Master's thesis

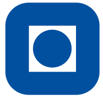
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Simen Been Kristiansen

Evaluating Post-Quantum Group Key Exchange

Master's thesis in Communication Technology
Supervisor: Colin Boyd, Bor de Kock
June 2020

NTNU
Norwegian University of
Science and Technology



NTNU – Trondheim
Norwegian University of
Science and Technology

Evaluating Post-Quantum Group Key Exchange

Simen Been Kristiansen

Master of Science in Communication Technology

Submission date:	June 2020
Responsible professor:	Colin Boyd, IIK
Supervisor:	Bor de Kock, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title:	Evaluating Post-Quantum Group Key Exchange
Student:	Simen Been Kristiansen

Problem description:

The purpose of this thesis is to evaluate efficient protocols for post-quantum group key exchanges based the Ring Learning-With-Errors problem, which is believed to be difficult to solve, even given a practical quantum computer. In particular, this thesis will apply design science iterating through three cycles - first implementing and evaluating basic post-quantum group key exchanges, on to group authenticated key exchanges in the second cycle. The final cycle will, if time allows, include an evaluation of a real-world implementation of a post-quantum group authenticated key exchange.

Responsible professor:	Colin Boyd, IIK
Supervisor:	Bor de Kock, IIK

Abstract

Public key cryptography is the foundation on which the security of many popular services are built. Recent innovations in the field of quantum computing could pose a risk to this current paradigm. Post-quantum cryptography is the solution to this threat, but with it comes a set of challenges in terms of efficiency, usability, and complexity.

In this thesis, we explore protocols for post-quantum group key exchange based on a mathematical problem that is believed to be hard to solve efficiently, even using quantum computers: Ring-learning with errors (RLWE). Specifically, we instantiate two protocols with an estimated 105 bits of security, and provide efficient implementations using the number-theoretic transform (NTT) and Barrett reduction for the group key exchange protocols and authenticated versions. Finally, we provide a prototype for a real-world application: interactive group messaging.

Information gathered in the process indicates that specific post-quantum group key exchange protocols are feasible in certain situations, such as on desktop and server computers. We achieve a performance comparable to several post-quantum two-party key exchange protocols. Finally, we note that further work on a protocol level is required to attain non-interactive protocols with auxiliary properties such as deniability.

Sammendrag

Offentlig nøkkelkryptografi er grunnlaget som sikkerheten til mange populære tjenester er bygget på. Nyere innovasjoner innen kvantedatamaskiner kan utgjøre en risiko for dette nåværende paradigmet. Kvantesikker kryptografi er løsningen på denne trusselen, men medfører en mengde utfordringer hva gjelder effektivitet, brukbarhet og kompleksitet.

I denne oppgaven utforsker vi protokoller for kvantesikker gruppenøkkelutveksling basert på et matematisk problem som antas å være vanskelig å løse effektivt, selv ved bruk av kvantedatamaskiner: Ringlæring med feil (RLWE). Spesifikt instansierer vi to protokoller med anslagsvis 105 biters sikkerhet, og gir effektive implementasjoner ved bruk av den tall-teoretiske transformasjonen (NTT) og Barrett-reduksjon for gruppenøkkelutvekslingsprotokollene samt autentiserte versjoner. Til slutt gir vi en prototype for en virkelighetsnær applikasjon: en interaktiv tekst-basert gruppesamtaleapplikasjon.

Informasjon samlet i prosessen indikerer at spesifikke kvantesikre gruppenøkkelutvekslingsprotokoller fungerer i visse situasjoner, for eksempel på stasjonære datamaskiner og servere. Vi oppnår en ytelse som kan sammenlignes med flere kvantesikre nøkkelutvekslingsprotokoller mellom to parter. Til slutt bemerker vi at det kreves ytterligere arbeid på et protokollnivå for å oppnå ikke-interaktive protokoller med tilleggsegenskaper, slik som fornektbarhet.

Preface

This Master's Thesis completes my Master of Science (Sivilingeniør) degree in Communication Technology with a specialization in information security, at the Department for Information Security and Communication Technology (IIK) at the Norwegian University of Science and Technology (NTNU).

It is worth 30 study credits in the spring semester of 2020 as part of the course TTM4905, and is a continuation of my pre-project from the autumn semester of 2019.

Acknowledgements

The work presented in this thesis is the result of much effort over the past months, guided by my brilliant supervisors, Colin A. Boyd and Bor de Kock, at the Department for Information Security and Communication Technology (IIK). This thesis could not have happened without your excellent guidance, interest, help, and all our discussions. I am deeply grateful to you for your support. Thank you.

Furthermore, I would like to extend my gratitude to all the people who have made my life in Trondheim amazing these past five years.

Finally, I would like to thank my family for their invaluable moral support.

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xv

List of Acronyms xvii

1 Introduction 1
- 1.1 Motivation 3
- 1.2 Methodology 3
- 1.3 Research Objective 5
- 1.4 Outcome 5
- 1.5 Limitations 6
- 1.6 Outline 6

2 Background and Theory 7
- 2.1 Mathematical background 7
  - 2.1.1 Group and Ring Theory 7
  - 2.1.2 The Fourier Series and the Number Theoretic Transform 9
- 2.2 Modern Cryptography 10
  - 2.2.1 Random Number Generators 11
  - 2.2.2 Bits of Security 12
  - 2.2.3 Side-Channel Attacks on Cryptography 12
  - 2.2.4 Secret Key Cryptography 12
  - 2.2.5 Public Key Cryptography 14
  - 2.2.6 Key Exchange 16
- 2.3 Ring Learning With Errors Cryptography 17
  - 2.3.1 Generic RLWE Key Exchange 18
  - 2.3.2 Key Reconciliation Protocols 19
  - 2.3.3 Attacks on RLWE Cryptography 20
- 2.4 RLWE Group Key Exchange 21
  - 2.4.1 Proposal 1: Ding 21

2.4.2 Proposal 2: Apon . . . 22
2.4.3 Proposal 3: Choi . . . 23
2.4.4 Peikert Reconciliation For Groups . . . 24
2.5 Transforming GKE to GAKE . . . 26
2.5.1 The Katz-Yung compiler . . . 26
2.5.2 Compiler for Authenticated GAKE by Bresson et al. . . . 27

3 Protocol Instantiation 29
3.1 A Parameter Proposal . . . 29
3.2 Efficient Implementation Building Blocks . . . 31
3.2.1 Polynomial Convolution . . . 31
3.2.2 Sampling from an Error Distribution . . . 33
3.2.3 Reconciliation Function . . . 35
3.3 Estimating the Computational Security . . . 35

4 Protocol Implementation 37
4.1 Data Collection . . . 37
4.2 Performance of Subroutines . . . 39
4.3 Group Key Exchange Implementations . . . 40
4.3.1 AponGKE and ChoiGKE . . . 40
4.3.2 Performance . . . 42
4.4 GAKE Implementation . . . 44
4.4.1 Choice of Digital Signature Scheme . . . 44
4.4.2 Protecting Against Active Attacks . . . 45
4.4.3 Implementation Modifications . . . 47
4.4.4 Performance . . . 52
4.5 Real-world Interactive Messaging . . . 52
4.5.1 Practical Considerations . . . 52
4.5.2 System Architecture . . . 53
4.5.3 Messages . . . 53
4.5.4 Network Effects and Package Sizes . . . 54

5 Analysis and Discussion 55
5.1 Notable Optimizations . . . 55
5.1.1 Modular Reduction . . . 55
5.1.2 Polynomial Convolution . . . 57
5.1.3 Error Sampling . . . 58
5.1.4 Vectorization and Loops . . . 58
5.2 Performance . . . 60
5.2.1 Central Processing Unit (CPU) Cycles of Subroutines . . . 60
5.2.2 Error Rate . . . 64
5.2.3 Key Sizes . . . 65

5.2.4 Performance of GKE 66
5.2.5 The Impact of Authentication 66
5.2.6 On the Real-world Implementation 68
5.2.7 Potential Trade-offs for Group Key Exchange 68
5.3 Constant-time RLWE Cryptography 69

6 Conclusion **71**
6.1 Research Objectives 71
6.2 Future work 72

References **75**

Appendices

A Compiler options **81**

B Raw data **83**
B.1 GAKE 83
B.1.1 ECDHGAKE 83
B.1.2 FALCON-AponGAKE 84
B.1.3 FALCON-ChoiGAKE 85
B.1.4 qTESLA-AponGAKE 86
B.1.5 qTESLA-ChoiGAKE 87
B.2 GAKE without keygen 88
B.2.1 ECDHGAKE 88
B.2.2 FALCON-AponGAKE 89
B.2.3 FALCON-ChoiGAKE 90
B.2.4 qTESLA-AponGAKE 91
B.2.5 qTESLA-ChoiGAKE 92

List of Figures

2.1 The adversary models for cryptographic protocols, showing both a passive and an active adversary. 10
2.2 A version of the Diffie-Hellman key exchange protocol over finite fields. 16
2.3 A generic KEM to achieve a shared secret between two parties. 17
2.4 A generic RLWE key exchange where Alice and Bob achieve an approximate shared secret ring element. 19
2.5 A generic key reconciliation protocol for RLWE. 20
2.6 An illustration of the Peikert reconciliation function for an odd modulus. The leftmost subfigure shows how the reconciliation vector and shared secret is generated, while the other subfigures show how a recipient with an approximate shared ring element may achieve the same shared secret using the reconciliation vector. 25

3.1 The equivalence relations that allow us to speed up the polynomial convolution through transforming to the NTT domain and performing a pointwise multiplication. The innermost text show the computational complexity of the associated function depicted on the outside. 32
3.2 The output of testing our sampler with SAGA [HPRR19]. The empiric data is close to what we expect, illustrated by the χ^2 statistic and p-value being above the critical value. This figure is the output of running SAGA on polynomials generated using DGS [AW18]. 35

4.1 A comparison of the CPU cycles required to perform an (unauthenticated) GKE for AponGKE, ChoiGKE and pairwise Elliptic Curve Diffie-Hellman (ECDH) using Curve25519. The comparisons are done on the server described above, using the same compilation options found in Appendix A. 44
4.2 An illustration of the encoding schema used to represent messages in the GAKE implementation. Here we use FALCON as an example, though it is easily exchangeable. 48
4.3 The average number of CPU cycles in millions to perform a GAKE using AponGAKE and ChoiGAKE with digital signature schemes FALCON and qTESLA compared with a naive ECDH-GAKE. 51

4.4 A comparison of the performance when discounting the generation of the digital signature scheme key pair. 51
4.5 A minimal example of the architecture used in the simple interactive messaging application. 53
4.6 Polynomial transfer from the server to a local client. Note the message length (in bytes), including overhead from the TCP, IPv4 and Ethernet headers. 54

5.1 A comparison of the CPU cycles to perform a GKE with the unoptimized versions of AponGKE and ChoiGKE. 56
5.2 The top subroutines used in the AponGAKE when excluding key pair generation. All subroutines shown except `GAKE_APON‘modmul` are from the Open Quantum Safe (OQS) implementation of FALCON. This further demonstrates how dominating the authentication part is. All other subroutines use less than 1% of the processor cycles. 67

List of Tables

4.1 The performance for a selection of subroutines that are common to AponGKE and ChoiGKE. Note that `GenPoly(`σ_2`)` which is only used once in a protocol run, by a single participant, and is included here for the sake of completeness. As a comparison, the performance of the naive approach to polynomial convolution is included. 39
4.2 The memory usage in bytes for 3, 10, and 20 participants for both roles (initiator and responder) of AponGKE and ChoiGKE. 43
4.3 Here we see the error rate given by AponGKE and ChoiGKE for a number of participants. We will discuss this figure more in the next chapter. . . 45

5.1 Polynomial convolution performance of a selection of RLWE-based cryptosystems. Due to differents in measuring, we include both the NTT/FFT transformations and the convolution as a whole, to give a fair comparison. We note that our implementation uses a larger modulus than all we compare to here. 61
5.2 A comparison of the performance of the reconciliation functions of BCNS, Singh and the work presented in this thesis. These all use Peikert reconciliation, which provides for a fair comparison. 62
5.3 A comparison of the error sampling methods demonstrated in various papers and specifications. We distinguish between constant-time (CT) and non-constant-time (non-CT) sampling as constant-time is ideal, but non-constant-time can yield better performance at the cost of security against side-channel attacks. As a result they are in seperate categories. Note that we are using the Gaussian sampler of Albrecht [AW18] alone for σ_1, and composited for σ_2. 63
5.4 The CPU cycles used to sample a uniform ring element by the implementation presented in this thesis, compared with the implementation in the NewHope cryptosystem. 64
5.5 Comparison between the key sizes of work here, and previous work, using notable key exchange and key encapsulation mechanisms from both classical schemes and post-quantum schemes. 65

List of Algorithms

3.1 Cooley-Tukey Forwards NTT 33
3.2 Gentleman-Sande Inverse NTT 34
4.1 AponGKE 42
4.2 AponGAKE 50

List of Acronyms

AES Advanced Encryption Standard.

AKE Authenticated Key Exchange.

AVX Advanced Vector Extensions.

CPU Central Processing Unit.

CSPRNG Cryptographically Secure Pseudo-Random Number Generator.

DFT Discrete Fourier Transform.

DH Diffie-Hellman Key Exchange.

ECDH Elliptic Curve Diffie-Hellman.

FFT Fast Fourier Transform.

GAKE Group Authenticated Key Exchange.

GCC GNU Compiler Collection.

GKE Group Key Exchange.

HMAC Hash-based Message Authentication Code.

KDF Key Derivation Function.

KEM Key Encapsulation Mechanism.

LWE Learning With Errors.

NIST The United States National Institute for Standards and Technology.

NTT Number-Theoretic Transform.

OQS Open Quantum Safe.

PRNG Pseudo-Random Number Generator.

RLWE Ring-Learning With Errors.

RNG Random Number Generator.

SHA-3 Secure Hash Algorithm 3.

SIMD Single Instruction Multiple Data.

SSE Streaming SIMD Extensions.

SVP Shortest Vector Problem.

TLS Transport Layer Security.

TRNG Truly Random Number Generator.

Chapter 1 Introduction

Quantum computing has, over the past decades, evolved to the point where we believe it to be capable of breaking current public key cryptography in a matter of years. Public key cryptography is one of the building blocks of the internet, supporting services such as online banking and secure communication. Quantum computers operate on quantum bits — *qubits* — where each qubit can be in three states, two corresponding to the classical 0 and 1 states, and a third state, which is a superposition of the two. This allows for computation on 2^k states using only k qubits. Superpositions and the parallelism of data is a fundamental property of quantum computing used in two theoretical algorithms from the mid-nineties, Shor's algorithm [Sho94] and Grover's algorithm [Gro96]. Shor's algorithm uses a quantum computer to factorize large integers and solve the discrete logarithm problem in polynomial time, breaking all practical cryptosystems based on these problems. Grover's quantum search algorithm gives a quadratic speedup in a search algorithm, which halves the effective key length of any cryptosystem.

As most widely used public key cryptosystems are based on the integer factorization problem or the discrete logarithm problem, these algorithms, coupled with a quantum computer, break the current asymmetric paradigm altogether. New cryptographic primitives that are not broken by the use of quantum computers are required. The term "post-quantum cryptography" was first coined in 2003 by Bernstein [Lan16], describing cryptography built upon problems in which adversarial access to quantum computers have little practical effect. An initial recommendation [ABB+15] lists McEliece with certain parameter sets as a viable post-quantum public key cryptosystem. It is based on a different class of problems than what a practical quantum computer is believed to efficiently break.

A consequence of the developments with regards to quantum computing, is the standardization process of post-quantum cryptographic primitives for key encapsulation and digital signatures by the The United States National Institute for Standards and Technology (NIST) [NIS16]. This process aims to settle on two or more primitives of

each sort for an official US standard. A similar process was attempted with success in standardizing both the Advanced Encryption Standard (AES) (1997-2001) and the Secure Hash Algorithm 3 (SHA-3) (2009-2015).

The primitives currently being evaluated fall into one of several categories [BBD09].

- *Hash-based cryptography*, which is used for post-quantum signature schemes. These are based on the problem of finding collisions in an underlying hash function. This is a problem with proven security guarantees, even against quantum attacks, as these may at best halve the effective computational security.
- *Code-based cryptography*, which uses error-correcting codes. Here a public key is a "disguised" encoding function, while the secret key is an optimal decoding function. This allows anyone to encode a bit-sequence, but only the party with the optimal decoding function is able to retrieve it, as finding an optimal decoding function from a "bad" one is believed to be difficult, even in the quantum-setting.
- *Multivariate cryptography* is based on the difficult problem of finding solutions to large systems of quadratic equations over finite fields. It is believed to be a hard problem, both for classical and quantum computers.
- *Lattice-based cryptography* depends on the difficulty of finding short vectors in an integer lattice (the shortest vector problem, and other similar problems). The public key can be a highly non-orthogonal basis for a lattice (e.g., a basis in Hermitian Normal Form), while the secret key is a nearly-orthogonal basis for the same lattice. It is believed to be difficult to go from a "bad" basis to a nearly-orthogonal lattice for the same basis efficiently, in many dimensions.
- *Supersingular isogeny cryptography* [Cos19] [SP19] is a class of post-quantum cryptography that is based on walks between different supersingular elliptic curves using isogenies.

This thesis focuses on a subset of lattice-based cryptography, Ring-Learning With Errors (RLWE). This class of lattice-based cryptography is a promising candidate for efficient key agreement protocols in real-world settings, and is the basis for two proposals in the NIST standardization process for Key Encapsulation Mechanism (KEM) (NewHope and LAC).

As with the current standards, NIST will standardize the basic primitives, and not more advanced primitives such as Authenticated Key Exchange (AKE) and Group Key Exchange (GKE). This is an area cryptographers and engineers have to explore

further, based on the basic primitives. This thesis will focus on post-quantum GKE based on RLWE.

1.1 Motivation

Post-quantum cryptographic primitives come with a cost when compared to today's standards for performance, even between only two parties. The performance-related issues are exacerbated when scaled up. One such scenario is in group communication, such as conference calls and messaging, grid computing, and collaborative tools. An N-way key exchange using post-quantum primitives in the naive way — every participant performs a key exchange with every other participant — will incur high costs, and is not scalable.

Furthermore, advanced adversaries are believed to be capable of intercepting and storing encrypted data streams, such as secure messaging, encrypted e-mails, and file transfers. These data streams can, at a later point, be decrypted using a quantum computer, if the cryptographic key exchange is included and current popular asymmetric cryptography is used, given that a practical quantum computer becomes feasible for this adversary. As a result, the confidentiality of much of today's communication is already at stake. Evaluating and implementing new post-quantum cryptosystems is intrinsic to the effort of creating and revising primitives and transitioning to post-quantum cryptography in everyday applications.

1.2 Methodology

We believe that design science is an appropriate research methodology, contrasted with many approaches from the natural sciences. In this thesis, we will attempt to discover new knowledge through treating an engineering problem. When solving engineering problems, we change the world around us, yielding knowledge, contrasted with natural science approaches only aiming to discover knowledge through observation. Furthermore, in this thesis, we aim to achieve research objectives, not necessarily to answer fundamental questions about reality. As a result, we believe that design science is an appropriate methodology.

Design science defines two important concepts, *artifacts* and *context*. An artifact is anything created by people for practical purposes [Wie14], most relevant for this thesis is software and algorithms. However, the term encompasses much more in the general sense — varying from abstract concepts to physical objects. The purpose of the artifact is to treat a problem, using it to interact with a context. Note that artifacts *treat* problems, they do not *solve* them. This distinction is important, as the artifact's interaction in the problem context solves the problem, not the artifact in itself.

The context for this thesis is the process of establishing a shared secret between several parties, given the existence of a practical quantum computer. The problem we want to treat is how to achieve the shared secret, specifically. In this thesis, the artifact will be a protocol instantiation and implementation, that as it is developed becomes more complex and efficient. The utility — the end goal of any design science process [HMPR04] — will be provided through the instantiations and implementations themselves, as well as through an evaluation of the proposed protocols based on the artifacts created.

The purpose of design science regarding the work with this thesis is to provide a structured approach to creating artifacts. These artifacts are what we will use as a basis for evaluating post-quantum key exchange. Specifically, the artifacts we aim to create are concrete instantiations and software representations of the combined protocols and instantiations. The design cycle is applied in the manner described below.

1. Problem Investigation We begin each cycle by doing a literature review of the most relevant research. This includes proposals for post-quantum key exchange protocols in general, with a specific focus on group key exchange. The literature review also includes techniques for performing the implementation of specific operations effectively in practice. Furthermore, we research other published, and relevant implementations as these may provide further insight. Based on this, we define the cycle goals.

2. Treatment Design In the design phase, we select the appropriate protocols suitable to treat the problem in an attempt to reach the cycle goals defined in the initial phase. We then provide an instantiation of the protocol(s) through providing a parameter set satisfying the protocol requirements, if this has not already been done in a previous cycle. Finally, we develop or iterate upon the artifact until the cycle goal is believed to have been reached.

3. Treatment Validation Validating the treatment is done through several measures — first, correctness. The artifact should achieve a shared secret between its parties with a low, if not negligible, error rate. Second, security — the group key exchange should achieve a reasonable security level compared with other published protocol implementations. Security is estimated analytically, using the proposed protocol instance as an artifact. Finally, we want the group key exchange to be efficient and constant-time. Efficiency is measured through processor cycles, which should be near-constant — at least for the operations concerning cryptographic data.

