockaddr *)&client_addr,
&client_addr_size);
        for (int i = 0; i < num_peer-1; i++)
        {
            if (client_socket[i] == 0)
            {
                client_socket[i] = new_socket;
                break;
            }
        }
    }

    for (int p = 0; p < num_peer-1; p++)
    {
        sd = client_socket[p];

        if (FD_ISSET(sd, &readfds))
        {
            recv(sd, &peer, sizeof(peer), 0);
            if (!(0 <= peer && peer < num_peer))
            {
                printf("peer number error\n");
                close(sd);
                continue;
            }

            if (!reconcile_calculated) // if rec is not computed
            {
                bool all_augmented_pub_keys = true;
                for (int i = 0; i < num_peer; i++)
                {
                    if (!check_augmented_pub_keys[i])
                    {
                        all_augmented_pub_keys = false;
                        break;
                    }
                }
                if (all_augmented_pub_keys) // if receive all X_i (0<=i<=N
-2)
                {
                    calculate_reconcile(num_peer, sec_keys[num_peer - 1],
reconcile, session_keys[num_peer - 1], hashed_keys[num_peer-1], &ctx);
```

```
                    reconcile_calculated = true;
                }
            }

            send(sd, &option, sizeof(option), 0); // send step i (i=option)

            if (option == 1) // if step 0 is done, compute X_{N-1}
            {
                calculate_augmented_pubkey(num_peer - 1, num_peer, sec_keys
[num_peer - 1], &ctx);
                check_augmented_pub_keys[num_peer - 1] = true;
            }

            first_process = !option_check[option][peer];
            send(sd, &first_process, sizeof(first_process), 0);

            if (!first_process)
                continue;

            switch (option)
            {
                case 0:
                {
                    recv(sd, pub_keys[peer], POLY_LEN * sizeof(uint32_t),
0); // receive z_i
                    printf("option 0 clear with peer %d!\n", peer);
                    break;
                }
                case 1:
                {
                    send(sd, pub_keys, sizeof(uint32_t) * num_peer *
POLY_LEN, 0); // broadcast z
                    recv(sd, result, sizeof(result), 0); // receive X_i
                    memcpy(augmented_pub_keys[peer], result, sizeof(
augmented_pub_keys[peer]));
                    check_augmented_pub_keys[peer] = true;
                    printf("option 1 clear with peer %d!\n", peer);
                    break;
                }
                case 2:
                {
                    send(sd, augmented_pub_keys, sizeof(uint32_t) *
num_peer * POLY_LEN, 0); // broadcast X
                    printf("option 2 clear with peer %d!\n", peer);
                    break;
                }
                case 3:
                {
```

```
                        send(sd, reconcile, sizeof(reconcile), 0); // broadcast
     rec
                recv(sd, hashed_keys[peer], sizeof(hashed_keys[peer]), 0); //
    receive sk_i
                        printf("option 3 clear with peer %d!\n", peer);
                    }
                }
                option_check[option][peer] = true;
                option = next_option(option, num_peer); // if communication
    with all peers is done, go to next step
            }
        }
    }

    printf("Arbiter hased key : "); // print sk_{N-1}
    for (int i = 0; i < 129; i++)
        printf("%c", hashed_keys[num_peer - 1][i]);
    printf("\n");
}

int main(int argc, char *argv[])
{
    int num_peer = 3; // N=3
    int server_port = 4000; // default port = 4000
    char op;

    while ((op = getopt(argc, argv, "p:")) != -1)
    {
        switch (op)
        {
            case 'p':
                server_port = atoi(optarg);
                break;
        }
    }

    run_server(num_peer, server_port);
    return 0;
}
```

B Source code of peer

```
#include <arpa/inet.h>
#include <ctype.h>
#include <getopt.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <fcntl.h>
#include <openssl/evp.h>
#include <openssl/sha.h>
#include "fft.h"
#include "rlwe.h"
#include "rlwe_a.h"
#include "rlwe_rand.h"

#define MAX_PEER 6
#define POLY_LEN 1024
#define KEY_LEN  16
#define HASH_LEN 129

uint32_t sec_keys[MAX_PEER][POLY_LEN];
uint32_t pub_keys[MAX_PEER][POLY_LEN];
uint32_t augmented_pub_keys[MAX_PEER][POLY_LEN];
uint64_t session_keys[MAX_PEER][KEY_LEN];
unsigned char hashed_keys[MAX_PEER][HASH_LEN];
uint64_t reconcile[KEY_LEN];

int calculate_pubkey(int peer, const uint32_t *a, uint32_t s[1024], FFT_CTX *
    ctx);
int calculate_augmented_pubkey(int peer, int num_peer, uint32_t s[1024],
    FFT_CTX *ctx);
int calculate_session_key(int peer, int num_peer, uint32_t s[1024], uint64_t
    rec[16], uint64_t k[16], unsigned char hk[129], FFT_CTX *ctx);

int calculate_pubkey(int peer, const uint32_t *a, uint32_t s[1024], FFT_CTX *
    ctx) // calculate z_i in Round 1 (i=peer)
{
  if (peer < 0 || peer > MAX_PEER)
  {
            printf("peer range error!\n");
            return -1;
      }
```

```
  int ret;
  uint32_t e[1024];
  RAND_CTX rand_ctx;
  ret = RAND_CHOICE_init(&rand_ctx); // initialize seed
  if (!ret)
  {
    return ret;
  }

#if CONSTANT_TIME
  rlwe_sample_ct(s, &rand_ctx); // sample s_i (constant)
  rlwe_sample_ct(e, &rand_ctx); // sample e_i (constant)
#else
  rlwe_sample(s, &rand_ctx); // sample s_i (non-constant)
  rlwe_sample(e, &rand_ctx); // sample e_i (non-constant)
#endif

  uint32_t tmp[1024];
  rlwe_key_gen(tmp, a, s, e, ctx);  // compute tmp=as_i+e_i
  for(int t=0; t<1024; t++)
  {
    pub_keys[peer][t]=tmp[t]; // save tmp as pub_keys[peer]
  }

  rlwe_memset_volatile(e, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp, 0, 1024 * sizeof(uint32_t));
  RAND_CHOICE_cleanup(&rand_ctx);
  return ret;
}

int calculate_augmented_pubkey(int peer, int num_peer, uint32_t s[1024],
    FFT_CTX *ctx) // calculate X_i in Round 2 (i=peer)
{
  int ret;
  uint32_t e[1024];
  RAND_CTX rand_ctx;
  ret = RAND_CHOICE_init(&rand_ctx); // initialize seed
  if (!ret)
  {
    return ret;
  }

  uint32_t result[1024]={0,};
  uint32_t tmp1[1024];
  uint32_t tmp2[1024];
  if (peer==num_peer-1) // if i = N-1
  {
#if CONSTANT_TIME
```

```
    rlwe_sample_ct(e, &rand_ctx); // sample e'_{N-1} (constant)
#else
    rlwe_sample(e, &rand_ctx); // sample e'_{N-1} (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[0][t]; // tmp1 = z_0
      tmp2[t]=pub_keys[peer-1][t]; // tmp2 = z_{N-2}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_0 - z_{N-2}
    FFT_mul(result, result, s, ctx);  // result = (z_0 - z_{N-2}) * s_{N-1}
    FFT_add(result, result, e); // result = (z_0 - z_{N-2}) * s_{N-1} + e'_{N
    -1}
  }
  else if (peer==0) // if i = 0
  {
#if CONSTANT_TIME
    rlwe_sample2_ct(e, &rand_ctx); // sample e'_0 from sigma2 (constant)
#else
    rlwe_sample2(e, &rand_ctx); // sample e'_0 from sigma2 (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[peer+1][t]; // tmp1 = z_1
      tmp2[t]=pub_keys[num_peer-1][t]; // tmp2 = z_{N-1}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_1 - z_{N-1}
    FFT_mul(result, result, s, ctx); // result = (z_1 - z_{N-1}) * s_0
    FFT_add(result, result, e); // result = (z_1 - z_{N-1}) * s_0 + e'_0
  }
  else // if 1<= i <= N-2
  {
#if CONSTANT_TIME
    rlwe_sample_ct(e, &rand_ctx); // sample e'_i (constant)
#else
    rlwe_sample(e, &rand_ctx); // sample e'_i (non-constant)
#endif
    for(int t=0; t<1024; t++)
    {
      tmp1[t]=pub_keys[peer+1][t]; // tmp1= z_{i+1}
      tmp2[t]=pub_keys[peer-1][t]; // tmp2 = z_{i-1}
    }
    FFT_sub(result, tmp1, tmp2); // result = z_{i+1} - z_{i-1}
    FFT_mul(result, result, s, ctx); // result = (z_{i+1} - z_{i-1}) * s_i
    FFT_add(result, result, e); // result = (z_{i+1} - z_{i-1}) * s_i + e'_i
  }