We go through three design cycles, each iterating on the results of the previous cycle. Each cycle attempts to achieve a new goal, with the initial cycle creating an

artifact for post-quantum GKE, the second a post-quantum Group Authenticated Key Exchange (GAKE), and finally, a GAKE in a practical application. As such, we are slowly moving the project from a "laboratory"-like environment into an environment more approximate to the real world. Each cycle attempts to gather information so that we may achieve our research objective.

1.3 Research Objective

The research questions are formulated based on the chosen methodology, *Design Science*, and were previously outlined in the pre-project report [Kri19]. The main research goal this thesis will focus on is *design a real-world implementation for group key exchange based on the Ring Learning-With-Errors problem* (**RO0**).

To accomplish this objective, we have divided it up into the following research objectives (RO).

- **RO1**: What proposed protocols are suitable candidates for post-quantum group key exchange based on the RLWE problem?
- **RO2**: Instantiate a protocol for group key exchange based on the RLWE problem with adequate security supporting a reasonable number of participants.
- **RO3**: Is this protocol instantiation usable in terms of delay, processing requirements and quality of experience?
- **RO4**: What, if any, trade-offs must be made in order to achieve post-quantum security in a group setting?
- **RO5**: How can we implement performant RLWE cryptosystems?
- **RO6**: How can we secure real-world implementations of RLWE cryptography from side-channel attacks?

1.4 Outcome

The goal of this thesis is to instantiate, implement, and evaluate practical protocols for performing group key exchange based on the RLWE-problem. The outcome from this thesis should then be one or more instantiations — protocols with specific building blocks, numbers, and other essential parts — and corresponding implementations. Furthermore, we wish to compare our implementation with corresponding current "state-of-the-art" implementations.

The implementations for the different group key exchange protocols may be found at https://github.com/simenbkr/rlwe-gke, using commit `1795995`.

1.5 Limitations

The main limitation of this thesis is imposed by existing RLWE group key exchange protocols, which are all interactive, meaning all participating parties have to participate in the key exchange concurrently. Furthermore, the frameworks used to achieve authenticated security require interactivity. The requirement of interactivity is undesirable, and many protocols used in group key exchange scenarios today are non-interactive.

1.6 Outline

Chapter 2 covers the necessary mathematical theory and cryptographic background. In Chapter 3 we provide an instantiation of two post-quantum group key exchange protocols. Then, in Chapter 4 we present an implementation and the results produced by this. Chapter 5 covers a discussion on what we presented in the preceding chapters. Finally, we provide a conclusion with a response to the research objectives in Chapter 6.

Chapter 2
Background and Theory

This chapter provides the necessary concepts and background material used in this thesis. It is divided into several parts, covering the essential mathematical background, modern cryptographic concepts, an introduction to RLWE, and a summary of proposed RLWE group key exchange protocols.

2.1 Mathematical background

This section explains the foundational mathematics used in this thesis.

2.1.1 Group and Ring Theory

Group and ring theory are important building blocks in organizing sets, and are especially useful in cryptography. For a more thorough reference, we refer the reader to e.g., Fraleigh [Fra14] or Stinson and Paterson [SP19].

Set A set is a collection of distinct objects, e.g., integers.

Groups A group, G, is a set with a group operator, $*$, such that the group axioms are fulfilled:

1. The group operator must be *associative*: $a, b, c \in G \implies (a * b) * c = a * (b * c)$.
2. The group has an *identity element*, e, such that $x * e = e * x = x$, $\forall x \in G$.
3. All group elements have an inverse: $a \in G \implies a * b = b * a = e$, for one and only one element $b \in G$.

The group operator can be any well-defined operation that combines two group elements into a third, as long as the above axioms hold true.

An Abelian Group A group is abelian if the group operator is commutative: $a, b \in G \implies a * b = b * a$, $\forall a, b \in G$.

Rings R is a ring if the ring axioms are fulfilled:

1. R is an abelian group under addition.
2. Multiplication is associative in the ring.
3. The left distribution law and the right distribution law applies to all operations in the ring: $a, b, c \in R \implies a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $a, b, c \in R \implies (a + b) \cdot c = (a \cdot c) + (b \cdot c)$.

Rings have a multiplicative identity, called *unity*, and an additive identity. For rings of integers, the unity is 1, and the additive identity is 0.

Ideals Given a ring, R, and an additive subgroup $A \leq R$. Then, if $rA \subseteq A$ and $As \subseteq A$ $\forall r, s \in R$, A is an ideal.

Quotient Rings Given R, ring, and A, an ideal of R, then R/A is a quotient ring, defined as the additive cosets of A with binary operation such that $a, b \in R/A \implies (a + A) + (b + A) = (a + b) + A$ and $(a + A) \cdot (b + A) = (a \cdot b) + A$.

Rings of Polynomials Let R be any ring. Then $R[x]$ is the corresponding ring of polynomials, with elements of the form $\sum_{i=0}^{\infty} a_i x^i$, $a_i \in R$. An example is the typical polynomials $\mathbb{C}[x]$ where each coefficient is a complex number.

Fields A field, F, is a ring in which all elements have an inverse. A typical example of a field is the integers modulo a prime number, denoted as $\mathbb{Z}_p$.

Root of unity A nth root of unity, x, in a ring, R, is defined as $x^n = 1$ — For instance, in $\mathbb{Z}_{11}^*$, a 2nd root of unity is 10 because $10^2 \equiv 1 \mod 11$.

Primitive root of unity An nth root of unity, x, is also a primitive root of unity if $x^k \neq 1$ for $1 \leq k < n$.

Lattice A lattice is a subgroup of $\mathbb{R}^n$, and is composed of integer linear combinations of $k \leq n$ linearly independent vectors. $\Lambda = \left\{ \sum_{i=1}^{k} \alpha_i \vec{v}_i \right\}$, where $\{\vec{v}_0, \ldots, \vec{v}_k\}$ are linearly independent in $\mathbb{R}^n$, and $\alpha_i \in \mathbb{Z}$.

2.1.2 The Fourier Series and the Number Theoretic Transform

The Fourier transform is a method to transform periodic functions and signals into the frequency domain. This will prove useful in efficiently calculating the convolution of polynomials, which is a fundamental operation in RLWE cryptosystems. Let $f(x) \in \mathbb{C}[x]$. Then the Fourier transform is given as

$$\hat{f}(w) = \mathcal{F}(f(x)) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-2\pi i w x} dx.$$

And the inverse transform is given as

$$f(x) = \mathcal{F}^{-1}(\hat{f}(w)) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(w) e^{2\pi i w x} dw.$$

The discrete version of the Fourier transform, the Discrete Fourier Transform (DFT), is defined on a set of L elements — $\{a_0, a_1, \ldots a_L\}$ —, where the kth result of the forwards transform is given as

$$\hat{a}_k = \sum_{j=0}^{L-1} a_k e^{\frac{-2\pi i k j}{L}}.$$

With a corresponding inverse transform.

There exists a further generalization of the Fourier transformations, that can be used over finite fields, namely the Number-Theoretic Transform (NTT). We recognize that $e^{\frac{2\pi k i}{n}}$ is an nth root of unity in the complex numbers. For a ring (field) $\mathbb{Z}_q$, we can find nth roots of unity as $x \in \mathbb{Z}_q$ s.t. $x^n \equiv 1 \mod q$.

Let ω denote the $2n$th root of unity in R_q. We define the functions NTT and $INTT$ as the forwards and the inverse number theoretic transforms.

$$\hat{f}(w) = NTT(f(x)) := \sum_{i=0}^{i=n-1} \hat{a}_i x^i \text{ where } \hat{a}_i = \sum_{j=0}^{j=n-1} a\omega^{ij}$$

$$f(x) = INTT(\hat{f}(w)) := n^{-1} \sum_{i=0}^{i=n-1} \bar{a} x^i \text{ where } \bar{a} = \sum_{j=0}^{j=n-1} \hat{a}\omega^{-ij}$$

For the NTT, the Fourier theorems hold [Kre02] [LN16] [AB75], and the convolution $(f * g)(x) = INTT(NTT((f * g)(x))) = INTT(NTT(f(x)) \circ NTT(g(x)))$, where $\circ$ denotes pointwise multiplication.

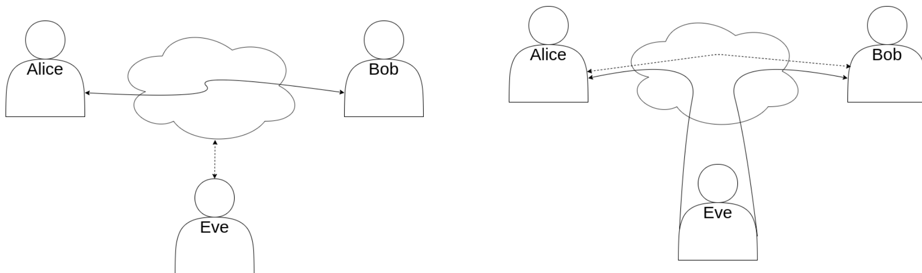
We also note that while the Fourier transform and the DFT work with floating point operations, the NTT works with, and return exact integer results.

2.2 Modern Cryptography

Cryptography is about protecting information. Protecting it from eavesdropping and tampering, and from being misrepresented. Typically, the basic properties cryptography aims to provide are defined as the following [SP19]:

- **Confidentiality** — information should only be available to the intended recipient.
- **Integrity** — after information has been created and transmitted, an adversary should be unable to tamper with it.
- **Authenticity** — recipients should be able to tell that information came from a specified sender with some verification mechanism.

Furthermore, we typically differentiate between two general types of attackers. Figure 2.1 depicts the different general types of adversary from which we attempt to protect our communications and data. There is a passive adversary model where the adversary can read all communication going over a network, and an active adversary, where the adversary controls the network and can read, edit, delete and redirect messages.



(a) A passive adversary is typically modeled as being able to read all messages going across the communications channel.

(b) An active adversary can intercept messages, initiate communications, redirect messages to another recipient, or delete messages entirely.

Figure 2.1: The adversary models for cryptographic protocols, showing both a passive and an active adversary.

Auxilliary properties that modern cryptographic protocols attempt to achieve include the following:

- **Non-repudiation** — after an interaction, the acting party should be unable to deny it committed the interaction in question. This is an important property for e.g., bank transactions and elections
- **Deniability** — communication or interaction should not be provable by a third party. This may be useful in communications scenarios such as instant messaging or voice calls.
- **Forward secrecy** — if the adversary obtains the long-term secret key, the confidentiality of previous interactions should not be at risk.
- **Backward secrecy** — also known as post-compromise security, is the ability of a cryptographic channel to self-heal after having been compromised such that future communication is secure, even if secret keys were compromised at an earlier stage.

In cryptography, we also have the notion of the lifetime of keys — roughly divided as long-term keys and ephemeral (short-term) keys. Often, a long-term key is used over several protocol runs in order to provide authenticity, in combination with an ephemeral key which is used for a single protocol run.

2.2.1 Random Number Generators

In cryptography we are often required to generate secret random numbers. For this purpose, we use a Random Number Generator (RNG). Typically, we define two general classes of RNG, Pseudo-Random Number Generator (PRNG) and Truly Random Number Generator (TRNG). A PRNG generates numbers that "look" random — in other words, for each output on average half the bits are different from the previous output — however the chain of random numbers generated are deterministic based on the internal state (e.g., a seed) of the PRNG. A type of PRNG is a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG), which is what we use for cryptographic purposes. A TRNG generates "true" random numbers based on a source of randomness such as radioactive decay or lava lamps[1].

In practice, we use a CSPRNG where the initial state is seeded by a TRNG. This is due to TRNG being slower than CSPRNG, and less practical to use.

[1] See https://blog.cloudflare.com/randomness-101-lavarand-in-production.

2.2.2 Bits of Security

"Bits of security" is a term used to describe the amount of work needed to break — for example recovering a secret key or the encrypted data — a cryptosystem, using the best possible attack known. If the attacker must perform $\mathcal{O}(2^k)$ operations to break it, we say it has k bits of security.

Concerning the security of a cryptosystem, we often use two security parameters, the computational security parameter, denoted λ, and the statistical security parameter, denoted ρ. These typically have a direct relation to the security of the cryptosystem, especially in symmetric cryptography. For certain other cryptosystems, it can be more convoluted.

2.2.3 Side-Channel Attacks on Cryptography

A side-channel attack targets the implementation of a cryptographic application or protocol. It attempts to exploit weaknesses in a specific implementation, so that they may be able to recover a secret key or some other confidential information. There is a wide variety of side-channel attacks. An example is timing attacks, where the attacker repeatedly queries a server, noting the time before getting a response. From there, the adversary may recover some secret data due to the implementation taking various paths in the source code, which consume a different amount of time, depending on some secret data [Koc96] [YGH17]. Another example of a side-channel attack is an electromagnetic attack, such as Van Eck phreaking [VE85], which picks up electromagnetic waves from digital equipment which leaks sensitive data.

The latter example is difficult to protect against purely through the implementation of cryptography, but the former is avoided by ensuring constant-time for all code paths in a program. In the context of cryptography, we define any operation on n bits as *constant-time* if and only if it uses a constant number of CPU cycles, independent of the bits it is operating on.

2.2.4 Secret Key Cryptography

Secret key cryptography, also known as symmetric cryptography, is an essential building block of most modern cryptosystems in use today. Using a symmetric cryptosystem, Alice can encrypt a message with a secret cryptographic key. Bob can only decrypt the resulting ciphertext if he also has the same secret key. There is a symmetry of information between the parties encrypting and decrypting. The goal of a symmetric cryptosystem is to provide confidentiality by making it infeasible to decrypt a ciphertext without the corresponding key. Furthermore, it can provide integrity services to ensure that messages are not tampered with in transit between the communicating parties.

A modern symmetric cryptosystem is instantiated by the following parameters, sets and algorithms [SP19, Definition 2.1]:

- The security parameter, with notation λ, such that the cipher yields λ bits of security.
- A set of keys, $\mathcal{K}$, with a corresponding key generation algorithm, $\mathcal{KG}$, outputting a uniform element in $\{0,1\}^\lambda$.
- A set of plaintexts $\mathcal{P}$.
- A set of ciphertexts, $\mathcal{C}$.
- An encryption algorithm, $\mathcal{E}_k$ that takes a key and a plaintext as input and outputs a ciphertext.
- A decryption algorithm, $\mathcal{D}_k$ that takes a key and a ciphertext as input and outputs the corresponding plaintext, or an error if decryption fails.

We require that $\mathcal{D}_k(\mathcal{E}_k(m)) = m$ for all $m \in \mathcal{P},\ k \in \mathcal{K}$ — the decryption under a specific key of any message encrypted under the same key should yield the original message.

Cryptographic Hash Functions

A hash function is defined in the following way:

$$h : \{0,1\}^* \to \{0,1\}^\lambda.$$

In other words, it takes a bit-string of arbitrary length, the *message*, and outputs a bit-string of a given length, λ, which is referred to as the *message digest* or *hash value*. The hash value should be computable in an efficient and deterministic manner (same message yields the same message digest). For it to be cryptographic, it needs to have the following additional properties [SP19, Chapter 5]:

- *Preimage resistance*: Given a hash value, $t = h(m)$, it should be infeasible to find any message, m', such that $h(m') = t = h(m)$.
- *Seond preimage resistance*: Given a message m, it should be infeasible to find $m' \neq m$ such that $h(m) = h(m')$.
- *Collision resistance*: Finding m, m' where $m \neq m'$ such that $h(m) = h(m')$ should be infeasible.

With infeasible in the above, we mean in relation to the security parameter, λ, meaning an attack should require computing $2^{\mathcal{O}(\lambda)}$ operations.

Cryptographic hash functions are used primarily for two things:

- Hash-based Message Authentication Code (HMAC): Hash a message combined with a secret key, thus implicitly authenticating the message, and ensuring its integrity simultaneously.
- For deriving unpredictable secrets, using the hash function as a random oracle seeded with some input, outputting data indistinguishable from a uniform element in $\{0,1\}^{\lambda}$.

Authenticated Encryption

Authenticated encryption combines the construct of message authentication codes, usually through an HMAC, and a symmetric cipher to create a construct offering both integrity, authentication, and confidentiality. There are three generic constructions for doing this:

- Encrypt-then-MAC — Encrypting the plaintext and then creating a MAC on the ciphertext.
- Encrypt and MAC — Encrypting the plaintext, and attaching a MAC computed on the plaintext.
- MAC-then-Encrypt — Compute a MAC on the plaintext, then encrypt both the MAC and the plaintext together.

A newer paradigm, authenticated encryption with associated data (AEAD), allows for encrypting data while providing authentication for e.g., headers (associated data) in addition to the data itself. This is useful in many scenarios where the header says something about how the data should be handled and may be sent in the clear, but it is still essential to ensure the integrity of the data being sent.

2.2.5 Public Key Cryptography

A problem with symmetric cryptography is that the communicating parties must share secret keys, and the sharing of keys must be accomplished in a way that does not compromise these. This is not always feasible, as it could require physical interaction — transporting books of keys or some other form of key agreement and key transport. This problem has been solved, with the first public key cryptosystem

being published by Diffie and Hellman in 1976 [DH76], marking a paradigm shift, effectively introducing public key cryptography. The Diffie-Hellman Key Exchange (DH) protocol is still a basis for much of modern cryptography.

These cryptosystems are also called asymmetric because encryption keys can be public, while decryption keys are secret — there is an asymmetry in information between the sender, who encrypts, and the receiver, who decrypts.

A public key cryptosystem consists of [SP19, Chapter 6 and 7]:

- A key generation algorithm, which outputs a public key, p, and a secret key, s.
- A message set, dependent on the public key, $\mathcal{M}_p$.
- An encryption algorithm, $\mathcal{E}$, taking a message from $\mathcal{M}_p$ and the public key, p, as input, outputting a ciphertext.
- A decryption algorithm, $\mathcal{D}$, taking a ciphertext and the private key, s, as input, outputting either the corresponding plaintext or an error if the decryption fails.

For a public key cryptosystem, it should be infeasible for any attacker to decrypt an encrypted message without the private key.

Two prominent examples of public key cryptosystems are the Diffie-Hellman [DH76] and RSA [RSA78] cryptosystems.

Digital Signature Schemes

The previously explained schemes protect against passive attacks when used alone. Protecting against active adversaries require us to authenticate the party, or parties, with whom we are communicating across a communications medium. To do this, we use a digital signature scheme, which consists of the following three algorithms [SP19, Definition 8.1]:

- A key generation algorithm, $\mathcal{K}$ that outputs a verification key, k, which should be public, and the signing key, s, which should be kept secret.
- A signing algorithm, $\mathcal{S}$, taking a message, m, and a signing key, and outputting a signature, τ, on m under s.
- A verification algorithm, $\mathcal{V}$, taking a message m, a signature τ, and a verification key v, outputting either success or failure depending on whether τ is a valid signature on m by the corresponding s.

Public paramaters:

Finite field $\mathbb{F}_p$ and generator $g \in \mathbb{F}_p$.

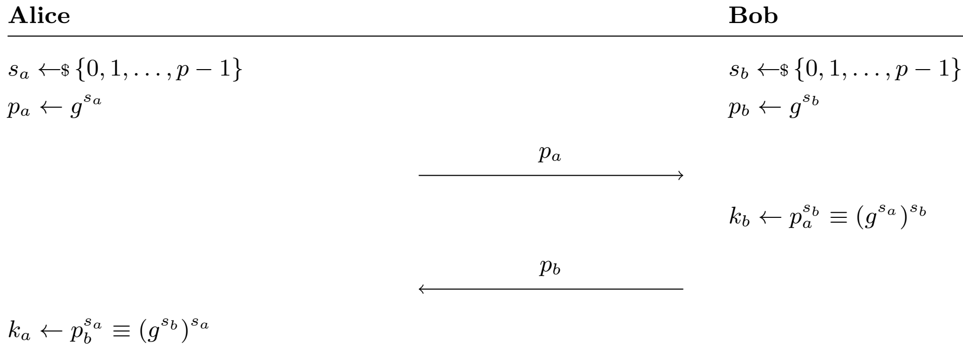


Figure 2.2: A version of the Diffie-Hellman key exchange protocol over finite fields.

2.2.6 Key Exchange

In most modern cryptographic applications, we wish to achieve a shared secret key using public key cryptography, which is then used to derive a key to be used with symmetric ciphers. This is due to public key encryption being inefficient in terms of computation and bandwidth compared with modern secret key cryptosystems.

The first key exchange protocol, the DH [DH76] protocol, versions of which is still in use, is constructed as illustrated in Figure 2.2.

It can be shown that Alice and Bob achieve the same shared secret, $k_a \equiv k_b$, and furthermore that it is a hard problem — using classical computing — to recover either secret keys, or the shared key. In modern cryptosystems, Diffie-Hellman is performed over elliptic curves, and not over finite fields as illustrated above. However, the overall protocol follows the same structure.

There are ways in which we can authenticate the participants during the key exchange, for instance through digital signatures, or out-of-band verification of keys. We call this an Authenticated Key Exchange (AKE).

Key Encapsulation Mechanisms

A KEM is similar in structure to public key encryption, but specialized in facilitating key agreement. The key agreed upon is then typically used in symmetric ciphers.

A KEM consists of the following:

- A key generation algorithm $\mathcal{K}$ that outputs an encapsulation key p and a decapsulation key s.
- An encapsulation algorithm $\mathcal{E}$ that takes an encapsulation key and outputs a secret key, k, and an encapsulation of the key.
- A decapsulation algorithm $\mathcal{D}$ that takes a decapsulation key and an encapsulation and outputs either a secret key, k, or an error in case of decapsulation failure.