  for(int t=0; t<1024; t++)
```

```
  {
    augmented_pub_keys[peer][t]=result[t]; // save result as augmented_pub_keys
    [peer]
  }

  rlwe_memset_volatile(result, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp1, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp2, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(e, 0, 1024 * sizeof(uint32_t));
  RAND_CHOICE_cleanup(&rand_ctx);
  return ret;
}

void sha512_session_key(uint64_t *in, char outputBuffer[129]) // calculate hash
     value of session key (SHA-512)
{
    unsigned char hash[SHA512_DIGEST_LENGTH]; // SHA512_DIGEST_LENGTH=64
    SHA512_CTX sha512;
    SHA512_Init(&sha512);
    SHA512_Update(&sha512, in, 8*16);
    SHA512_Final(hash, &sha512);
    int i = 0;
    for(i = 0; i < SHA512_DIGEST_LENGTH; i++)
    {
        sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
    }
    outputBuffer[128]=0;
}

int calculate_session_key(int peer, int num_peer, uint32_t s[1024], uint64_t
    rec[16], uint64_t k[16], unsigned char hk[129], FFT_CTX *ctx) // compute
    sk_i
{
  uint32_t Y[MAX_PEER][POLY_LEN];
  uint32_t tmp[1024];
  uint32_t tmp2[1024];

  for(int t=0; t<1024; t++)
  {
    tmp[t]=pub_keys[(peer+num_peer-1)%num_peer][t]; // tmp = z_{i-1} (peer=i)
    tmp2[t]=augmented_pub_keys[peer][t];  // tmp2 = X_i
  }

  FFT_mul(tmp, tmp, s, ctx); // tmp = z_{i-1} * s_i
  FFT_add(tmp, tmp2, tmp); // tmp = X_i + z_{i-1} * s_i

  for(int t=0; t<1024; t++)
  {
```

```
    Y[peer][t]=tmp[t]; // save tmp as Y_i
    tmp2[t]=augmented_pub_keys[(peer+1)%num_peer][t]; // tmp2 = X_{i+1}
  }
  for (int j=1; j<num_peer; j++)
  {
    FFT_add(tmp, tmp, tmp2); // tmp = Y_{i+j-1} + X_{i+j}
    for(int t=0; t<1024; t++)
    {
      Y[(peer+j)%num_peer][t]=tmp[t]; // save tmp as Y_{i+j}
      tmp2[t]=augmented_pub_keys[(peer+j+1)%num_peer][t]; // tmp2 = X_{i+j+1}
    }
  }

  uint32_t result[1024]={0,};
      for (int i = 0; i < num_peer; i++) // compute b_i
    {
    for(int k=0; k<1024; k++)
    {
      tmp[k]=Y[i][k]; // tmp = Y_i
    }
            FFT_add(result, result, tmp); // result = result + Y_i
      }
#if CONSTANT_TIME
  rlwe_rec_ct(k, result, rec); // compute key k_i (constant)
#else
  rlwe_rec(k, result, rec); // compute key k_i (non-constant)
#endif
  sha512_session_key(k, hk); // compute hash value of k_i and save as hk_i