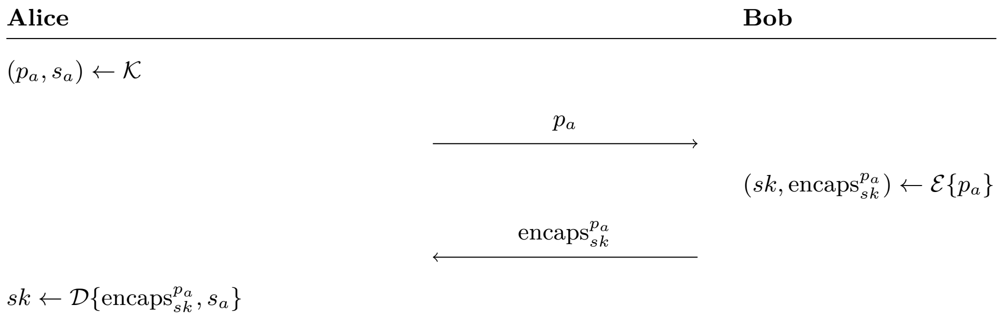


Figure 2.3: A generic KEM to achieve a shared secret between two parties.

By publishing an encapsulation key, there need not be a key exchange. Instead, by the initiator uses the encapsulation algorithm on the recipients public key, receiving the shared secret and the encapsulated message, which is transmitted to the recipient, who decapsulates it and retrieves the shared secret using their decapsulation key. A generic KEM is showed in Figure 2.3.

2.3 Ring Learning With Errors Cryptography

Regev [Reg05] first introduced the Learning With Errors (LWE) problem in 2005, which consists of finding solutions to systems of linear equations where a subset of the given equations has a small added error, e. Given a set of n linearly independent vectors, $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{F}_p^n$ and a set of n results on the form $b_i = \mathbf{a}_i \cdot \mathbf{s} + e_i$ — find $\mathbf{s}$. It is believed that, if n is large enough, this is a difficult problem. LWE forms the basis of many proposed post-quantum cryptosystems, such as FrodoKEM [NAB+17], while subtypes of the LWE problem form the basis of other candidates.

One such subtype is the LWE problem over algebraic rings. First formulated by Lyubashevsky, Peikert, and Regev [LPR13], they add more structure to the LWE problem, which proves to be advantageous as it gives a quadratic reduction in key

sizes. This makes it more suitable for practical applications. Specifically they suggest an LWE variant over cyclotomic power-of-two rings, $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where n is a power of two and q is a sufficiently large prime number such that $q \mod 2n \equiv 1$. In RLWE, we deal with polynomials from R_q, and not vectors as in LWE. Each coefficient is a representative of $\mathbb{Z}_q$, and the polynomial itself is reduced modulo $x^n + 1$.

In RLWE, the secret and error polynomials are generated by sampling from the ring using a probability distribution — typically a discrete Gaussian distribution centered at zero. The public polynomial is either agreed upon or generated as a uniform element of the ring.

The NIST standardization process currently has two candidates based on the RLWE problem — NewHope [ADPS15] and LAC [LLZ$^+$18]. LAC operates on comparatively small numbers, focusing more on error correction than NewHope, which is a highly efficient cryptosystem tested in practice [Bra16], and with many similarities to the different GKE we will evaluate in this thesis. Another interesting implementation, Bos *et al.* [BCNS15] demonstrated the use of a plain RLWE key exchange in practice, applying it to the Transport Layer Security (TLS) protocol.

2.3.1 Generic RLWE Key Exchange

A general RLWE cryptosystem is instantiated by the following parameters:

- An ideal generated by an irreducible polynomial on the form $f(x) = x^n + 1 \in \mathbb{Z}_q[x]$ where n is a power of two, which generates the quotient ring, $R_q = \mathbb{Z}_q[x]/\langle f(x)\rangle$.
- The modulus, q, a prime number such that $q \mod 2n \equiv 1$.
- A public polynomial, $\mathbf{a} \in R_q$.
- A probability distribution χ_σ on R_q for generating secrets and errors, with parameter σ. We denote sampling a polynomial $\mathbf{a}$ from R_q using χ_σ as $\mathbf{a} \leftarrow \chi_\sigma$.

The secret key, $\mathbf{s}$ and the error, $\mathbf{e}$ are generated by extracting n coefficients using χ on R_q n times, independently. The public key is then $\mathbf{p} = \mathbf{s} \cdot \mathbf{a} + \mathbf{e}$. A key exchange can be performed by sharing public keys between two parties, and then multiplying their secret key with their public key as $\mathbf{s}_0 \cdot \mathbf{p}_1 = \mathbf{s}_0 \cdot \mathbf{a}\mathbf{s}_1 + \mathbf{s}_0\mathbf{e}_1 \approx \mathbf{s}_1 \cdot \mathbf{a}\mathbf{s}_0 + \mathbf{s}_1\mathbf{e}_0 = \mathbf{s}_1 \cdot \mathbf{p}_0$, given that the error distribution is "short" for some definition of short, typically depending on the parameter set. We illustrate this generic key exchange in Figure 2.4, where we see Alice and Bob achieve $\mathbf{b}_a$ and $\mathbf{b}_b$, respectively. Due to this being an approximate equality — $\mathbf{b}_a \approx \mathbf{b}_b$ — an error reconciliation is typically used to achieve a fixed shared secret.

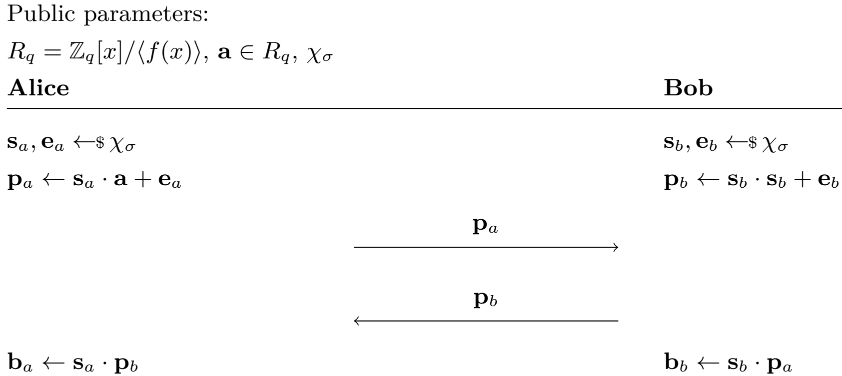


Figure 2.4: A generic RLWE key exchange where Alice and Bob achieve an approximate shared secret ring element.

2.3.2 Key Reconciliation Protocols

The purpose of this section is to introduce the concept of key reconciliation to the reader. It will later be used in conjunction with a group key exchange to achieve a shared secret.

In RLWE based cryptosystems, the participants usually achieve an approximate equality. As the resulting RLWE-element is passed through a Key Derivation Function (KDF) to achieve a key for use in symmetric cryptosystems, it is required that the approximate equality be turned into an exact equality due to the fact that two approximately equal elements passed through a KDF will give widely different results.

In practice, the approximate equality is turned into a definite shared secret with high probability through a reconciliation protocol. We define two functions, *recMsg* and *recKey*:

$$recMsg\colon R_q \to \{0,1\}^n \times \{0,1\}^n$$

$$recKey\colon R_q \times \{0,1\}^n \to \{0,1\}^n.$$

Where $\{0,1\}^n$ denotes a vector of length n with elements from the set $\{0,1\}$. Following an RLWE key exchange, Alice and Bob have two ring elements, $\mathbf{b}_a, \mathbf{b}_b$ that are approximately the same. Now, a generic reconciliation protocol between these two parties is performed in the manner shown in Figure 2.5. The initiator generates a reconciliation vector and the fixed shared secret using $recMsg$. The reconciliation vector is sent to the responder, who is able to determine the fixed shared secret

Public parameters: R_q, $\mathbf{a} \in R_q$, χ_σ

Alice		**Bob**
$(\mathbf{rec}, sk) \leftarrow recMsg(\mathbf{b}_a)$		
	$\xrightarrow{\mathbf{rec}}$	
		$sk' \leftarrow recKey(\mathbf{b}_b, \mathbf{rec})$

Figure 2.5: A generic key reconciliation protocol for RLWE.

recKey using the reconciliation vector and the previously achieved approximate shared secret.

Now, $sk = sk'$ with a high probability.

2.3.3 Attacks on RLWE Cryptography

In order to provide a useful instantiation of an RLWE cryptosystem, we need to know how they are attacked, so we may pick parameters resistant to these attacks. However, discussing these in-depth is out of scope for this thesis, and we must defer to the provided references for extended discussions, most notably the initial RLWE paper [LPR13], Peikert's paper on safe instantiations [Pei16], and Player's doctorate on parameter selection for lattice cryptography [Pla18].

The security of an RLWE cryptosystem is based on the difficulty of getting the secret ring element, $\mathbf{s}$, from a public key $\mathbf{p} = \mathbf{s} \cdot \mathbf{a} + \mathbf{e}$. We believe that finding $\mathbf{s}$ given $\mathbf{p}$ and $\mathbf{a}$ is hard when $\mathbf{a}$ is a uniform ring element and $\mathbf{s}$ and $\mathbf{e}$ are from non-trivial discrete Gaussian distributions. This is called *the search-RLWE problem* (sometimes just referred to as *search-RLWE*) — note that there is an equivalence between finding the error, $\mathbf{e}$, and the secret key, $\mathbf{s}$. A related problem is *the decision-RLWE problem*, where an attacker must distinguish between a public key, $\mathbf{p}$ and a uniform ring element [LPR13].

The main known attacks on an RLWE instance is through reducing it to an instance of an ideal lattice, given that such a reduction is possible, which is tighly coupled with a secure instance of RLWE [Pla18] [Pei16] [APS15]. Using this strategy, solving search-RLWE is believed to be at least as hard as solving the approximate Shortest Vector Problem (SVP) problem [CDW16] in an ideal lattice. This problem consists of finding a non-trivial vector which is smaller than some given approximation, related by a factor of $\mu \in \mathbb{Z}$ to a shortest non-zero vector in the given lattice [SP19, p. 349].

2.4 RLWE Group Key Exchange

Currently, only the following papers relate to post-quantum group key exchange protocols based on the RLWE problem: Ding *et al.* [JD12], Apon *et al.* [ADSGK19] and Choi *et al.* [CHK20].

In this section, we will examine the following protocols for GKE:

- **Proposal 1**: Ding *et al.* [JD12],
- **Proposal 2**: Apon *et al.* [ADSGK19] and,
- **Proposal 3**: Choi *et al.* [CHK20].

Common to all proposals is that they are based on the RLWE problem, and use a Gaussian distribution to generate secret and error polynomials. If nothing else is specified, the protocol uses the parameters $R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$, where χ is a probability distribution centered at 0, with a non-zero standard deviation. Furthermore, the modulus, q is a "large" prime number, for some definition of "large" — typically larger than $8-$bit — such that $q \mod 2n \equiv 1$. Let bold lowercase letters denote polynomials of dimension n, and let N be the number of participants. Indexes are modulo N. Finally, let $\mathcal{H}$ denote a cryptographic hash function.

2.4.1 Proposal 1: Ding

1. Each participant, P_i, generates a secret key and an error $\mathbf{s}_i \xleftarrow{\chi} R_q$, $\mathbf{e}_i \xleftarrow{\chi} R_q$, and calculates its public key $\mathbf{p}_i^0 = \mathbf{s}_i \cdot \mathbf{a} + \mathbf{e}_i$.
2. Participant P_i sends its public key, $\mathbf{p}_i^0$, to participant P_{i+1} (indexes are modulo N).
3. All participants P_{i+j} calculate $\mathbf{p}_i^j = \mathbf{s}_{i+j} \cdot \mathbf{p}_i^{j-1} + 2\mathbf{e}_i^j$ and send $\mathbf{p}_i^j$ to participant P_{i+j+1}.
4. Participant P_0 calculates $\mathbf{k}_0 = \mathbf{p}_1^{N-2} \cdot \mathbf{s}_0 + 2\mathbf{e}'_0$, and generates a reconciliation vector, $\mathbf{rec}, k_0 = recMsg(\mathbf{k}_0)$, which is broadcast to all P_i.
5. Participant P_i first calculates $\mathbf{k}_i = \mathbf{p}_{i+1}^{N-2} \cdot \mathbf{s}_i + 2\mathbf{e}'_i$.
6. All participants recover the key via a reconciliation technique — $k_i = recKey(\mathbf{rec}, \mathbf{k_i})$, and the session key is set to $sk_i \leftarrow \mathcal{H}(k_i)$.

This protocol requires $\mathcal{O}(N)$ rounds, thus scaling poorly. However, it was the first proposed GKE based on RLWE, and is included here as helpful context.

2.4.2 Proposal 2: Apon

The proposal by Apon *et al.*, is similar in structure to the Burmester-Desmedt [BD95] protocol for GKE using the Diffie-Hellman problem as its basis. This protocol requires two Gaussian probability distributions, χ_1, χ_2 on the ring R_q, to ensure that the sum of "neighbourhood"-keys are close to uniform.

1. Each participant, P_i, generates a secret key and an error, $\mathbf{s}_i \xleftarrow{\chi_{\sigma_1}} R_q$, $\mathbf{e}_i \xleftarrow{\chi_{\sigma_1}} R_q$, and calculates its public key $\mathbf{p}_i = \mathbf{s}_i \cdot \mathbf{a} + \mathbf{e}_i$. The public key is broadcast to the group.

2. Participant P_0 samples $\mathbf{e}'_0 \xleftarrow{\chi_{\sigma_2}} R_q$, while P_i for $i \neq 0$ samples $\mathbf{e}'_i \xleftarrow{\chi_{\sigma_1}} R_q$. Then all P_i calculate "neighbourhood"-keys: $\mathbf{X}_i = (\mathbf{p}_{i+1} - \mathbf{p}_{i-1})\mathbf{s}_i + \mathbf{e}'_i$, which are broadcasted to the group.

3. All participants compute

$$\mathbf{b}_i = N \cdot \mathbf{p}_{i-1}\mathbf{s}_i + (N-1)\mathbf{X}_i + (N-2)\mathbf{X}_{i+1} + \ldots + \mathbf{X}_{i+N-2}$$

   Participant $N-1$ samples $\mathbf{e}''_{N-1} \xleftarrow{\chi_{\sigma_1}} R_q$ and recomputes $\mathbf{b}_{N-1} = \mathbf{b}_{N-1} + \mathbf{e}''_{N-1}$. From this, P_{N-1} computes the reconciliation vector **rec** and key, k_{N-1} as $\mathbf{rec}, k_{N-1} = recMsg(b_{N-1})$. **rec** is broadcast to the group.

4. Participants $i \neq N-1$ calculate the key using $\mathbf{b_i}$ and **rec** — $k_i = recKey(\mathbf{rec}, b_i)$.

5. Finally, the session key is set to $sk_i \leftarrow \mathcal{H}(k_i)$

Apon *et al.* [ADSGK19] provide the following equations for correctness and security of their protocol:

$$(N^2 + 2N) \cdot \sqrt{n} \cdot \rho^{3/2} \cdot \sigma_1^2 + \left(\frac{N^2}{2} + 1\right) \cdot \sigma_1 + (N-2) \cdot \sigma_2 \leq \beta_{rec} \tag{2.1}$$

$$2N\sqrt{n}\lambda^{3/2} \cdot \sigma_1^2 + (N-1) \cdot \sigma_1 \leq \beta_{\text{Rényi}} \tag{2.2}$$

$$\sigma_2 = \Omega(\beta_{\text{Rényi}}\sqrt{n/\log \lambda}). \tag{2.3}$$

We shall henceforth refer to the protocol proposal by Apon as AponGKE.

2.4.3 Proposal 3: Choi

The proposal by Choi *et al.* is a suggested improvement upon AponGKE, but they base their proposal on an RLWE-version of the Dutta-Barua protocol for GKE [DB05]. As a result, there are some similarities.

1. Each participant, P_i generates a secret key and an error, $\mathbf{s}_i \xleftarrow{\chi_{\sigma_1}} R_q$, $\mathbf{e}_i \xleftarrow{\chi_{\sigma_1}} R_q$, and calculates its public key $\mathbf{p}_i = \mathbf{s}_i \cdot \mathbf{a} + \mathbf{e}_i$. The public key is broadcast to the group.

2. Participant P_0 samples $\mathbf{e'}_0 \xleftarrow{\chi_{\sigma_2}} R_q$, while P_i for $i \neq 0$ samples $\mathbf{e'}_i \xleftarrow{\chi_{\sigma_1}} R_q$. Then all P_i calculate "neighbourhood"-keys: $\mathbf{X}_i = (\mathbf{p}_{i+1} - \mathbf{p}_{i-1})\mathbf{s}_i + \mathbf{e'}_i$, which are broadcasted to the group.

3. Participant P_{N-1} samples $\mathbf{e}''_{N-1} \xleftarrow{\chi_{\sigma_1}} R_q$ and computes

   $$\mathbf{Y}_{N-1,N-1} = \mathbf{X}_{N-1} + \mathbf{p}_{N-2}\mathbf{s}_{N-1} + \mathbf{e''}_{N-1}.$$

   Then, P_{N-1} computes

   $$\mathbf{Y}_{N-1,N-1+j} = \mathbf{X}_{N-1+j} + \mathbf{Y}_{N-1,N-2+j}$$

   for $\forall j \in \{1, \ldots, N-1\}$. Finally, P_{N-1}'s approximate secret, b_{N-1} is calculated as

   $$b_{N-1} = \sum_{j=0}^{N-1} \mathbf{Y}_{N-1,N-1+j}.$$

   Now, P_{N-1} generates the reconciliation vector, **rec**, and the key k_{N-1} from $\mathbf{rec}, k_{N-1} = recMsg(\mathbf{b}_{N-1})$. **rec** is broadcasted to the group.

4. All other participants P_i calculate

   $$\mathbf{Y}_{i,i} = \mathbf{X}_i + \mathbf{p}_{i-1}\mathbf{s}_i.$$

   Then they compute

   $$\mathbf{Y}_{i,i+j} = \mathbf{X}_{i+j} + \mathbf{Y}_{i,i-1+j}$$

   for $\forall j \in \{1, \ldots, N-1\}$. The approximate polynomial is calculated as

   $$\mathbf{b}_i = \sum_{j=0}^{N-1} \mathbf{Y}_{i,i+j}.$$

   Finally, the shared secret is recovered using an error reconciliation function — $k_i = recKey(\mathbf{b}_i, \mathbf{rec})$.

5. Then, the session key is set to $sk_i \leftarrow \mathcal{H}(k_i)$.

We will refer to Choi's protocol proposal as ChoiGKE.

Choi *et al.* [CHK20] is secure due to the same equations as AponGKE, however the correctness bound is lower, and AponGKE's equation 2.1 becomes equation 2.4 for ChoiGKE.

$$\frac{1}{2}\left(N^2 - N\right)\sqrt{n}\rho^{3/2}\sigma_1^2 + \frac{1}{2}\left(N^2 + 3N\right)\sigma_1 + (N-2)\sigma_2 \leq \beta_{rec} \tag{2.4}$$

2.4.4 Peikert Reconciliation For Groups

In Subsection 2.3.2 we introduced the concept of key reconciliation. This section will introduce a specific protocol we can use to do this. The below is based on Peikert [Pei14] and BCNS [BCNS15].

When performing an RLWE key exchange, the participants end up with ring elements that are approximately the same in every dimension — seen from a lattice perspective, or in every coefficient from a polynomial ring perspective — and they need to agree on a shared secret from this approximate equality. The general idea is that a specific group member generates a reconciliation element based on their approximate shared value, and shares this with the rest of the group. All participants then use this element to recover the same shared secret.

Assume that we have performed a RLWE-based key exchange, and all participants end up with approximately the same values $\mathbf{b}_0 \approx \mathbf{b}_1 \approx \ldots \approx \mathbf{b}_N$. We must now reconciliate the errors such that all participants end up with the same key — $k_0 = k_1 = \ldots = k_N$. Furthermore, assume that participant P_j will be creating the reconciliation vector for the group.

We will here look at one technique for error reconciliation that is simple and efficient, which we will use when instantiating our chosen protocols at a later point. Peikert [Pei14] proposes a method for turning approximate agreement into a definite shared secret — key reconciliation — by extracting a single bit from each coefficient such that both participants end up with the same value. Extending this to the case where we have ≥ 3 participants is trivial, as all protocol participants will end up with an approximate shared secret. Then one participant creates the reconciliation vector, which is broadcast to all participants who are then able to recover the shared secret. The relevant reconciliation technique for prime moduli will be explained here. First we define the function *dbl*,

$$dbl\colon \mathbb{Z}_q \to \mathbb{Z}_{2q} \mapsto dbl(x) = 2x + e. \tag{2.5}$$

Where e is drawn from a discrete Gaussian distribution centered at zero, with a magnitude less than or equal to one (i.e. the probability that we draw 0 is 0.5, while both 1 and -1 have a probability of 0.25).

Next, we define the functions for rounding — $\lfloor\cdot\rceil_{q,2}$ — and cross-rounding — $\langle\cdot\rangle_{q,2}$.

$$\lfloor\cdot\rceil_{q,2} \colon \mathbb{Z}_q \to \mathbb{Z}_2 \mapsto \lfloor x\rceil_{q,2} = \left\lfloor \frac{2}{q} \cdot x \right\rceil \mod 2 \tag{2.6}$$

$$\langle\cdot\rangle_{q,2} \colon \mathbb{Z}_q \to \mathbb{Z}_2 \mapsto \langle x\rangle_{q,2} = \left\lfloor \frac{4}{q} \cdot x \right\rfloor \mod 2 \tag{2.7}$$

Defining $I_0 = \{0, 1, \ldots, \lfloor \frac{q}{4} \rceil\}$ and $I_1 = \{\lceil \frac{3q}{4} \rceil, \ldots, q-1\}$, which are disjoint sets of coefficients from $\mathbb{Z}_q$, and the set $E = \{\lceil \frac{q}{4} \rceil, \ldots, \lfloor \frac{3q}{4} \rfloor\}$ Finally, define the reconciliation function, $rec \colon \mathbb{Z}_{2q} \times \mathbb{Z}_2 \to \mathbb{Z}_2$ as

$$rec(w, b) = \begin{cases} 0, & w \in I_b + E \mod 2q \\ 1, & \text{otherwise} \end{cases} \tag{2.8}$$

where $b \in \{0, 1\}$.

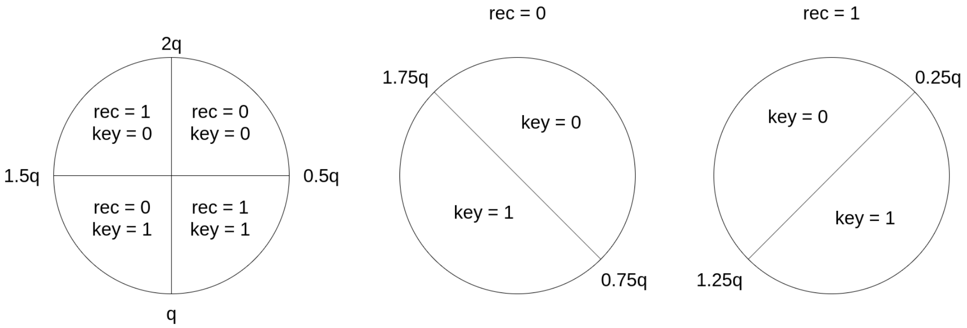


Figure 2.6: An illustration of the Peikert reconciliation function for an odd modulus. The leftmost subfigure shows how the reconciliation vector and shared secret is generated, while the other subfigures show how a recipient with an approximate shared ring element may achieve the same shared secret using the reconciliation vector.

The kth value of the reconciliation vector, $\mathbf{r}$, is generated by participant P_j by taking the kth coefficient from $\mathbf{b}_j$, applying *dbl* on it, and then applying the crossrounding function on the result — $\mathbf{r}_k = \langle dbl(\mathbf{b}_{j,k})\rangle_{q,2}$. The key for each participant P_i is constructed using the *rec*-function on each value of $\mathbf{b}_i$ inputting the result of *dbl* on

the kth coefficient and the kth reconciliation vector value. A figure illustrating the process of Peikert error reconciliation is shown in Figure 2.6.

We note that Peikert reconciliation is an improvement upon previous work done by Ding *et al.* [JD12]. The reconciliation mechanism by Ding *et al.* resulted in a biased result when using an odd modulus, which is a challenge Peikert's proposal overcomes through the use of the *dbl*-function.

Peikert [Pei14] shows that this technique can be used to recover a shared secret, when participants have approximate equality. Practical experiments in applying a version of this reconciliation technique in an RLWE key exchange for TLS [BCNS15], show a negligible error rate.

2.5 Transforming GKE to GAKE

A GKE protects against a passive adversary, capable of recording protocol runs. However, it does not protect against active adversaries, controlling or influencing the medium by which messages are communicated across. Protecting against these types of attacks require us to authenticate the parties with which we communicate. In this setting, we require a GAKE. In this context, a compiler is a method used to go from a GKE to a GAKE, mutually authenticating all participants. This means that all participants authenticate all other participants.

The major difference between the two compilers explained below, is the accompanying theoretical framework. Katz-Yung provides only the compiler, and thus protocols compiled only have provable AKE-security in a static context. Bresson *et al.* however, work in a framework consisting of several algorithms for a dynamic group setting, allowing participants to join and leave in a authenticated and secure manner.

In the following, let $a\|b$ denote the concatenation of a and b.

2.5.1 The Katz-Yung compiler

It is possible to transform any general GKE to a GAKE by adding digital signatures. Katz and Yung [KY03] propose a compiler, turning any constant round GKE into a GAKE with an additional round. This is accomplished in the following way.

1. Initialization: An ephemeral verification/signing key pair is generated by all communicating parties.

2. Each participant, P_i with unique identifier i, selects a nonce, n_i, at random from $\{0,1\}^\lambda$, and broadcasts $i\|0\|n_i$, where 0 denotes the sequence number of

the message, which is defined to be zero. The nonces of all parties, along with the identifiers of these are stored as state variables for every participant.

3. The next rounds of the GAKE are done according to the original GKE, but modified to add the corresponding nonce and a digital signature on the messages sent. Every time a participant receives a message, the signature is verified. If the signature is invalid, the protocol aborts.

4. If nothing went wrong, all participants end up with the same shared secret, and are all mutually authenticated.

2.5.2 Compiler for Authenticated GAKE by Bresson et al.

Another technique for transforming a GKE to a GAKE is provided by Bresson *et al.* [BMS06]. This compiler proposes the following:

1. Initialization: An ephemeral verification/signing key pair is generated by all communicating parties. Furthermore, all parties generate a nonce, n_i, which is broadcast.

2. Each participant, P_i with unique identifier i, selects a nonce, n_i at random from $\{0,1\}^\lambda$, and broadcasts $i\|0\|n_i$, where 0 denotes the sequence number of the message, starting at zero. The nonces of all parties, along with the identifiers of these are stored as state variables for every participant. Having received all the nonces, each party creates a session identifier, $sid = 0\|n_0\|1\|n_1\|\ldots\|N-1\|n_{N-1}$.

3. The next rounds of the GAKE are done according to the original GKE, but modified such that when a message m, with sequence number k is to be sent, a signature, τ on $k\|m\|sid$ is concatenated with the original message, so $i\|k\|m\|\tau$ is sent.

4. When a party P_i receives $j\|k\|m\|\tau_j$ from P_j, he checks if j is a valid participant identifier by searching through the session identifiers. Furthermore, he checks whether k is an expected sequence number. Finally, the signature is checked using the verification key belonging to P_j, which P_i has stored. If any checks fail, the protocol run is terminated by P_i.

5. The parties compute the same shared secret as in the original GKE.

Chapter 3
Protocol Instantiation

In this chapter, we will provide an instantiation of two protocols for post-quantum group key exchange. By instantiation, we mean defining the variables to be used in a practical implementation and the specific methods we will use to perform certain operations required by the protocols, like sampling from Gaussians and performing polynomial convolution.

The goal of our instantiation is to provide a reasonable level of security while still providing a set of parameters that allow for an efficient implementation. We aim for a security level fitting into to what NIST defines to be category 1 [NIS16, 4.A.5], which is the lowest level corresponding to roughly 128 bits of security in the classical model — i.e., not considering attacks using quantum computers.

3.1 A Parameter Proposal

For AponGKE and ChoiGKE, we propose the following set of parameters:

- $\lambda = \rho = 256$,
- $n = 1024$,
- $q = 45510033409$, a 35-bit prime,
- $3 \leq N \leq 20$,
- $\sigma_1 = 2$,
- $\sigma_2 = 94371960$.

These parameters should satisfy the general requirements for hardness of the RLWE problem, and that the best method for attacking an RLWE cryptosystem, at this time, is the same as the approximate SVP problem in an ideal lattice, which we

previously discussed in Section 2.3.3. We select the security parameters, λ, the computational security parameter, and ρ, the statistical security parameter to be 256 — we discussed these in Section 2.2.2. This will help us reach an acceptable level of security — satisfying the equations for AponGKE and ChoiGKE in accordance with our security and performance targets. The dimension is chosen as $n = 1024$, which compared with other practical RLWE cryptosystems such as NewHope [ADPS15] and Singh [Sin15] is a reasonable — previously seen as conservative — dimension for an acceptable security level. Selecting parameters with some margin of security is advantageous to accommodate the possibility of future results weakening its security, given that RLWE-based key exchange, and RLWE GKE in particular, is a young field, and better attacks are likely to occur in the future.

We recall the equations from Section 2.4.2, given by Apon *et al.* [ADSGK19] for their protocol:

$$(N^2 + 2N) \cdot \sqrt{n} \cdot \rho^{3/2} \cdot \sigma_1^2 + \left(\frac{N^2}{2} + 1\right) \cdot \sigma_1 + (N-2) \cdot \sigma_2 \leq \beta_{rec} \tag{3.1}$$

$$2N\sqrt{n}\lambda^{3/2} \cdot \sigma_1^2 + (N-1) \cdot \sigma_1 \leq \beta_{Rényi} \tag{3.2}$$

$$\sigma_2 = \Omega(\beta_{Rényi}\sqrt{n/\log\lambda}). \tag{3.3}$$

The changes made by Choi *et al.* [CHK20] presented in full in Section 2.4.3, only affect the equation for correctness, i.e. the first equation, which becomes:

$$\frac{1}{2}\left(N^2 - N\right)\sqrt{n}\rho^{3/2}\sigma_1^2 + \frac{1}{2}\left(N^2 + 3N\right)\sigma_1 + (N-2)\sigma_2 \leq \beta_{rec}. \tag{3.4}$$

We see that equations 3.1 and 3.4 are both dominated by the large term σ_2, and that equation 3.4 will be smaller than equation 3.1 for all valid values of N, n, ρ and λ. As a result we are able to use equation 3.1 in the process of selecting parameters for both protocols.

We select $\sigma_1 = 2$, due to Peikert [Pei16] showing that this is the lowest possible value that still makes the RLWE problem hard enough. Furthermore, selecting a value for σ_1, the ring dimension and the security parameters induce a lower bound on σ_2 through the equations of Apon *et al.* This bound, in turn, gives a minimum value for the modulus, q. Smaller moduli and standard deviations are advantageous, as calculations can be done more effectively, which is consistent with the overall design goal — security and efficiency. Selecting the smallest possible modulus increases the relative noise, which additionally raises the security level of the protocol, compared with larger moduli and a greater signal-to-noise ratio. The signal-to-noise ratio

also decreases with the number of participants. At some point, this will make it impossible for all parties to recover the shared secret with an insignificant error rate.

Regarding Ding et al. We choose not to propose an instantiation of the protocol by Ding *et al.*, as its round complexity is linear in the number of participants, making supporting even small to medium-sized groups computationally expensive, in addition to the network delays caused by a significant number of rounds. In Section 2.4.1, we included it as a context for the environment in which AponGKE and ChoiGKE have been proposed, as Ding provides the initial post-quantum GKE protocol based off the RLWE problem. Due to the issues with round complexity we choose to focus on AponGKE and ChoiGKE, disregarding Ding.

3.2 Efficient Implementation Building Blocks

In evaluating group key exchange, an efficient prototype provides insights we could not otherwise obtain. Through experiences in the implementation process and the performance of the prototype itself, and through comparing it with other solutions, we obtain new knowledge about the protocol itself. This gives insight into the real-world feasibility of the protocol, and where it may be useful. As solutions we would like to compare with often are optimized, we must also provide an efficient prototype. As RLWE-based cryptosystems are "new", we are not aware of all possible performance enhancements; however, we will attempt to use the techniques known to us for achieving an efficient implementation so that our evaluation is based on the "state of the art" knowledge. This also makes a comparison with existing optimized solutions fair and useful.

In this section, we will explain how we implement the instantiations provided above, using specific techniques for the most computationally expensive operations — polynomial convolution and Gaussian sampling.

3.2.1 Polynomial Convolution

Given $f(x), g(x) \in R_q = \mathbb{Z}_q[x]/\langle x^{1024} + 1\rangle$, the convolution $(f * g)(x)$ (i.e., the multiplication of the polynomials in the polynomial ring) can efficiently be calculated in a number of ways. Bernstein [Ber08] performs a comparison of some methods that may be applied, and Roche [Roc09] performs a comparison between Fast Fourier Transform (FFT) based approaches and Karatsuba [WP06]. For our use case, Roche shows that the FFT-based NTT approach both has the best asymptotic computational complexity, and is the fastest for our specific selected dimension. Scott [Sco17] further optimizes the implementation of NTT, by using the forwards transform of Cooley-Tukey adjusted for NTT, but then using the Gentleman-Sande algorithm for the inverse. We have included the Cooley-Tukey forwards transform in Algorithm

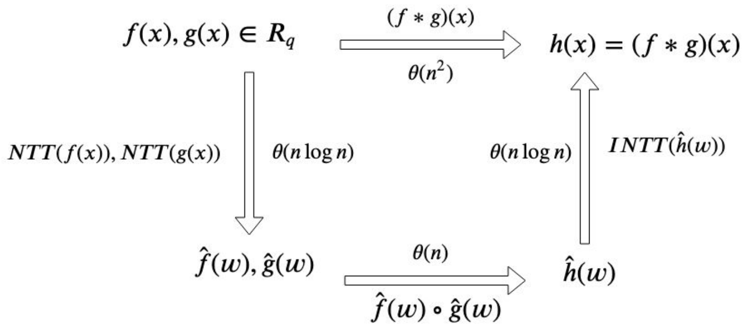


Figure 3.1: The equivalence relations that allow us to speed up the polynomial convolution through transforming to the NTT domain and performing a pointwise multiplication. The innermost text show the computational complexity of the associated function depicted on the outside.

3.1, and the Gentleman-Sande inverse transform in Algorithm 3.2, based on the descriptions by Longa and Naehrig [LN16], and Agarwal [AB75]. This approach yields a computational complexity of the convolution operation of $\Theta(n \log n)$, compared to the naive solution which has a run-time complexity of $\mathcal{O}(n^2)$. As each participant in a protocol run uses $2N + 1$ ring multiplication operations, this improvement in the computational complexity speeds up the protocol significantly. In Figure 3.1 we see the equivalence mapping, showing the process of performing convolution through NTT.

Using the NTT to speed up the convolution process, requires us to transform polynomials into NTT-domain, and after the pointwise multiplication, back to R_q. This computation takes as input the powers of $2n$th roots. These constants are precomputed using NTT4RLWE[1], which is partially adapted from the BLISS digital signature project [DDLL13][2] for this thesis. For performance, the powers of $2n$th roots of unity are included as constants in the source code (in bit-reversed order). This is similar to other approaches employed by other RLWE-based cryptosystem implementations, and is the most performant option.

The approach of including variables as constants saves a significant amount of computation power. However, it also limits the participant count to, as defined above, 20, as we are unable to dynamically adjust the modulus (and other variables) to accommodate an increased participant count. We believe that this is the best approach regardless, and supporting more participants is out of the scope of this thesis.

[1] Available at https://github.com/simenbkr/ntt4rlwe, commit ID `f557f35`.

[2] Specifically https://github.com/SRI-CSL/Bliss/blob/master/ntt_variants/make_red_tables.c, commit ID `82a9d04`

Algorithm 3.1 Cooley-Tukey Forwards NTT

Input: A polynomial, $\mathbf{x} \in R_q$, and $\omega = \{\omega_0, \omega_1, \ldots, \omega_n\}$ — powers of $2n$th roots of unity, stored in bit-reversed order.

Output: $\hat{\mathbf{x}} = \mathbf{x} \leftarrow NTT(\mathbf{x})$

```
t ← n
for m ← 1; m < n; m = 2m do
    t ← t/2
    for i ← 0; i < m; i ← i + 1 do
        j1 ← 2it
        j2 ← j1 + t − 1
        S ← ω[m + i]
        for j ← j1; j ≤ j2; j ← j + 1 do
            U ← x[j]
            V ← S · x[j + t]                ▷ Arithmetic in the ring, R_q.
            x[j] ← U + V
            x[j + t] ← U − V
        end for
    end for
end for
return x
```

3.2.2 Sampling from an Error Distribution

In RLWE, the error distributions are typically described as discrete Gaussian distributions in papers. The usage of these distributions is crucial to the security and correctness of the cryptosystems. In practice, these are implemented in widely different manners, due to the complexity involved in implementing efficient and cryptographically secure samplers. Some notable papers argue that high-quality sampling is not necessary, such as the NewHope paper [ADPS15]. Their practical approach includes approximating a discrete Gaussian distribution through the centered binomial distribution. While others, such as the FALCON digital signature project [FHK+18], focus on creating practical, high-quality discrete Gaussian samplers. There exist many projects on creating such samplers, notably Zhao *et al.* [ZSS18] and Karmakar *et al.* [KRVV19].

However, the implementation of efficient and cryptographically secure Gaussian distributions is deemed out of scope for this thesis. As such, our approach will include using the flexible, albeit not constant-time implementation of Albrecht *et al.* — DGS[3] [AW18], as it fits our other criteria relating to performance and closeness to a proper discrete Gaussian.

A shortcoming of this tool is that it does not support standard deviations as high

[3] Code available at https://bitbucket.org/malb/dgs/src/master/, commit ID `4c5531d`.

Algorithm 3.2 Gentleman-Sande Inverse NTT

Input: A polynomial, $\hat{\mathbf{x}}$ in the NTT-domain, and $\omega^{-1} = \{\omega_0^{-1}, \omega_1^{-1}, \ldots, \omega_n^{-1}\}$ — inverse powers of $2n$th roots of unity, stored in bit-reversed order.

Output: $\mathbf{x} = \hat{\mathbf{x}} \leftarrow INTT(\hat{\mathbf{x}})$

```
t ← 1
m = n/2
while m > 0 do
    k ← 0
    for i ← 0; i < m; i ← i + 1 do
        S ← ω^{-1}[m + i]
        for j ← k; j < k + t; j ← j + 1 do
            U ← x̂[j]
            V ← x̂[j + t]
            x̂[j] ← U + V
            x̂[j + t] ← (U − V) · S
        end for
        k ← k + 2t
    end for
    t ← 2t
    m ← m/2
end while
for i ← 0; i < n; i ← i + 1 do
    x̂[i] ← x̂ · n^{-1}
end for
return x̂
```

as our instantiation proposal. To circumvent this issue, we use the composability of normal distributions. We know that if $X \sim \mathcal{N}(0, \rho_0^2)$ and $Y \sim \mathcal{N}(0, \rho_1^2)$, then $X + Y \sim \mathcal{N}(0, \sqrt{\rho_0^2 + \rho_1^2})$ — note that $Z \sim \mathcal{N}(0, \alpha^2)$ means that Z is a random variable from a centered Gaussian distribution with parameter α. This way, we are able to sample for $\sigma_2 = 94371960$ by using smaller distributions, sampling uniquely and independently, and summing up the result — for each coefficient. As this is done by a single party once during the entire protocol, it does not impact performance to a large degree, though it is slower than an implementation supporting larger numbers could be.

We can verify the correctness of the discrete Gaussian distribution sampler using SAGA [HPRR19][4]. This is a software suite designed for testing the output of samplers of discrete Gaussian distributions against what we expect. Given a statistically significant sample, SAGA is able to determine whether the given sample could come from a given discrete Gaussian with overwhelming probability. Testing 100000 polynomials for statistical correctness in sampling using σ_1 is shown in Figure 3.2.

[4]Code is available at https://github.com/PQShield/SAGA, commit ID `bc3341d`.

```
Testing a Gaussian sampler with center = 0 and sigma = 2.0
Number of samples: 102400000

Moments   |   Expected     Empiric
----------+----------------------
Mean:     |   0.00000      0.00005
St. dev.  |   2.00000      1.99999
Skewness  |   0.00000      -0.00021
Kurtosis  |   0.00000      0.00051

Chi-2 statistic:   12.429527956187837
Chi-2 p-value:     0.9004992020583328   (should be > 0.001)

How many outliers? 0

Is the sample valid? True
```

Figure 3.2: The output of testing our sampler with SAGA [HPRR19]. The empiric data is close to what we expect, illustrated by the χ^2 statistic and p-value being above the critical value. This figure is the output of running SAGA on polynomials generated using DGS [AW18].

3.2.3 Reconciliation Function

We will use Peikert's reconciliation function [Pei14], which we previously discussed in Section 2.4.4. We believe that this is a suitable reconciliation protocol due to its efficiency and negligible security effect. Furthermore, it is easy to implement in software with high performance, making it even more desireable.

3.3 Estimating the Computational Security

When instantiating a protocol, it is useful to estimate the computational security it provides. How secure is the proposal given above?

Apon *et al.* provide the maximum advantage given to an attacker as:

$$\mathrm{Adv}^{GKE}(t, q) \leq 2^{-\lambda+1} +$$

$$\sqrt{\left(N \cdot \mathrm{Adv}^{RLWE}_{n,q,\chi_{\sigma_1},3}(t_1) + \mathrm{Adv}_{KeyRec}(t_1) + \frac{q}{2^\lambda}\right) \cdot \frac{\exp\{2\pi n(\beta_{R\acute{e}nyi}/\sigma_2)^2\}}{1 - 2^{-\lambda+1}}}$$

where $t_1 = t + \mathcal{O}(N \cdot t_{ring})$ and $\mathcal{O}(N \cdot t_{ring})$ denotes the computational complexity of N group operations in the ring R_q.

The attack surface of Apon and Choi primarily consists of:

1. The underlying pure RLWE and derivative LWE problems,
2. the key reconciliation protocol used, and
3. through distinguishing based on the approximation to a uniform distribution using a large standard deviation.