  rlwe_memset_volatile(result, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(Y, 0, 1024 * MAX_PEER * sizeof(uint32_t));
  rlwe_memset_volatile(tmp, 0, 1024 * sizeof(uint32_t));
  rlwe_memset_volatile(tmp2, 0, 1024 * sizeof(uint32_t));
  return 1;
}

int main(int argc, char *argv[])
{
    int client_socket;
    client_socket = socket(PF_INET, SOCK_STREAM, 0);
    if (client_socket == -1)
    {
        printf("socket() error!\n");
        exit(1);
    }

    char *server_ip = "127.0.0.1";
    int server_port = 4000;
```

```
char op;
int option = -1;
int peer = -1;
bool first_process;
int num_peer = 3;
FFT_CTX ctx;
FFT_CTX_init(&ctx);

while ((op = getopt(argc, argv, "h:p:o:w:")) != -1)
{
    switch (op)
    {
        case 'h':
            server_ip   = optarg;
            break;
        case 'p':
            server_port = atoi(optarg);
            break;
        case 'o':
            option      = atoi(optarg);
            break;
        case 'w':
            peer        = atoi(optarg);
            break;
    }
}

struct sockaddr_in server_addr;
memset(&server_addr, 0, sizeof(server_addr));
server_addr.sin_family      = AF_INET;
server_addr.sin_port        = htons(server_port);
server_addr.sin_addr.s_addr = inet_addr(server_ip);

if (connect(client_socket, (struct sockaddr *)&server_addr, sizeof(
server_addr)) == -1)
{
    printf("connect() error!\n");
    exit(1);
}

while (true)
{
    send(client_socket, &peer, sizeof(peer), 0);
    recv(client_socket, &option, sizeof(option), 0);
    recv(client_socket, &first_process, sizeof(first_process), 0);

    if (!first_process)
        continue;
```

```
    if (option > 3)
        break;

    switch (option)
    {
        case 0:
        {
            calculate_pubkey(peer, rlwe_a, sec_keys[peer], &ctx); //
compute z_i
            send(client_socket, pub_keys[peer], sizeof(pub_keys[peer]), 0);
 // send z_i
            break;
        }
        case 1:
        {
            recv(client_socket, pub_keys, sizeof(uint32_t) * num_peer *
POLY_LEN, 0); // receive z
            calculate_augmented_pubkey(peer, num_peer, sec_keys[peer], &ctx
); // compute X_i
            send(client_socket, augmented_pub_keys[peer], sizeof(
augmented_pub_keys[peer]), 0); // send X_i
            break;
        }
        case 2:
        {
            recv(client_socket, augmented_pub_keys, sizeof(uint32_t) *
num_peer * POLY_LEN, 0); // receive X
            break;
        }
        case 3:
        {
            recv(client_socket, reconcile, sizeof(reconcile), 0); //
receive rec
            uint64_t result[KEY_LEN];
 unsigned char hashed_result[HASH_LEN];
            calculate_session_key(peer, num_peer, sec_keys[peer], reconcile
, result, hashed_result, &ctx); // compute sk_i
   send(client_socket, hashed_result, sizeof(hashed_result), 0); // send
sk_i

            printf("Peer %d hased key : ", peer); // print sk_i
        for (int i = 0; i < 129; i++)
                printf("%c", hashed_result[i]);
            printf("\n");
            break;
        }
        default:
        {
```