As we are provided with the advantage given to an attacker through the key reconciliation protocol, and we are able to directly calculate the advantage given by the use of a Gaussian with a high standard deviation, we only miss the advantage given by the underlying RLWE scheme.

As our chosen σ_2 is about $30\beta_{Rényi}$ — $\Omega(\beta_{Rényi})$ — $\frac{\exp\{2\pi n(\beta_{Rényi}/\sigma_2)^2\}}{1-2^{-\lambda+1}}$ reduces to $\exp\{2\pi n(1/30)^2\}$, as $1-2^{-\lambda+1} \approx 1$. Calculating $\frac{q}{2^\lambda} = \frac{45510033409}{2^{256}} \approx \frac{2^{35}}{2^{256}} = 2^{-221}$. Furthermore, the advantage given by Peikert's key reconciliation function is negligible [Pei14], and we set $\mathrm{Adv}_{KeyRec}(t_1) = 0$.

There are multiple methods to attack RLWE-based cryptosystems, the most relevant of which are through reductions to the general LWE problem or to ideal lattices, all of which are explained in detail by Player [Pla18]. The most relevant attack against AponGKE and ChoiGKE are the unique shortest-vector problem (uSVP), the bounded distance decoding problem (BDD) [LM09] and dual lattice problem [APS15]. Using the tool, *LWEestimator*, created by Albrecht *et al.* [APS15] we can estimate the advantage an adversary has in the general RLWE problem with our parameters — $\mathrm{Adv}^{RLWE}_{n,q,\chi_{\sigma_1},3}(t_1) = 2^{-2034.4}$, by uSVP. This allows for us to estimate the security of the protocol instance as a whole:

$$\begin{aligned} \mathrm{Adv}^{GKE}_{Apon}(t,q) \leq 2^{-255} + \\ \sqrt{\left(N \cdot 2^{-2034.4} + 1 + \frac{2^{35}}{2^{256}}\right) \cdot \exp\{2048\pi(1/30)^2\}} \\ = 2^{-255} + 2^{-105.14} = 2^{-105.14}. \end{aligned} \tag{3.5}$$

Where we assume that N is a reasonable number of participants. Note that in the above equation, the value of N is irrelevant for any reasonable number of participants as $2^{-2034} \approx 5.06 \cdot 10^{-613}$. We thus achieve about 105 bits of security for AponGKE and ChoiGKE — two post-quantum group key exchange protocols. Comparing this security level with many proposed two-party key exchange and key agreement protocols estimated by Albrecht *et al.* [ACD+18], indicates that our claimed security level is reasonable, especially when taking into consideration the fact that these parameters support up to 20 simultaneous participants.

Chapter 4
Protocol Implementation

In this chapter, we will look at implementations of instantiations provided in the previous chapter for post-quantum group key exchange. Furthermore, we will see these in an authenticated setting, resulting from the Katz-Yung compiler, and the framework of Bresson *et al.* respectively, as we have previously seen in Section 2.5. Then, we will show a real-world-like messaging application using one of the two implemented GAKE. These results correspond to the three design science cycles performed during this thesis. Finally, we will take a closer look at the performance of some fundamental building blocks of the group key exchange.

Chapter 5 will focus on discussing the results presented here and in the previous chapter.

4.1 Data Collection

Data gathered in this section come from tests run on a server with the following processor: Intel(R) Core(TM) i7-7700K CPU with a clock speed of 4.20GHz. On average, the processor completes 4200328733 clock cycles in a second.

We compile all the test programs using the GNU Compiler Collection (GCC)[1] version 9.3.0 with the following flags:
`-O3 -fomit-frame-pointer -msse2avx -march=corei7-avx`. We use CMake[2] to build and compile our programs, using the template `CMakeLists.txt` file printed in Appendix A.

In this chapter, we will compare the performance of both authenticated and unauthenticated AponGKE and ChoiGKE. To give the reader a notion of what the numbers mean, we will compare these to the current "state-of-the-art" ECDH key exchange over Curve25519 [Ber06], using X25519 according to RFC7748 [LMT16].

[1]GNU C Compiler, available at https://gcc.gnu.org/
[2]Website: https://cmake.org

Specifically, we will perform key exchange for groups in a similar way as it is done in the current Signal Protocol [Sig][CGCD+17] — pairwise between all communicating parties [CGCG+17]. However, instead of the triple Diffie-Hellman key exchange, we will use a singular one, meaning each participant performs a single ECDH key exchange with every other participant. For the GAKE version of this, we reuse the Curve25519 private key for signing.

Implementing ECDH is out of scope for this thesis, so we will use the implementation provided by the efficient and small cryptography library *Monocypher* [Vai20], specifically version 3.0.0. This is a minimal library written in pure C, with a reasonable amount of optimization. We believe this will be suitable as a baseline comparison, given that the software developed in this thesis is optimized to a similar degree.

Computational Performance We measure the computational performance through counting CPU cycles over 1000 iterations of the protocol for each number of participants. The x86-64 processor we use support the built in (x86) Assembly instruction to do this, which we call from our C-code function `count_cycles()` as seen in Listing 4.1. The measurements are performed by calling this function before and after the operation to be measured, with the difference being the number of CPU cycles used between the calls, and thus the cycle count for the operation in question. We use CPU cycles as a measure of performance, as this is reproducible across most newer generations of x86 processors.

```
1  long long count_cycles() {
2    unsigned long long result;
3    asm volatile(
4      "rdtsc;"
5      "shlq $32, %%rdx;"
6      "orq %%rdx, %%rax"
7      : "=a" (result)
8      :
9      : "%rdx");
10
11   return result;
12 }
```

Listing 4.1: This function is called right before and directly after the code we wish to measure the performance of. The difference between the two values obtained are the number of CPU cycles required for the operation.

Memory Usage We are able to analytically determine the amount of memory required by any given protocol run. This will not give an exact number, as compiling on different platforms yield different machine code; however, it does yield a good approximation of reality.

Network Usage Finally, in Section 4.5 of this chapter, we will analyze the network usage required to set up a key exchange in a real-world scenario. We do this by inspecting a protocol run in a real-world-like scenario using tcpdump[3] and Wireshark[4].

4.2 Performance of Subroutines

The performance of the group key exchange protocols as a whole is dependent on the performance of many crucial subroutines. Before looking at the group key exchange protocols as a whole, we will see the performance of these crucial subroutines for generating polynomials (uniform, with σ_1, and with σ_2), the forwards and inverse NTT transformations, and the key reconciliation functions. The numbers in Table 4.1 are rounded to the nearest integer, based on data from 1 000 iterations.

Subroutine	Mean CPU cycles	Standard deviation
`GenUniformPoly()`	149 372	2 366
`GenPoly(`σ_1`)`	307 745	30 088
`GenPoly(`σ_2`)`	6 012 056	262 746
`NTT()`	98 407	13 541
`InverseNTT()`	117 253	19 783
`PolyConvolution()`	327 635	23 691
`NaivePolyConvolution()`	17 389 609	953 746
`RecMsg()`	67 753	6 664
`RecKey()`	23 152	4 163

Table 4.1: The performance for a selection of subroutines that are common to AponGKE and ChoiGKE. Note that `GenPoly(`σ_2`)` which is only used once in a protocol run, by a single participant, and is included here for the sake of completeness. As a comparison, the performance of the naive approach to polynomial convolution is included.

Below we provide a brief explanation of what the different subroutines do.

- `GenUniformPoly()` samples a uniform ring element by reading raw bytes from `/dev/urandom` — the operating system CSPRNG — and placing them into the field, $\mathbb{Z}_q$,
- `GenPoly(`σ_1`)` and `GenPoly(`σ_2`)` samples ring elements from a discrete Gaussian distribution, as described in Section 3.2.2,

[3]Website: https://www.tcpdump.org
[4]Website: https://www.wireshark.org

- `NTT()` and `InverseNTT()` are the forwards and inverse NTT transforms used to perform polynomial convolution,
- `PolyConvolution()` is the function taking two polynomials to be convolved and returning the result, using the NTT-functions to perform the convolution,
- `NaivePolyConvolution()` is included here for illustration purposes only, and performs polynomial convolution the direct way, and
- `RecMsg()` and `RecKey()` are directly corresponding to the functions described in Section 2.4.4 about Peikert reconciliation.

4.3 Group Key Exchange Implementations

In this section, we will see the implementation of unauthenticated AponGKE and ChoiGKE, as well as the results provided by these. We have implemented these in C, and the code is publicly available at https://github.com/simenbkr/rlwe-gke, using commit ID `1795995`. Due to optimization and practicality, the explanations below does not necessarily have a one-to-one correspondence to the actual code.

4.3.1 AponGKE and ChoiGKE

The implementations for AponGKE and ChoiGKE are similar in many ways. The major difference is the key exchange itself, while the way we represent polynomials, and perform ring operations are identical. As q was previously selected as a 35-bit prime, a polynomial is $n = 1024$ coefficients and we choose to represent a coefficient as a 64-bit unsigned integer, a polynomial is represented in our implementation as a list of 1024 64-bit integers.

State Variables

A participant, P_i, in AponGKE and ChoiGKE, has a state variable containing his own polynomials:

- The secret polynomial, $\mathbf{s}_i$,
- the errors $\mathbf{e}_i$, $\mathbf{e}'_i$, (optionally $\mathbf{e}''_i$),
- the public key, $\mathbf{z}_i = \mathbf{s}_i\mathbf{a} + \mathbf{e}_i$,
- the "neighbourhood"-key, $\mathbf{X}_i = (\mathbf{p}_{i+1} - \mathbf{p}_{i-1})\mathbf{s}_i + \mathbf{e}'_i$, and
- the approximate shared secret, $\mathbf{b}_i$.

Furthermore, each participant has to store the broadcasted polynomials associated with the other participants. Let P_i^j denote the state variable for P_i containing the broadcasted keys from P_j and holding the following:

- P_j's public key — $\mathbf{p}_j$ if P_j is a neighbour of $P_i \iff j = i + 1 \mod N$ or $j = i - 1 \mod N$,
- P_j's "neighbourhood"-key $\mathbf{X}_j = (\mathbf{p}_{j+1} - \mathbf{p}_{j-1})\mathbf{s}_j + \mathbf{e}'_j$

Finally, the reconciliation vector can be represented as a 128-byte value (in practice a list of 32 8-byte values) directly corresponding to 1024 bits, one for each coefficient, using Peikert reconciliation, as we previously saw in Section 2.4.4.

P_i in ChoiGKE has an additional state variable, $\mathbf{Y}_i$, which holds N polynomials. This is a set of intertwined "neighbourhood"-keys, which we previously reviewed in Section 2.4.3.

Subroutines

We define a set of subroutines for the group key exchange protocols. These act using the state variable P_i as input. These are the common functions:

- `InitializeParticipant`(P_i) — Generates the secret and error polynomials, and calculates P_i's public key, which is subsequently broadcasted to the other participants.
- `ComputeX`(P_i) uses the receieved public keys of P_i's neighbours to compute the neighbourhood-key, $\mathbf{X} = (\mathbf{p}_{i+1} - \mathbf{p}_{i-1}\mathbf{s}_i + \mathbf{e}'_i)$, and broadcasts it to the other participants.
- `RecKey`(P_i, rec) and `RecMsg`(P_i) — These are the key reconciliation functions previously reviewed in Section 2.3.2, instantiated as Peikert's key reconciliation function, as seen in Section 2.4.4. We use these to go from $\mathbf{b}_i$ to a definite shared value k_i, which only works when $\mathbf{b}_0 \approx \mathbf{b}_1 \cdots \approx \mathbf{b}_{N-1}$.

In addition to these, AponGKE has the following subroutine:

- `ComputeApproxSecret`(P_i) calculates the approximate shared secret value, $\mathbf{b}_i = N \cdot \mathbf{p}_{i-1}\mathbf{s}_i + \sum_{j=0}^{N-1}(N - j - 1)\mathbf{X}_{i+j \mod N}$

While ChoiGKE has the following additional subroutines:

– `ComputeYList(`P_i`)` — This function creates the list of "neighbourhood"-keys intertwined with our secret. We start with $\mathbf{Y}_{i,i} = \mathbf{X}_i + \mathbf{p}_i \mathbf{s}_i$, and then iteratively compute $\mathbf{Y}_{i,i+j} = \mathbf{X}_{i+j} + \mathbf{Y}_{i,i-1+j}$.

– `ComputeApproxSecret(`P_i`)` — Using $\mathbf{Y}_i$, we derive the approximate shared secret — $b_i = \sum_{j=0}^{n} \mathbf{Y}_{i,i+j}$.

Algorithmic Illustration

Putting the state variables and subroutines together, we can create an algorithmic version of both AponGKE and ChoiGKE from the perspective of P_i, as illustrated by Algorithm 4.1. Let $P_i^{j,\mathbf{p}_j}$ denote the storage of P_j's public key, $\mathbf{p}_j$ in P_i's state variable, and likewise for other variables.

Algorithm 4.1 AponGKE

Input: N, n, q, λ, ρ, χ_{σ_1}, χ_{σ_2}.
Output: Shared secret, *session_key*, between N parties.

`InitializeParticipant(`P_i`)`
$P_i^{j,\mathbf{p}_j} \leftarrow \mathbf{p}_j$, $j \in \{i-1, i+1\}$
`ComputeX(`P_i`)`
$P_i^{j,\mathbf{X}_j} \leftarrow \mathbf{X}_j$, $\forall j \in \{0, \ldots, N-1\}$ and $j \neq i$.
`ComputeApproxSecret(`P_i`)`
if $i = N-1$ **then**
    $rec \leftarrow$ `recMsg(`P_i`)`
    $k_i \leftarrow$ `recKey(`P_i`,` rec`)`
else
    $k_i \leftarrow$ `recKey(`P_i`,` rec`)`
end if
$session_key \leftarrow \mathcal{H}(k_i)$
return $session_key$

4.3.2 Performance

The performance of the group key exchange protocols is divided into the analytical analysis of memory requirements and an experimental part showing the artifacts' performance, which should give an accurate indication as to how a more refined implementation could perform.

Memory Consumption

From an analytical point of view, we can count the number of polynomials needed by each protocol participant in the above explanation of state variables.

– ApponGKE requires seven polynomials to represent the local participant state. Furthermore, it requires two additional ones for each additional participant. Ergo, $7 + 2N$ polynomials. As each polynomial is 1024 64-bit values, each participant needs $5.73 + N \cdot 1.64$kB of memory.

– ChoiGKE requires the same seven, plus N polynomials, to represent the local participant state. It also requires two additional polynomials for each additional participant. This results in needing $7 + 3N$ polynomials, which results in $5.73 + N \cdot 2.46$kB of memory.

Protocol (participant context)	N = 3	N = 10	N = 20
AponGKE ($i \neq N-1$)	123 008	213 120	376 960
AponGKE ($i = N-1$)	106 624	221 312	385 152
ChoiGKE ($i \neq N-1$)	131 200	303 232	548 992
ChoiGKE ($i = N-1$)	123 008	295 040	540 800

Table 4.2: The memory usage in bytes for 3, 10, and 20 participants for both roles (initiator and responder) of AponGKE and ChoiGKE.

We summarize the dynamic memory usage of AponGKE and ChoiGKE in Table 4.2. In addition to this comes the static memory of the compiled binary, which, when statically compiled, is on the order of a megabyte.

Computational Performance

Figure 4.1 shows a comparison of the CPU-cycles for both of the GKE implementations, compared with the baseline explained at the start of this chapter. The arrow points to the mean cycle time, with the horizontal caps indicating the standard deviation of the observed CPU cycle times. The raw data presented by Figure 4.1 is available in Appendix B.

Error Rate

Both AponGKE and ChoiGKE have a non-negligible error rate for the selected parameters when the number of participants reaches above 10. This is due to the noise level of this many participant's public keys being mixed, making the reconciliation process at times impossible. The empiric error rate seems to be exponential as more participants are added, as shown in Table 4.3.

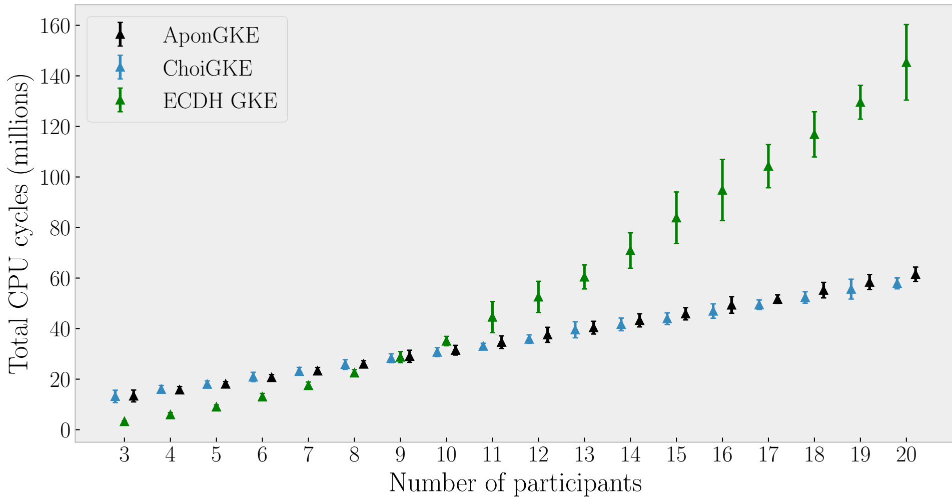


Figure 4.1: A comparison of the CPU cycles required to perform an (unauthenticated) GKE for AponGKE, ChoiGKE and pairwise ECDH using Curve25519. The comparisons are done on the server described above, using the same compilation options found in Appendix A.

4.4 GAKE Implementation

In this section, we provide the specifics for turning AponGKE and ChoiGKE into authenticated group key exchange protocols. We will refer to these as AponGAKE and ChoiGAKE.

4.4.1 Choice of Digital Signature Scheme

In Section 2.5, we discussed different compilers for turning a GKE into a GAKE. Both of these require a digital signature scheme. Also, in this thesis, we are specifically looking at RLWE-schemes for group key exchange. In the context of this thesis, it makes sense to also use a digital signature algorithm based on the same, or related, problem. Three candidates from the NIST standardization process[5] — which we discussed briefly in Chapter 1 — are based on lattice cryptography, namely FALCON (NTRU-based), qTESLA (RLWE-based), and CRYSTALS-Dilithium (Module-RLWE). In this thesis, we use FALCON and qTESLA showing how they can be used, and how they perform in the group authenticated key exchange. We use the lowest security levels these provide, which is close to the estimated security level of the instantiated protocols treated here. For FALCON we then use FALCON-512, and with qTESLA we use qTESLA-p-I.

[5]Website: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

Participants	AponGKE Error rate	ChoiGKE Error rate
3	0.0000	0.0000
4	0.0000	0.0000
5	0.0000	0.0000
6	0.0000	0.0000
7	0.0000	0.0000
8	0.0000	0.0000
9	0.0000	0.0000
10	0.0000	0.0000
11	0.0000	0.0001
12	0.0006	0.0002
13	0.0015	0.0014
14	0.0036	0.0055
15	0.0116	0.0121
16	0.0288	0.0254
17	0.0448	0.0479
18	0.0820	0.0808
19	0.0950	0.1336
20	0.1994	0.2049

Table 4.3: Here we see the error rate given by AponGKE and ChoiGKE for a number of participants. We will discuss this figure more in the next chapter.

As implementing digital signature schemes is deemed out of scope for this thesis, we use the implementation provided by the OQS[6] project.

4.4.2 Protecting Against Active Attacks

As before, we assume the ring, a public polynomial, $\mathbf{a} \in R_q = \mathbb{Z}_q[x]/\langle x^n + 1\rangle$ and the distributions χ_{σ_1} and χ_{σ_2} to be known. Furthermore, we assume that the participants have agreed on a digital signature scheme to use, with known algorithms and constants for key generation, $\mathcal{K}$, signature generation, $\mathcal{S}$, and signature verification, $\mathcal{V}$. We let $a \| b$ denote the concatenation of a and b. Finally, recall that $\mathcal{H}$ denotes a cryptographic hash function, as described in Section 2.2.4.

[6]Website available at https://openquantumsafe.org, and code at https://github.com/open-quantum-safe/liboqs. We use commit `38c47f7`.

AponGAKE

We recall the form AponGKE, outlined in Section 2.4.2. Using the Katz-Yung compiler, as suggested by Apon [ADSGK19], yields a protocol with the following rounds:

1. All participants, P_i, generate a digital signature key pair, $(vk_i, sk_i) \leftarrow \mathcal{K}$, a nonce selected at (uniformly) random from $n_i \leftarrow \{0,1\}^\lambda$ (λ being Kleene notation). P_i then broadcasts $i\|0\|n_i$. This is stored in a state for every participant.

2. P_i samples $\mathbf{s}_i \xleftarrow{\chi_{\sigma_1}} R_q$, $\mathbf{e}_i \xleftarrow{\chi_{\sigma_1}} R_q$ and calculates $\mathbf{z}_i = \mathbf{s}_i\mathbf{a} + e_i$. P_i broadcasts the message $i\|1\|\mathbf{z}_i\|\rho_{i,1}$, where $\rho_{i,1} \leftarrow \mathcal{S}(sk_i, 1\|\mathbf{z}_i\|n_i)$.

3. $P_i, i \neq 0$ samples $\mathbf{e}'_i \xleftarrow{\chi_{\sigma_1}} R_q$ and P_0 samples $\mathbf{e}'_0 \xleftarrow{\chi_{\sigma_2}} R_q$. P_i computes $\mathbf{X}_i \leftarrow (\mathbf{z}_{i+1} - \mathbf{z}_{i-1})\mathbf{s}_i + \mathbf{e}'_i$. Creates a signature, $\rho_{i,2} \leftarrow \mathcal{S}(sk_i, 2\|\mathbf{X}_i\|n_i)$ and broadcasts $i\|\mathbf{X}_i\|\rho_{i,2}$.

4. P_i sets $\mathbf{b}_i \leftarrow \mathbf{z}_{i-1}N\mathbf{s}_i + (N-1)\mathbf{X}_i + (N-2)\mathbf{X}_{i+1} + \ldots + \mathbf{X}_{i+N-2}$. Participant P_{N-1} samples $\mathbf{e}''_{N-1} \xleftarrow{\chi_{\sigma_1}} R_q$ and sets $\mathbf{b}_{N-1} \leftarrow \mathbf{b}_{N-1} + \mathbf{e}''_{N-1}$. P_{N-1} creates a reconciliation vector and recoveres the session key $(\mathbf{rec}, k_{N-1}) \leftarrow recMsg(\mathbf{b}_{N-1})$. Then P_{N-1} sets $\rho_{N-1,3} \leftarrow \mathcal{K}(sk_{N-1}, 3\|\mathbf{rec}\|n_i)$ and broadcasts $N-1\|\mathbf{rec}\|\rho_{N-1,3}$.

5. All participants recover the key as $k_i \leftarrow recKey(b_i, rec)$.

ChoiGAKE

Choi *et al.* demonstrates compiling ChoiGKE into ChoiGAKE in their paper, [CHK20]. We will repeat a version of that GAKE here. Recall the description of ChoiGKE in Section 2.4.3.

1. All participants, P_i, generate a digital signature pair $(vk_i,\ sk_i) \leftarrow \mathcal{K}$ and a nonce selected uniformly from $\{0,1\}^\lambda$. They then generate a secret key and an errors from the ring — $\mathbf{s}_i \xleftarrow{\chi_{\sigma_1}} R_q$, $\mathbf{e}_i \xleftarrow{\chi_{\sigma_1}} R_q$ and calculates$\mathbf{z}_i = \mathbf{s}_i\mathbf{a} + e_i$. Then the message $m_{i,1}$ is constructed as $m_{i,1} \leftarrow i\|1\|\mathbf{z}_i$, and signed, $\rho_{i,1} \leftarrow \mathcal{S}(sk_i, m_{i,1})$. $m_{i,1}\|\rho_{i,1}$ gets broadcasted to its neighbours.

2. Each received message is verified using the corresponding verfication key. If $i = 0$, then $\mathbf{e}'_i \xleftarrow{\chi_{\sigma_2}} R_q$, if not then $\mathbf{e}'_i \xleftarrow{\chi_{\sigma_1}} R_q$. P_i calculates their "neighbourhood"-key — $\mathbf{X}_i = (\mathbf{z}_{i+1} - \mathbf{z}_{i-1})\mathbf{s}_i + \mathbf{e}'_i$. P_i sets $m_{i,2} \leftarrow i\|2\|\mathbf{X}_i\|n_i$, which is signed, $\rho_{i,2} \leftarrow \mathcal{S}(m_{i,2})$, and $m_{i,2}\|\rho_{i,2}$ is broadcasted to all participants.

3. All messages with the "neighbourhood"-keys are verified, and the message nonces are extracted and added to the state variables of the participants. Then, P_{N-1} samples $\mathbf{e}''_i \xleftarrow{\chi_{\sigma_1}} R_q$, and computes

$$\mathbf{Y}_{N-1,N-1} = \mathbf{X}_{N-1} + \mathbf{p}_{N-2}\mathbf{s}_{N-1} + \mathbf{e}''{}_{N-1}.$$

Then, P_{N-1} computes

$$\mathbf{Y}_{N-1,N-1+j} = \mathbf{X}_{N-1+j} + \mathbf{Y}_{N-1,N-2+j}.$$

for $\forall j \in \{1, \ldots, N-1\}$. Finally, P_{N-1}'s approximate secret, b_{N-1}, is calculated as

$$b_{N-1} = \sum_{j=0}^{N-1} \mathbf{Y}_{N-1,N-1+j}$$

Now, P_{N-1} generates the reconciliation vector, **rec**, and the key k_{N-1} from $\mathbf{rec}, k_{N-1} = recMsg(\mathbf{b}_{N-1})$. The reconciliation message is constructed as $m_{N-1,3} \leftarrow N-1\|3\|\mathbf{rec}$. Then P_{N-1} broadcasts $m_{N-1,3}\|\rho_{N-1,3} \leftarrow \mathcal{S}(m_{N-1,3})$.

4. The reconciliation message is verified by all $P_i, i \neq N-1$. All other participants P_i calculate

$$\mathbf{Y}_{i,i} = \mathbf{X}_i + \mathbf{p}_{i-1}\mathbf{s}_i.$$

Then they compute

$$\mathbf{Y}_{i,i+j} = \mathbf{X}_{i+j} + \mathbf{Y}_{i,i-1+j}$$

for $\forall j \in \{1, \ldots, N-1\}$. The approximate polynomial is calculated as

$$\mathbf{b}_i = \sum_{j=0}^{N-1} \mathbf{Y}_{i,i+j}.$$

Finally, the shared secret is recovered using an error reconciliation function — $k_i = recKey(\mathbf{b}_i, \mathbf{rec})$.

5. Then the session key is set to $sk_i \leftarrow \mathcal{H}(k_i)$.

4.4.3 Implementation Modifications

In order to accommodate the changes required by the different compilers applied to AponGKE and ChoiGKE, we need some modifications regarding the state variables and subroutines used by the authenticated group key exchange protocols. We also introduce a new concept, *messages*, which will provide further abstractions and simplifications.

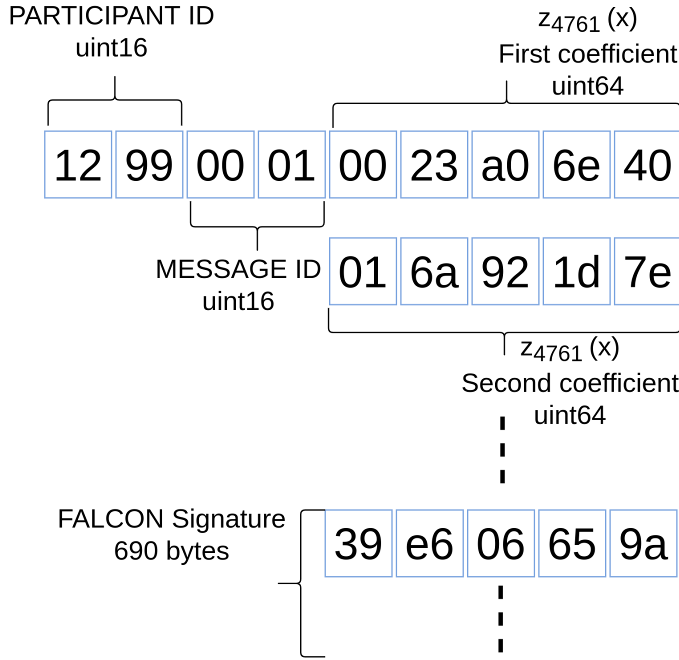


Figure 4.2: An illustration of the encoding schema used to represent messages in the GAKE implementation. Here we use FALCON as an example, though it is easily exchangeable.

Messages

The description of AponGAKE and ChoiGAKE uses messages, instead of just sending polynomials and vectors. The messages are a concatenation of the polynomial or vector, and associated metadata — i.e., nonces and identifiers. We apply this approach in the implementation as well. Both polynomials and relevant identification numbering of messages and participants are represented as a collection of byte values (8-bit unsigned integers) in a structured element. This makes integration with OpenQuantumSafe's digital signature algorithms relatively trivial, due to the similar way to represent raw data. The encoding schema is illustrated in Figure 4.2. Here we see an example of encoding the public key broadcasting message from P_{4761} (note that $4761 = 1299$ in hexadecimal). We note that even though the numbers are treated as 64-bit integers when performing operations on them, due to the modulus being 35-bit, we can encode them as 40-bit integers. This reduces the memory usage by close to 40%, which will become even more critical when we move to networked usage of AponGAKE and ChoiGAKE.

State Variables

To accommodate the transformation to authenticated group key exchange, we need to add a keypair of verification and signature keys to each participant's internal state, as well as $N-1$ verification keys to each participants protocol state, one for each other participant. Additionally, each participant needs a nonce for their internal state, and the state representing the protocol as a whole. The protocol dictates that the nonce should be selected uniformly at random from $\{0,1\}^\lambda$, which corresponds directly to an unsigned 32-bit integer.

Other than these small changes, the state variables remain the same. The "paradigm" has changed somewhat though, as we no longer deal directly with polynomials, but with messages wrapped around them.

Subroutines

The additional subroutines we need are related to the messages and the authentication of these, and are the same for both AponGAKE and ChoiGAKE, though applied in different manners due to the inherent differences between the protocols. The subroutines correspond to the following:

- `CreateMessageForRound(`P_i`, `k`)` — Creates a message object for the specific round, depending on the participant's internal state, signing the message in the process.
- `VerifyMessage(`P_i`, `$m_{j,k}$`)` — Takes the internal state of the verifying participant, P_i and a message for round k received from P_j, and outputs either 1 if verified successfully or 0 if the verification failed.
- `ExtractPoly(`m`)` extracts a polynomial from a (signed) message.
- `AbortGKE()` aborts the protocol run, tearing down any connections and deleting the secret data from memory.

Furthermore, `InitializeParticipant(`P_i`)` now also generates a digital signature key pair, and a nonce, which is broadcast upon generation to the other participants.

Algorithnic Illustration

In Algorithm 4.2, we provide a logical illustration of the group authenticated key exchange, AponGAKE.

Algorithm 4.2 AponGAKE

Input: N, n, q, λ, ρ, χ_{σ_1}, χ_{σ_2}.
Output: Shared secret between N parties, or NIL, indicating an error.

```
InitializeParticipant(P_i)
P_i^{j,nonce} ← nonce_j, ∀j ∈ {0,...,N−1} and j ≠ i        ▷ Get the nonce and
P_i^{j,vk} ← vk_j                 ▷ verification key from all the other participants.
CreateMessageForRound(P_i, 1)      ▷ The sending of this message to the
neighbours of participant i is left implied.
if VerifyMessage(m_{j,1}), j ∈ {i−1, i+1} then
    P_i^{j,p_j} ← p_j, j ∈ {i−1, i+1}
else
    AbortGKE()
    return NIL
end if
ComputeX(P_i)
CreateMessageForRound(P_i, 2)      ▷ The broadcast of this message to all other
participants is left implied.
if VerifyMessage(m_{j,2}) ∀j ∈ {0,...,N−1} and j ≠ i then
    P_i^{j,X_j} ← X_j = ExtractPoly(m_{j,2}), ∀j ∈ {0,...,N−1} and j ≠ i.
else
    AbortGKE()
    return NIL
end if
ComputeApproxSecret(P_i)
if i = N − 1 then
    rec ← recMsg(P_i)
    k_i ← recKey(P_i, rec)
else
    k_i ← recKey(P_i, rec)
end if
session_key ← H(k_i)
return session_key
```

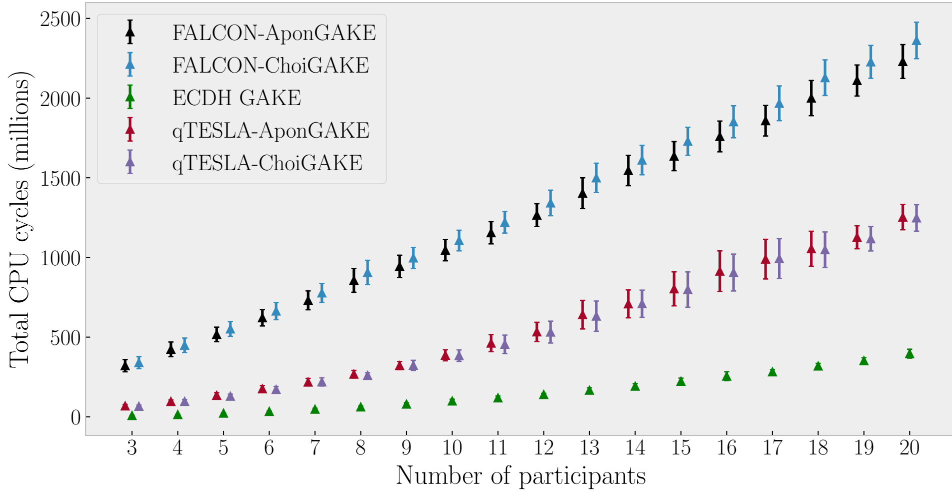


Figure 4.3: The average number of CPU cycles in millions to perform a GAKE using AponGAKE and ChoiGAKE with digital signature schemes FALCON and qTESLA compared with a naive ECDH-GAKE.

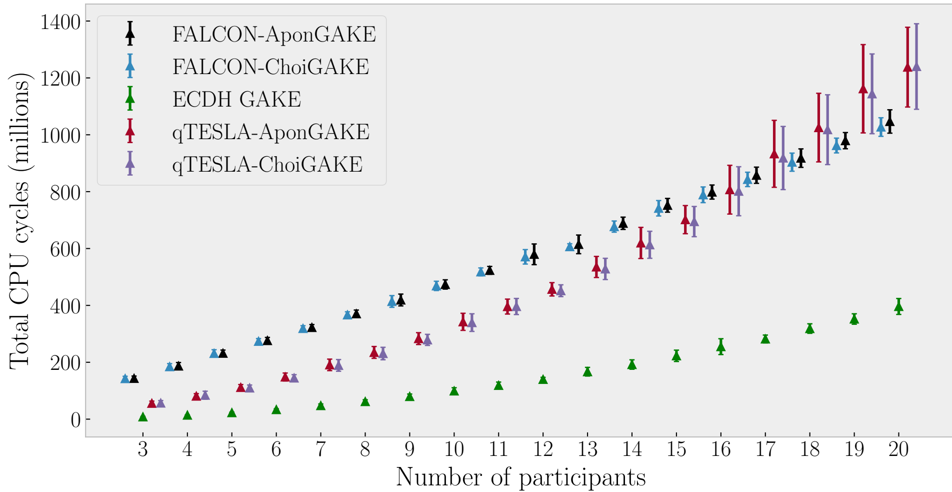


Figure 4.4: A comparison of the performance when discounting the generation of the digital signature scheme key pair.

4.4.4 Performance

Memory Consumption

For FALCON, the digital signature keypair adds an overhead of 897 bytes per verification key, 1 281 bytes per signing key, and 690 bytes per signature. For qTESLA, we need 14 880 bytes for the verification key, 5 224 for the secret key, and 2 592 for each signature. Additionally, the messaging paradigm adds an overhead of 6 bytes per message.

Processor Usage

For both AponGAKE and ChoiGAKE, the CPU usage of the group authenticated key exchange protocols are dominated by the signing and verification algorithms of FALCON, as may be surmised comparing Figures 4.1 and 4.3. As the key generation part of FALCON is the dominating factor, we also present a comparison of cycles without this subroutine included in Figure 4.4. Now the dominating subroutines are the verification and signing routines.

4.5 Real-world Interactive Messaging

Putting it all together in a real-world like scenario is useful in evaluating the feasibility of the protocols as a whole. In this section, we will see an implementation of a simple messaging application built upon the protocols described in the previous sections. Specifically, the implementation will use the FALCON version of AponGAKE.

4.5.1 Practical Considerations

We make some practical considerations concerning the implementation.

- Move all sampling of polynomials to the same point in time — P_i samples all of $\mathbf{s}_i$, $\mathbf{e}_i$, $\mathbf{e}'_i$, and possibly $\mathbf{e}''_i$ in succession at the start of the protocol run, before initiating any network connection. This ensures a minimal amount of delay, as the major factor in the time required to set up the connection is network throughput.

- Although ChoiGKE supports the dynamic joining and leaving of groups, we have not included this in our practical implementation, though it will be included in the evaluation as a whole.

- In our runs of the protocols, we did not generate the public polynomial a on the fly, but instead included it statically in all the clients.

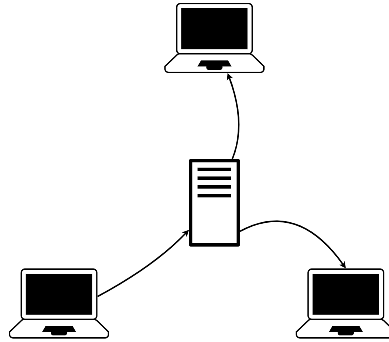


Figure 4.5: A minimal example of the architecture used in the simple interactive messaging application.

4.5.2 System Architecture

As a peer-to-peer setup is complicated, we have chosen to use a simple client-server architecture, with the client being a C implementation performing all the cryptographic operations, and the server, in Python, acting as a hub with each client publishing to it, and the server forwarding the messages to the other, relevant, clients.

Figure 4.5 provides an illustration of this architecture, showing a participant broadcasting, for example, their public key to its neighbours. In our specific implementation we use this client-server architecture, but in general it is not required. However, we note that peer-to-peer applications can typically be more complex, which for our direct purposes is irrelevant. Additionally, many communications systems over networks and between multiple parties use a similar approach, notable examples being Signal [Sig], Wire [Wir], and Zoom [Zoo16]. These examples are applications where post-quantum group key exchange may be useful, thus making our prototype system architecture even more relevant.

4.5.3 Messages

There are essentially three message types:

1. Polynomial message, consisting of the sending participant identifier, the sequence number, the signature, and the polynomial itself.
2. Reconciliation message, consisting of the sending participant identifier, the sequence number, the signature, and the reconciliation vector itself.
3. A chat message, consisting of sending participant identifier, sender's distinguisher, and the encrypted message. We encrypt the entire message using an AEAD scheme.

Source	Destination	Protocol	Length	Info
51.15.124.10	192.168.1.107	TCP	1514	4000 → 36334 [ACK] Seq=19304 Ack=12671 Win=54016 Len=1448 TSval=128824419 TSecr=2162480510
51.15.124.10	192.168.1.107	TCP	1514	4000 → 36334 [ACK] Seq=20752 Ack=12671 Win=54016 Len=1448 TSval=128824419 TSecr=2162480510
192.168.1.107	51.15.124.10	TCP	66	36334 → 4000 [ACK] Seq=12671 Ack=22200 Win=64128 Len=0 TSval=2162480547 TSecr=128824419
51.15.124.10	192.168.1.107	TCP	1514	4000 → 36334 [ACK] Seq=22200 Ack=12671 Win=54016 Len=1448 TSval=128824419 TSecr=2162480510
51.15.124.10	192.168.1.107	TCP	1538	4000 → 36334 [PSH, ACK] Seq=23648 Ack=12671 Win=54016 Len=1472 TSval=128824419 TSecr=2162480510

Figure 4.6: Polynomial transfer from the server to a local client. Note the message length (in bytes), including overhead from the TCP, IPv4 and Ethernet headers.

These are encoded in the same manner as described earlier, for Figure 4.2, with slightly differing fields. Messages one and two are used during the group (authenticated) key exchange, which derives a key to be used with message three.

4.5.4 Network Effects and Package Sizes

The maximum frame size for Ethernet is set to 1518 bytes, where 18 bytes are part of the header, leaving 1500 for the payload for the upper network layers. This effectively makes 1500 bytes the maximum transmission unit (MTU) on the internet, and affects any application transmitting significant amounts of data, such as our group authenticated key exchange.

The message sizes, corresponding to the types in the previous section, are the following:

- Polynomial message: 5816 bytes — 5120 bytes from the polynomial, 690 bytes from the signature and 6 bytes for metadata,
- Reconciliation message: 824 bytes — 128 bytes for the reconciliation vector, 690 bytes from the signature and 6 bytes for metadata, and
- Chat message: 1088 bytes — 1024 bytes for the encrypted message, and 64 bytes of metadata (i.e., nonce, identifier, sequence numbers).

The polynomial message, which is sent twice and received $2N$ times per run of the protocol, is much larger than the MTU, and is thus fragmented into several network packets — three carrying 1448 bytes each, and a fourth carrying 1472 bytes (assuming TCP over IPv4) as shown in Figure 4.6. Thus, for each protocol run, a given participant needs to receive $2N$ polynomials, equating to $8N$ packets, plus the packet carrying the reconciliation message. The participant also needs to send two polynomials — 8 packets. Additionally, the synchronization overhead costs at least two packets — arranging unique identifiers and figuring out which participant should create the reconciliation vector.

Chapter 5

Analysis and Discussion

In this chapter, we will present a discussion on the two preceding chapters, and the results contained therein. We will focus on the research objectives set out in Section 1.3.

We will begin by explaining the primary optimization techniques in detail, along with potential trade-offs and decisions made in order to respond to **RO3** and **RO5**. Then, we will discuss the performance, as shown in the preceding chapter, both concerning the specific subroutines and the group key exchange protocols as a whole, and how it relates to the real world, responding to **RO3** and **RO4**. Finally, we will look to discuss **RO6** about constant-time cryptography in the context of RLWE.

The discussion on these research objectives will, taken together, answer the defining research objective of this thesis, **RO0**.

Comparisons are made against relevant (R)LWE instantiations and implementations for key encapsulation protocols such as NewHope, CRYSTALS-Kyber and the BCNS RLWE key exchange for TLS. We will compare against cryptosystems with a similar security level, around 100 bits of post-quantum security. We note that we compare group key exchange which have a different set of requirements than two-party KEMs do, and this in itself may explain much of the performance difference.