```
                printf("unknown option!\n");
                break;
            }
        }
    }
    close(client_socket);
    return 0;
}
```

Acknowledgments in Korean

이 논문을 작성하기까지 많은 분들의 도움이 있었습니다. 먼저, 끊임없는 조언과 연구 지도를 통해 연구자로서의 삶을 가르쳐주신 김광조 교수님께 진심으로 감사드립니다. 또한, 바쁜 와중에도 귀한 시간 내서 학위논문심사에 참여해주신 신인식 교수님과 이주영 교수님께도 깊은 감사의 말씀을 드립니다.

그리고 2년동안 함께 지낸 연구실 동료들에게 감사의 말을 전하고 싶습니다. 항상 랩장으로서 연구실 사람들을 이끌어준 락용이형, 책임감 있게 힘든 일도 꿋꿋이 해나가던 지은이, 같이 지낸 동안 아낌없이 조언해준 성숙이 누나, 석사 생활동안 계속해서 진심어린 조언을 해준 형철이 형, 연구실 동기로서 항상 의지할 수 있었던 낙준이, 비슷한 주제로 연구하고 토론하면서 같이 고생했던 동연이, 연구실의 든든한 존재가 되어주었던 나비 누나, 에이스로서 부족한 부분을 잘 채워주었던 승근이, 항상 친근한 모습으로 다가왔던 Harry, 연구실에 색다른 활기를 가져다 준 Aminanto와 Edwin에게 늘 고마웠습니다.

또한, 석사 과정을 함께하며 고생한 정보보호대학원 석사 동기분들과, 행정적인 일을 수월하게 처리해 주신 박찬수 선생님, 이지선 선생님, 그리고 홍지연 선생님께 감사의 말씀을 드립니다.

끝으로, 어떤 상황에서도 저의 버팀목이 되어주시던 부모님, 항상 응원해준 친구들에게 감사의 말을 전하고 싶습니다. 꾸준히 나아가는 사람이 되도록 하겠습니다. 감사합니다.

Curriculum Vitae in Korean

이 름: 한 성 호

생 년 월 일: 1991년 6월 14일

전 자 주 소: hansh09@kaist.ac.kr

학 력

2007. 3. – 2009. 2. 서울 한성과학고등학교

2009. 2. – 2016. 2. 한국과학기술원 수리과학과 (B.S.)

2017. 9. – 2020. 2. 한국과학기술원 정보보호대학원 (M.S.)

경 력

2018. 9. – 2018. 12. 한국과학기술원 고급 사이버보안 실무 일반조교

2019. 9. – 2019. 12. 한국과학기술원 고급 정보보호 일반조교

연 구 과 제

2017. 12. – 2018. 1. 양자컴퓨터 공격에 안전한 새로운 래티스 기반 완전 준동형 서명 방식 설계 및 안전성 분석

2018. 3. – 2019. 12. 양자 컴퓨터 환경에서 래티스 문제를 이용한 다자간 인증키 교환 프로토콜 연구

2018. 5. – 2018. 10. 암호화폐와 스마트 컨트랙트 응용 시스템 설계 및 보안 취약성 분석 연구

2019. 9. – 2019. 12. 양자 난수 생성기의 보안성 및 성능 연구

연 구 업 적

1. 안형철, **한성호**, 최낙준, 김광조, "**OQS 프로젝트 중 격자 기반 키 교환 방식의 타이밍 등 공격 분석**", 한국정보보호학회 동계학술대회(CISC-W'17), 2017.12.09. 고려대학교, 서울.

2. **Seongho Han**, Nakjun Choi, Hyeongcheol An, Rakyong Choi, and Kwangjo Kim, "**Prey on Lizard : Mining Secret Key on Lattice-based Cryptosystem**", 2018 Symposium on Cryptography and Information Security, Session 3A4-2 (SCIS 2018), Jan., 23-26, 2018, Niigata, Japan.

3. **한성호**, 홍동연, 최낙준, 이나비, 김광조, "**(D)PoS 기반 블록체인의 거래 및 합의 방식 분석**", 한국정보보호학회 하계학술대회(CISC-S'18), 2018.06.21. 동신대학교, 나주.

4. **한성호**, 안형철, 김광조, "**EOS 암호화폐의 블록 생성에 대한 인센티브 분석**", 한국정보보호학회 동계학술대회(CISC-W'18), 2018.12.08. 세종대학교, 서울.

5. **Seongho Han**, Rakyong Choi, and Kwangjo Kim, "**Adding Authenticity into Tree-based Group Key Agreement by Permissionless Public Ledger**", 2019 Symposium on Cryptography and Information Security, Session 2G3-3 (SCIS 2019), Jan., 22-25, 2019, Otsu, Japan.

6. **한성호**, 최락용, 김광조, "**RLWE 기반 분산형 그룹키 교환 방식의 성능 분석**", 한국정보보호학회 동계학술대회(CISC-W'19), 2019.11.30. 중앙대학교, 서울.