5.1 Notable Optimizations

Here we will explain four of the most notable areas of optimization for RLWE in the context of our implementations and our rationale for our chosen strategy in this regard, relating to modular reduction, polynomial convolution, error sampling and vectorization.

5.1.1 Modular Reduction

```
1 static const uint128_t SHIFT = 72;
```

```
2 static const uint128_t F = 103765392576;
3
4 inline uint64_t barrett_128(uint128_t a) {
5   uint128_t t = (a − ((a ∗ F) >> SHIFT) ∗ GKE_Q);
6   return t < GKE_Q ? t : t − GKE_Q;
7 }
```

Listing 5.1: The Barrett reduction with constants for our chosen field — efficiently calculating the reduction of a 128-bit unsigned integer mod $q = 45510033409$.

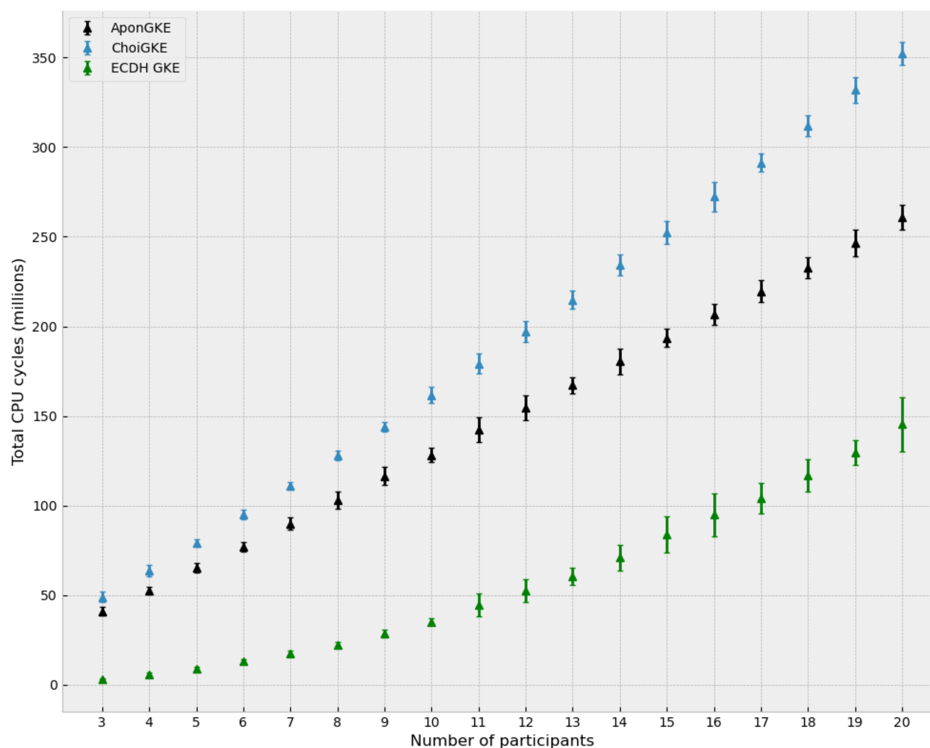


Figure 5.1: This figure compares the CPU cycles required to perform an (unauthenticated) GKE for AponGKE, ChoiGKE and pairwise ECDH using Curve25519. These data are from the unoptimized version, using no special modular arithmetic or special optimization, except for the NTT.

The remainder operator in C, “%”, CPU notably inefficient. To perform a single modulo operation through this operator, the CPU must perform an integer division, which is notoriously slow, consuming on the order of tens of processor cycles. Therefore, an efficient algorithm for modular reduction is used instead of the remainder operator, avoiding any division. There exist several options for this, but we chose to use Barrett reduction, as explained by Menezes *et al.* [MKVOV96] (Chapter 14),

leading to the subroutine illustrated in Listing 5.1. In reality, two versions were used, one for representatives we are able to guarantee are below $2^{64} - 1$, and one for results up to 2^{71}. The former is useful for reductions following addition and subtraction operations, while the latter is applied after a modular multiplication. Splitting it up this way allows us to use smaller integer sizes, which uses fewer clock cycles on average.

The speedup resulting from using Barrett reduction is significant. It constituted a speedup of up to 55%. The performance of the unoptimized implementation is shown in Figure 5.1, which, when contrasted with Figure 4.1 shows the significant performance increase achieved using relatively simple methods.

5.1.2 Polynomial Convolution

Using the Number-Theoretic Transform, in the way described in Section 3.2.1 provides a significant speedup. It is a simple algorithm with low computational complexity, and including pre-calculated constants for the transform makes the convolution operation highly efficient compared to a naive approach.

```
1  void convolution(const poly_t *a, const poly_t *b, poly_t *result) {
2
3    poly_t a_copy, b_copy;
4
5    memcpy(a_copy, a, sizeof(poly_t));
6    memcpy(b_copy, b, sizeof(poly_t));
7
8    forwards_ntt(omega, a_copy);
9    forwards_ntt(omega, b_copy);
10
11   int i;
12   for (i = 0; i < GKE_N; ++i) {
13     result->coefficients[i] = modmul(a_copy->coefficients[i], b_copy->
       coefficients[i]);
14   }
15
16   inverse_ntt(omega_inverse, result);
17 }
```

Listing 5.2: The code used for polynomial convolution, edited slightly to appear more readable and understandable for the reader.

As the conversion to and from NTT-domain is the most computationally expensive operation in the convolution process, a modification of the protocol so that we keep polynomials in NTT-domain as much as possible might be advantageous, converting back only when needed. Our current convolution approach is shown in Listing 5.2. This approach includes copying the two input polynomials, then transforming these into the NTT-domain, pairwise multiplying and transforming back the result.

The copying of the input polynomials ensures that we only need to convert the result back. If we could keep polynomials in NTT-domain implicitly, we would need fewer transformations, and could also skip the expensive copy operation. Keeping polynomials in NTT-domain is being done by, among others, Kyber [ABD+] and NewHope [AAB+].

5.1.3 Error Sampling

How we sample from Gaussian distributions is a point in which there is significant potential for an increase in performance. An open problem in the literature is to find methods that support sampling with both small and large standard deviations. As explained in Section 3.2.2, we chose to solve this using a sampler made for relatively small samples and in an inefficient manner combining these for the single large sample. The sampler used is efficient, but not specialized in any way to the needs of the protocol. Furthermore, it is not constant-time, which poses other problems related to the security of the protocols.

Ideally, we would like a sampler that can approximate a discrete Gaussian for both the smaller σ_1 and the larger σ_2. The next best thing would be two specialized samplers for each case. The most promising sampling technique in terms of performance is through the centered binomial, as used in NewHope [AAB+]. However, there has also been some promising work recently in sampling directly from discrete Gaussian distributions concerning several post-quantum digital signature schemes [ZSS18]. We intentionally leave this for future work.

5.1.4 Vectorization and Loops

Vectorization is a technique in which we increase instruction parallelism by performing the same operation on multiple data — often referred to as Single Instruction Multiple Data (SIMD). On certain processor architectures, like the x86, this is supported on the instruction set level. This means that, in theory, we can move from adding coefficients in a polynomial pairwise manner — $c_i = a_i + b_i$ — to summing up sub-vectors created by coefficients of the polynomial itself, summing up $[c_i, c_{i+1}, c_{i+2}, c_{i+3}] = [a_i, a_{i+1}, a_{i+2}, a_{i+3}] + [b_i, b_{i+1}, b_{i+2}, b_{i+3}]$ at what is essentially the same cost. This would effectively reduce the runtime of polynomial addition to a fourth of the original. However, the widespread usage of processors supporting such operations is focused on 8−, 16−, and 32−bit integers through Intel (x86) Streaming SIMD Extensions (SSE) and Advanced Vector Extensions (AVX). As we are dealing with integers over a field with a modulus of 35-bits, this is not suitable for our needs. It is possible to get around this by using floating-point numbers. However, the overhead associated with conversion between 64-bit integers and floating-point numbers impacts performance to a degree where any potential gain is close to nil. For

possible future work, the investigation of AVX-512, which supports 64-bit integers natively, and RLWE-based group key exchange could be interesting. We note that both NewHope [AAB+] and Kyber [ABD+], among other NIST contenders[1] provide AVX implementations.

As a result, we have chosen not to manually use vectorization optimizations in our implementation, though the compiler may choose to do so when suitable determined according to the compiler options found in Appendix A. It appears that the compiler will choose not to add vectorization. If this is due to using 64-bit integers, which is not supported (without using floating-point tricks) by Intel vectorization (SSE and AVX), or due to the wrapping of operations (as in Listings 5.3 and 5.4) in function calls is unclear.

```
1 void poly_add(const poly *a, const poly *b, poly *result) {
2   for (int i = 0; i < 1024; i++) {
3     result->coefficients[i] = barrett_64(a->coefficients[i] + b->
      coefficients[i]);
4   }
5 }
```

Listing 5.3: The polynomial add function, summing up two elements in the field and reducing using Barrett reduction.

```
1 void poly_add(const poly *a, const poly *b, poly *result) {
2   for (int i = 0; i < 256; i++) {
3     result->coefficients[i] = barrett_64(a->coefficients[i] + b->
      coefficients[i]);
4
5     result->coefficients[256 + i] = barrett_64(a->coefficients[256 + i]
       + b->coefficients[256 + i]);
6
7     result->coefficients[512 + i] = barrett_64(a->coefficients[512 + i]
       + b->coefficients[512 + i]);
8
9     result->coefficients[768 + i] = barrett_64(a->coefficients[768 + i]
       + b->coefficients[768 + i]);
10  }
11 }
```

Listing 5.4: An unrolled version of the polynomial add function, summing up two elements in the field and reducing using Barrett reduction.

In the same spirit as vectorization, we have taken advantage of loop-unrolling and pipelining, yielding modest performance gains. This is the only difference between the Listings 5.3 and 5.4. Unrolling allows us to fetch data into the cache at an earlier time, decreasing the time used on waiting for the memory to fetch data associated

[1] Available at the NIST website: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

with operations. Pipelining allows us to use the processor better when it otherwise would have been idle. Still, the performance gain is modest due to the compiler unrolling in a slightly different way under normal circumstances.

5.2 Performance

We presented the performance results in Sections 4.2, 4.3.2 and 4.4.4, for some important subroutines, the group key exchange protocols and authenticated versions respectively. In this section, we will provide a discussion on the results presented previously, regarding processor and memory usage, error rate, and key sizes as relates to sending data over a network.

5.2.1 CPU Cycles of Subroutines

The performance of a selection of subroutines were given in Section 4.2, and specifically Table 4.1. Here we aim to give a discussion and comparison concerning other known implementations for LWE-based cryptography.

Be aware that the comparisons made here are between measurements from different cryptosystems, and as such, contain different assumptions and operating parameteres. The most important aspect to note is that the comparisons are against cryptosystems with two participants in a typical client-server fashion.

NTT and Polynomial Convolution

Table 4.1 already makes it clear that convolution through NTT is significantly more efficient than a naive approach, being about 53 times faster. Our NTT-based convolution is slightly faster than the FFT-based one used to implement RLWE for TLS [BCNS15] — about 20000 cycles faster on average — where they use $q = 2^{32} - 1$. This modulus is specifically chosen for efficient modular reduction, is smaller than ours, and allows their implementation to use 32-bit integers internally. Thus, it is unclear why our implementation is somewhat faster. It seems like they do not precompute any constants, which would trivially make our implementation faster. Furthermore, it could be related to the compilation flags used, as we allow for SSE and AVX optimization if the compiler finds it useful — which in practice is rarely — but BCNS do not[2]. Another plausible explanation is that the usage of the Cooley-Tukey forwards and Gentleman-Sande inverse NTT is faster than the Nussbaumer FFT.

Comparing with Newhope and Crystals-Kyber — both LWE-based cryptosystems and contenders in the NIST standardization contest — which use moduli $q = 12289$ and $q = 3329$ respectively, both representable as 16-bit integers, indicates that

[2] See Makefile https://github.com/dstebila/rlwekex/blob/master/Makefile, commit `1fbbd8c`

Implementation	NTT/FFT	Inverse NTT/FFT	Convolution
BCNS [BCNS15]	-	-	342 800
CRYSTALS-Kyber [ABD+]	320	290	-
NH-512-CPA-KEM [AAB+]	21 772	23 384	-
NH-512-CPA-KEM (AVX) [AAB+]	4 888	4 820	-
Singh [Sin15]	95 300	98 800	300 500
This thesis	98 407	117 253	327 635

Table 5.1: Polynomial convolution performance of a selection of RLWE-based cryptosystems. Due to differents in measuring, we include both the NTT/FFT transformations and the convolution as a whole, to give a fair comparison. We note that our implementation uses a larger modulus than all we compare to here.

our NTT implementation is not as performant as possible. Kyber uses only 320 CPU cycles for the forward transformation and 290 for the inverse transformation [ABD+], while NewHope uses 21 772 for the forwards and 23 384 for the inverse transformations [AAB+], as seen in Table 5.1, which provides a comparison of the performance of the discussed convolution techniques. Compared to our 98 407 and 117 253, the speed difference with Kyber is significant but explainable partly due to the assumptions for that cryptosystem. Furthermore, Kyber's low CPU cycle count is due to a highly optimized version in Assembly heavily utilizing x86 vectorization. It is possible that our implementation could have been more performant had we taken a similar approach. The performance difference to NewHope is explainable, mostly due to their usage of smaller integers. Finally, we see that Singh [Sin15], who uses $q = 40961$, is similar in performance to us despite using a significantly smaller modulus. The small difference is most likely due to our usage of Cooley-Tukey forwards and Gentleman-Sande inverse transforms, while Singh uses Cooley-Tukey inverse and Gentleman-Sande forwards. The reason for this is unclear, and we find no indication that this is better than what Longa and Naehrig [LN16] suggest, and which we implement.

Reconciliation

The reconciliation subroutines are only run once per participant, making optimal performance here less of a focus than in convolution, for instance. Comparing the CPU usage of our subroutines, `RecMsg()`, which directly corresponds to the doubling and crossrounding function in Peikert's error reconciliation protocol described in Section 2.4.4, and the corresponding function used by BCNS [BCNS15] yields an interesting result. It is for practical purposes identical to ours. Yet their subroutines analogous to `RecMsg()` use 23 500 CPU cycles, against 67 753 for our implementation. Singh [Sin15] achieves as few as 12 900 cycles for $n = 1024$. We suspect that these

more performant results stem from the way randomness is generated in the `dbl()`-subroutine, which is used in `RecMsg()`. The implementations other than the source and method for getting random bits is, for practical purposes, the same between all these three. BCNS and Singh generate random numbers using AES as a PRNG, which is seeded by a CSPRNG (e.g., `/dev/urandom`), thus keeping a random state and not initiating every time it needs a random number. This differs from our approach, which simply reads from `/dev/urandom` directly. Using a PRNG in the program directly is faster than reading from the operating systems random number generator, which is likely a major reason for these implementations being faster in this regard.

Function	Implementation	CPU cycles (CT)	CPU cycles (non-CT)
`RecMsg()`	BCNS [BCNS15]	23 500	21 300
`RecMsg()`	Singh [Sin15]	-	12 900
`RecMsg()`	This thesis	67 753	-
`RecKey()`	BCNS [BCNS15]	14 400	6 800
`RecKey()`	Singh [Sin15]	-	4 200
`RecKey()`	This thesis	23 152	-

Table 5.2: A comparison of the performance of the reconciliation functions of BCNS, Singh and the work presented in this thesis. These all use Peikert reconciliation, which provides for a fair comparison.

The approach utilizing a PRNG in the *dbl*-function is secure in relation to the security parameter λ given that the initial seed is randomly selected uniformly from $\{0,1\}^\lambda$, and a suitable PRNG is used. An example is AES, where the initial data is uniformly random, encrypted under a uniformly random key iteratively. We could also use SHA-3, with a uniformly random bit-string of length λ as the initial state, which is the correct usage of a cryptographic primitive for this task.

The other reconciliation function, `RecKey()` is closer to that of BCNS — 27 196 to their 14 400. The difference here stems from the different types of integer sizes — 32-bit against 64-bit unsigned integers — as otherwise, the functions are the same.

Table 5.2 provides a summarized comparison of the performance of the reconciliation techniques discussed above.

Error Sampling

As reasoned in Section 3.2.2, implementing an error sampler specifically for the implementations provided in this thesis was deemed out of scope. We therefore use the one created by Albrecht *et al.* [AW18], with a trick to make it work for σ_2, which is larger than what it originally supports. There are some challenges with

this sampler. First and foremost, it does not protect against side-channel attacks. Furthermore, it is not suitable for sampling with large standard deviations, as this was not the intended goal of the implementation.

Implementation	CPU cycles (CT)	CPU cycles (non-CT)
Singh [Sin15]	-	663 400
BCNS [BCNS15]	1 042 700	668 000
Kyber-512 (AVX) [ABD+]	20 004	-
NH-512-CPA-KEM [AAB+]	56 236	-
This thesis (σ_1)	-	307 745
This thesis (σ_2)	-	6 012 056

Table 5.3: A comparison of the error sampling methods demonstrated in various papers and specifications. We distinguish between constant-time (CT) and non-constant-time (non-CT) sampling as constant-time is ideal, but non-constant-time can yield better performance at the cost of security against side-channel attacks. As a result they are in seperate categories. Note that we are using the Gaussian sampler of Albrecht [AW18] alone for σ_1, and composited for σ_2.

Regardless, the efficiency of Albrecht's sampler is acceptable when compared with other highly optimized implementations, as shown in Table 5.3. We believe a vital takeaway of this comparison is that for an efficient implementation, we would require a specialized discrete Gaussian sampler that could support both small and relatively large standard deviations. This would allow us to avoid sampling several times for the larger σ_2. Due to Gaussians being notoriously difficult[3] to implement in practice, a centered binomial could be more suitable.

Uniform Sampling

Uniform sampling is in our implementation done, as described in Section 4.2, through reading bytes from the operating system CSPRNG (`/dev/urandom`). In the same vein as the discussion regarding reconciliation, if we had used a PRNG seeded by a TRNG, like AES, (which though strictly speaking is not a PRNG, for practical purposes could function as a PRNG using a cryptographically secure random key and initialization and data, chained together in succession) or SHA-3, we could have achieved a near-NewHope performance in this regard. The only difference here is due to us requiring 1024 35-bit integers, while NH-512-CPA-KEM needs 512 16-bit integers. The resulting performance gap should then be smaller, estimated to be at about two to four times the number of cycles used by NH-512-CPA-KEM, due to

[3] See, for instance, https://groups.google.com/a/list.nist.gov/forum/#!msg/pqc-forum/7Z8x5AMXy8s/Spyv8VYoBQAJ

using approximately double the size in integers and a dimension of 1024 compared to their dimension of 512.

Uniformly sampling using a seed generated by a TRNG as the initial state in a chained cryptographic hash function, which is then expanded iteratively is secure due to the hash function generating seemingly uniformly random bytes. This is enough for the generation of a uniformly random polynomial, which is public in any case.

Previous Work	CPU cycles (constant-time)
NH-512-CPA-KEM [AAB+]	10 804
This thesis	149 372

Table 5.4: The CPU cycles used to sample a uniform ring element by the implementation presented in this thesis, compared with the implementation in the NewHope cryptosystem.

We were unable to find data on processing time for uniform generation by BCNS, Singh or Kyber, which is the reason these are absent in this comparison.

5.2.2 Error Rate

The error rate, as demonstrated in Table 4.3, is, for 8 or more participants, significant. Error rate at $\approx 20\%$ for $N = 20$ is, in most circumstances, unacceptable. This is far above the error rate of comparable protocols, operating with "negligible" error rates in the case of NewHope [AAB+], and 2^{-100} in the case of CRYSTALS-KYBER [ABD+]. In their paper [BCNS15] on implementing an RLWE key exchange in TLS, the authors notably do not mention an error rate.

The error rate given by the instantiations given in this thesis may be significantly decreased by utilizing an error correction approach where we correct errors based on sets of coefficients, and not just one-by-one. For instance, Saarinen [Saa17] provides some improvements over Peikert reconciliation, going from $\approx 2^{-60}$ error rate to $\leq 2^{-128}$ — which is a significant improvement. This could be possible for the group key exchange protocols presented here.

Another approach to decrease the error rate is to use a larger modulus. As we, in the implementation of AponGAKE and ChoiGAKE already use 64-bit integers internally, this should be a relatively straight-forward procedure once suitable moduli and accompanying parameter sets are determined. The increase in size of modulus may be to the detriment of the protocol security for fewer users, as the signal-to-noise ratio will necessarily be lower. As such, the most suitable solution may be to use our parameter set for $N \leq 10$, and another for $N \in \{10, \ldots, 20\}$, having a parameter set

for each range of participants, to better achieve successful error reconciliation while keeping a reasonable noise level.

5.2.3 Key Sizes

To reiterate, the public and neighbourhood keys of our implementations fit into 5120 bytes — 1024 coefficients which are modulo a 35-bit number fits into 5 bytes totaling 5120 bytes for a polynomial. We provide a comparison to previous work in Table 5.5. The comparison illustrates that our approach is, compared to other post-quantum schemes, inefficient in terms of key sizes. The key size comes from using large fields in order to accommodate the necessary signal-to-noise ratio and the required difference between the standard deviations of the two error distributions for the security of the protocols.

Cryptosystem	Public key size (bytes)
X25519 [LMT16]	32
RSA-2048 [RSA78]	256
KYBER-512 [ABD+]	800
NH-512-CPA-KEM [AAB+]	928
This thesis	5 120

Table 5.5: Comparison between the key sizes of work here, and previous work, using notable key exchange and key encapsulation mechanisms from both classical schemes and post-quantum schemes.

Theoretically, it is possible to fit a number existing modulo a $35-$bit integer into $36-$bits of data. However, this does not align with the $8-$bit sizes of bytes, meaning there is no conventional support for this on a processor level. This adds significant complexity, for a gain of 4 bits per coefficient, adding up to 512 bits for a public key. This is not worth the added complexity due to us still requiring the same number of network packets to transfer a public key regardless.

Key Compression

The compression of public keys sent over the network was considered in the process of developing a real-world-like implementation; however, we found two good reasons not to do so — Compression of public keys fundamentally alters the protocol, according to NIST [AASA+, p. 7]. Furthermore, the potential gain in network performance due to the potentially smaller public keys is offset by the actual compression and decompression operations. Additionally, the public keys should be indistinguishable from uniform over a large field; in other words holding much entropy, meaning compression would be ineffective.

5.2.4 Performance of GKE

We previously presented the performance of the group key exchange protocols in Section 4.3.2. In Figure 4.1 we see a graph depicting the performance of AponGKE, ChoiGKE and an ECDH-GKE. Our implementations seem to beat the ECDH-GKE version when the number of participants grows to $N = 9$. This can partially be explained by the exponential growth of the chosen ECDH, and an argument can be made that we are not providing a fair comparison in that regard. However, we would argue that comparing with the ECDH-based group key exchange where all participants perform a key exchange with every other participant is a useful, if not fair, comparison due to the widespread adoption of (a more complex version of) this group key exchange in various practical applications such as the Signal app [Sig] and Wire [Wir]. Due to the complexity of AponGKE and ChoiGKE being linear, and the ECDH-GKE being quadratic, it is a given that for a certain number of participant we would have to beat it. However, reaching the performance of the "state-of-the-art" at $N = 9$ participants is pretty good, especially considering the security goals. In a way, it may be seen as unfair comparing the other way around as well, as ECDH is only safe in a world without practical quantum computers, but our protocol instantiation of ChoiGKE and AponGKE have a theoretical (approximate) 105 bits of post-quantum security.

Overall, the performance of the GKE is acceptable, but could likely be improved upon given the usage of the optimization techniques discussed above concerning specific subroutines.

5.2.5 The Impact of Authentication

Comparing the performance of the group key exchange protocols with the authenticated versions of Sections 4.3.2 and 4.4.4 shows how dominant the digital signature schemes are in terms of processing requirements. Comparing Figures 4.1 and 4.3 shows that the ECDH-GAKE is close in terms of performance to its unauthenticated version, while both the FALCON and qTESLA versions of AponGAKE and ChoiGAKE are significantly affected by the signature schemes. In both the graphs of Figures 4.3 and 4.4 the digital signature algorithms are the dominant performance factor. In the case of qTESLA, it is also the dominating factor when sending networked messages due to the size of the signatures and public keys, with the signature of a message occupying close to two network packets alone.

In Figure 4.4 we have excluded the generation of the verification/signing key pairs, which could be a plausible scenario where these keys are generated for long-term usage and exchanges out-of-band. Still the subroutines fro¨m the digital signature algorithm is dominating — using more than half the processor cycles on verification

Method	
38.8%	GAKE_apon`PQCLEAN_FALCON512_CLEAN_fpr_add
13.8%	GAKE_apon`PQCLEAN_FALCON512_CLEAN_fpr_mul
▶ 5.4%	GAKE_apon`KeccakF1600_StatePermute
4.3%	GAKE_apon`zint_rebuild_CRT.constprop.3
3.4%	GAKE_apon`PQCLEAN_FALCON512_CLEAN_fpr_scaled
2.9%	GAKE_apon`PQCLEAN_FALCON512_CLEAN_fpr_div
2.5%	GAKE_apon`solve_NTRU_intermediate
2.2%	GAKE_apon`modp_mkgm2
1.9%	GAKE_apon`poly_sub_scaled
1.7%	GAKE_apon`poly_small_mkgauss
1.2%	GAKE_apon`modp_NTT2_ext.part.0.constprop.0
1.2%	GAKE_apon`keccak_inc_squeeze.constprop.1
1.1%	GAKE_apon`modmul
1.1%	GAKE_apon`PQCLEAN_FALCON512_CLEAN_FFT

Figure 5.2: The top subroutines used in the AponGAKE when excluding key pair generation. All subroutines shown except `GAKE_APON‘modmul` are from the OQS implementation of FALCON. This further demonstrates how dominating the authentication part is. All other subroutines use less than 1% of the processor cycles.

and signing, shown in Figure 5.2. The image is taken from the JetBrains CLion[4] profiling tool, which uses perf[5] during a profiling run of the FALCON version of AponGAKE. We also note that there is a memory-time trade-off between qTESLA and FALCON, with qTESLA using significantly more memory and less time in generating key pairs, but with a worse run-time regarding the signing and verification of messages than FALCON as seen when comparing Figures 4.3 and 4.4. It is also apparent that FALCON uses a significant amount of time in the creation of key pairs, with the signing and verification operations consuming a comparatively small portion of the processor cycles.

It is possible that other digital signature algorithms are more suitable, and that what the standardization process lands on would be more suitable. The intention of the inclusion here is to provide the reader with a detailed implementation of an authenticated group key exchange, and what digital signature algorithm is used is of little consequence as it may easily be exchanged with another.

[4] A C/C++ IDE: https://www.jetbrains.com/clion/, using version `2020.1.1`

[5] A Linux performance tool, see https://perf.wiki.kernel.org/index.php/Main_Page, using version `5.6.g7111951b8d49`

5.2.6 On the Real-world Implementation

In Section 4.5, we provided a real-world-like implementation in the form of a group messaging application, which is one of many applications in which a group key exchange would be useful.

The dominating factor in setting up a key exchange using either AponGAKE or ChoiGAKE is the time required to synchronize all the participants. As no one can calculate their neighbourhood-keys before getting their neighbours' public keys, and no one can calculate the shared secret before receiving all neighbourhood-keys, the protocol execution has to happen in a synchronized fashion. The other major factor is the network delay, which is on the order of tens of milliseconds. For the processor used in the tests, 10ms corresponds to 42 003 287 cycles, which is on the order of what the entire group key exchange uses of processing cycles. The networked messages have to be sent several times as well, and is divided into several packets due to the message sizes being above the maximal transfer unit of Ethernet further compounding the network delay. This seems to be a trade-off one would have to make in order to ensure post-quantum safety for many protocols, as none seem to beat the current quantum-insecure paradigm when it comes to key sizes and computational performance.

5.2.7 Potential Trade-offs for Group Key Exchange

Using either AponGAKE, ChoiGAKE, or any other similar protocol in the future will yield a significant benefit over the current paradigm — post-quantum security. The inherent performance issues notwithstanding, in the process, we may make other desired properties of our group key exchange protocols unobtainable, or obtainable only at a high cost. Such properties that at least AponGAKE and ChoiGAKE do not provide, but are provided for by certain currently used protocols are deniability and post-compromise security. Furthermore, a major disadvantage of these group key exchange protocols is that all participants need to share keying material in a synchronous fashion, which is not how, for example, current messaging applications work. Synchronous group key exchange can be suitable for certain applications such as video conferencing, but is certainly more limited in its usefulness than non-interactive group key exchange protocols would be.

As a result of the problem with interactivity, it is possible that future post-quantum group key exchange could be performed using NIST-standardized KEMs coupled with digital signature algorithms or other techniques to achieve desired cryptographic properties. However, if such an approach does not achieve a linear growth in computational complexity and a constant number of rounds, this will not be more efficient than AponGKE and ChoiGKE for larger numbers of participants.

5.3 Constant-time RLWE Cryptography

We previously defined constant-time cryptography in Section 2.2.3, which protects against timing-related side-channel attacks, which we introduced in Section 2.2.3. Constant-time cryptographic operations are important, as the volatility in the time required to perform these may leak secret data [YGH17] [Sch19]. Implementing constant-time cryptography is a difficult process, further complicated by RLWE cryptography being a young area of research. In this section, we will explain our efforts to make our implementation as constant-time as possible within our research scope.

```
uint64_t test = (uint64_t) b->coefficients[i] << (uint64_t) 1;

if (rec->coefficients[i] == 0) {
  if (test >= 34132525058 && test <= 79642558468) {
    key->coefficients[i] = 1;
  }
} else {
  if (test >= 11377508352 && test <= 56887541762) {
    key->coefficients[i] = 1;
  }
}
```

Listing 5.5: This code is part of the Peikert reconciliation protocol — `recKey()`, implemented the naive, non-constant-time way.

```
uint64_t test = (uint64_t) b->coefficients[i] << (uint64_t) 1;
uint64_t t1 = (ct_eq_u64(rec->coefficients[i], 0) & ct_ge_u64(test,
    34132525058) & ct_le_u64(test, 79642558468));

uint64_t t2 = (ct_eq_u64(rec->coefficients[i], 1) & ct_ge_u64(test,
    11377508352) & ct_le_u64(test, 56887541762));

key->coefficient[i] = 0;
key->coefficient[i] |= t1;
key->coefficient[i] |= t2;
```

Listing 5.6: Here we provide a constant-time version of the inner loop of the `recKey()`-subroutine, borrowing certain helper-functions from BCNS [BCNS15].

In our implementations we have attempted to remove any conditional branching and table lookups dependent on secret data. For instance, making the `RecKey()`-function, which inner loop is shown in Listing 5.5, constant-time includes removing all the branching statements, replacing them with functions that do this in constant time. The constant-time version is depicted in Listing 5.6, borrowing from the BCNS implementation[6] the constant-time comparison functions — for instance `ct_eq_u64()` checks in constant time whether two unsigned 64-bit integers are equal.

[6] Available at https://github.com/dstebila/rlwekex/blob/master/rlwe.c, commit `1fbbd8c`

Much of the cryptographic core functions of AponGKE and ChoiGKE are polynomial addition, subtraction, and convolution. Addition and subtraction are constant-time by themselves. The subroutines `barrett_64()` and `barrett_128()` — which reduces a 64-bit integer or a 128-bit integer modulo q respectively — only performs operations which are constant-time, at least on our target instruction set architecture, the Intel x86. We believe that our NTT-implementation, and thus also the polynomial convolution, is constant-time as well. However, we have not taken advantage of more advanced techniques such as blinded polynomial multiplication [Saa16], which we believe is the way to go when implementing real-world polynomial convolution.

Figure 4.1 shows the CPU cycles used for a variable number of participants of both protocols. Furthermore, it plots the standard deviation of the samples as bars above and below, with a marker in the middle indicating the sample mean. We believe that the small standard deviation is an indicator of the implementation being close to constant-time. However, we remark that it cannot be constant-time due to the sampler, and possibly other unknown (to us) causes.

Finally, as noted previously, the error sampling library, DGS [AW18] is not constant-time. Previously in this chapter, we noted that an efficient implementation of AponGKE and ChoiGKE would require a specialized sampler. This specialized sampler should also be constant-time, which would solve this problem as well.

Chapter 6

Conclusion

This chapter provides a conclusion on the work presented in the previous chapters, and suggests some future work that were out of scope for this thesis.

6.1 Research Objectives

In Section 1.3 we set the following research objective: *Design a real world implementation for group key exchange based on the Ring Learning-With-Errors problem.* We will briefly summarize the outcomes of the secondary research objectives.

– **RO1**: *What proposed protocols are suitable candidates for post-quantum group key exchange based on the RLWE problem?*
  AponGAKE and ChoiGAKE are interesting candidates that could be used for post-quantum group key exchange. The discussion in Chapter 5 indicates that the performance is decent compared with RLWE-based schemes for two-party key exchange — especially when considering the different use cases. A disadvantage of these is the interactivity requirement, and the lack of auxiliary properties, which are widely used in today's protocols for group key exchange.

– **RO2**: *Instantiate a protocol for group key exchange based on the RLWE problem with adequate security supporting a reasonable number of participants.*
  An instantiation is provided in Chapter 3, which supports multiple participants, though the error rate is non-negligible when the number of users is above 10 for our proposed parameter set. A reduction in error rate could be achieved through specialized parameter sets for larger numbers of users.

– **RO3**: *Is this protocol instantiation usable in terms of delay, processing requirements and quality of experience?*
  The protocol is usable on the platforms tested in this work, primarily x86 workstations and servers. For constrained networks and devices, it may not be ideal, but an increase in computation time and key sizes is to be expected

for post-quantum primitives, including those discussed in this thesis. At the moment, there exists no better performing post-quantum group key exchange implementation based on RLWE.

- **RO4**: *What, if any, trade-offs must be made in order to achieve post-quantum security in a group setting?*
  Given the proposed protocols for group key exchange based on RLWE, we seem to require interactivity in the key exchange, which makes deniability and backward security difficult. Furthermore, some performance overhead, which is to be expected, is induced. We discussed all of these trade-offs in Section 5.2.7.

- **RO5**: *How can we implement performant RLWE cryptosystems?*
  Implementing performant RLWE-based cryptosystem is an open-ended field of research. This thesis outlines some techniques, such as using NTT and techniques for modular reduction. For a thorough discussion, see Chapter 5.

- **RO6**: *How can we secure real-world implementations of RLWE cryptography from side-channel attacks?*
  Removing branching instructions and secret-based table lookups removes most of the attack surfaces when it comes to side-channel attacks. However, the techniques used in this thesis, which we discussed in Section 5.3, are not complete, and much remains to be understood in making constant-time efficient implementations.

In conclusion, we have provided an instantiation and implementation of two post-quantum group key exchange protocols, along with a real-world prototype, completing our primary research goal successfully. In the process, we have been able to answer all of the research objectives and questions posed; however, there remains a considerable amount of potential regarding future work.

6.2 Future work

In this thesis, we have touched upon a multitude of interesting topics that might be desirable to know more about. We will name a selection of these here.

The most interesting future work, in our opinion, is in finding methods to perform post-quantum non-interactive group key exchange, which to our knowledge, does not yet exist in a practical manner.

Furthermore, practical implementations of common RLWE mathematics such as the NTT using instruction set level optimizations for larger integers, like the x86 AVX-512, could be useful in speeding up RLWE even more. For embedded devices,

increasing the performance probably requires different approaches, which could also be an avenue for new research.

In the process of working with this thesis, we had considerable issues in finding acceptable discrete Gaussian samplers in terms of performance and closeness to a real Gaussian. Should there be a single performant sampler capable of sampling both small and large errors, or should we use two specialized samplers, one for each? Perhaps using a centered binomial is the way to go, but what constraints must be taken into consideration, especially for distributions as large as the ones defined by σ_2 in Chapter 3? Further research into Gaussian — or near-Gaussian — sampling could considerably increase the performance of many protocols.

References

[AAB+] Erdem Alkim, Roberto Avanzi, Joppe Bos, Léo Ducas, Antonio de la Piedra, Thomas Pöppelmann, Peter Schwabe, Douglas Stebila Martin R. Albrecht, Emmanuela Orsini, Valery Osheter, Kenneth G. Paterson, Guy Peer, and Nigel P. Smart. NewHope — Algorithm Specifications and Supporting Documentation. https://newhopecrypto.org/data/NewHope_2020_04_10.pdf. (Fetched 2020-04-16).

[AASA+] Gorjan Alagic, Jacob Alperin-Sheriff, Daniel Apon, David Cooper, Quynh Dang, Yi-Kai Liu, Carl Miller, Dustin Moody, Rene Peralta, Ray Perlner, Angela Robinson, and Daniel Smith-Tone. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf. (Fetched 2020-04-16).

[AB75] Ramesh C Agarwal and C Sidney Burrus. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, 63(4):550–560, 1975.

[ABB+15] Daniel Augot, Lejla Batina, Daniel J. Bernstein, Joppe Bos, Johannes Buchmann, Wouter Castryck, Orr Dunkelman, Tim Guneysu, Shay Gueron, Andreas H¨ulsing, Tanja Lange, Mohamed Saied Emam Mohamed, Christian Rechberger, Peter Schwabe, Nicolas Sendrier, Frederik Vercauteren, and Bo-Yin Yang. Initial recommendations of long-term secure post-quantum systems. https://pqcrypto.eu.org/docs/initial-recommendations.pdf, sep 2015.

[ABD+] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. https://pq-crystals.org/kyber/data/kyber-specification-round2.pdf. (Fetched 2020-04-16).

[ACD+18] Martin R Albrecht, Benjamin R Curtis, Amit Deo, Alex Davidson, Rachel Player, Eamonn W Postlethwaite, Fernando Virdia, and Thomas Wunderer. Estimate all the {LWE, NTRU} schemes! In *International Conference on Security and Cryptography for Networks*, pages 351–367. Springer, 2018.

[ADPS15] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - a new hope. Cryptology ePrint Archive, Report 2015/1092, 2015. https://eprint.iacr.org/2015/1092.

[ADSGK19] Daniel Apon, Dana Dachman-Soled, Huijing Gong, and Jonathan Katz. Constant-Round Group Key Exchange from the Ring-LWE Assumption. Cryptology ePrint Archive, Report 2019/398, 2019. https://eprint.iacr.org/2019/398.

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

[AW18] Martin R. Albrecht and Michael Walter. dgs, Discrete Gaussians over the Integers. Available at https://bitbucket.org/malb/dgs, 2018.

[BBD09] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Introduction to post-quantum cryptography*. Springer Berlin Heidelberg, 2009.

[BCNS15] Joppe W. Bos, Craig Costello, Michael Naehrig, and Douglas Stebila. Post-Quantum Key Exchange for the TLS Protocol from the Ring Learning with Errors Problem. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 553–570. IEEE Computer Society, 2015.

[BD95] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system. In Alfredo De Santis, editor, *Advances in Cryptology — EUROCRYPT'94*, pages 275–286. Springer Berlin Heidelberg, 1995.

[Ber06] Daniel J Bernstein. Curve25519: new Diffie-Hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[Ber08] Daniel J Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008.

[BMS06] Emmanuel Bresson, Mark Manulis, and Joerg Schwenk. On security models and compilers for group key exchange protocols. Cryptology ePrint Archive, Report 2006/385, 2006. https://eprint.iacr.org/2006/385.

[Bra16] Matt Braithwaite. Experimenting with Post-Quantum Cryptography. https://security.googleblog.com/2016/07/experimenting-with-post-quantum.html, 2016. (Reviewed 2019-11-08).

[CDW16] Ronald Cramer, Léo Ducas, and Benjamin Wesolowski. Short stickelberger class relations and application to ideal-svp. Cryptology ePrint Archive, Report 2016/885, 2016. https://eprint.iacr.org/2016/885.

[CGCD+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

[CGCG+17] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. Cryptology ePrint Archive, Report 2017/666, 2017. https://eprint.iacr.org/2017/666.

[CHK20] Rakyong Choi, Dongyeon Hong, and Kwangjo Kim. Constant-round dynamic group key exchange from rlwe assumption. Cryptology ePrint Archive, Report 2020/035, 2020. https://eprint.iacr.org/2020/035.

[Cos19] Craig Costello. Supersingular isogeny key exchange for beginners. Cryptology ePrint Archive, Report 2019/1321, 2019. https://eprint.iacr.org/2019/1321.

[DB05] Ratna Dutta and Rana Barua. Constant round dynamic group key agreement. In *International Conference on Information Security*, pages 74–88. Springer, 2005.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. Cryptology ePrint Archive, Report 2013/383, 2013. https://eprint.iacr.org/2013/383.

[DH76] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 1976.

[FHK+18] Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, Thomas Pornin, Thomas Prest, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon: Fast-Fourier lattice-based compact signatures over NTRU. *Submission to the NIST's post-quantum cryptography standardization process*, 2018.

[Fra14] John B. Fraleigh. *A First Course in Abstract Algebra. Seventh edition.* Pearson Education Limited, 2014.

[Gro96] Lov K. Grover. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219. ACM, 1996.

[HMPR04] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.

[HPRR19] James Howe, Thomas Prest, Thomas Ricosset, and Mélissa Rossi. Isochronous gaussian sampling: From inception to implementation. Cryptology ePrint Archive, Report 2019/1411, 2019. https://eprint.iacr.org/2019/1411.

[JD12] Xiaodong Lin Jintai Ding, Xiang Xie. A Simple Provably Secure Key Exchange Scheme Based on the Learning with Errors Problem. Cryptology ePrint Archive, Report 2012/688, 2012. https://eprint.iacr.org/2012/688.

[Koc96] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[Kre02] Erwin Kreyszig. *Advanced engineering mathematics.* John Wiley and Sons, Inc., 2002.

[Kri19] Simen Been Kristiansen. Evaluating post-quantum multi-party key exchange. Project report in TTM4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, Dec. 2019.

[KRVV19] Angshuman Karmakar, Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. Pushing the speed limit of constant-time discrete Gaussian sampling. A case study on the Falcon signature scheme. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[KY03] Jonathan Katz and Moti Yung. Scalable Protocols for Authenticated Group Key Exchange. In *Advances in Cryptology - CRYPTO 2003*, pages 110–125. Springer Berlin Heidelberg, 2003.

[Lan16] Tanja Lange. PQCRYPTO Overview. In *PQCrypto Scientific advisory board meeting*, 2016.

[LLZ+18] Xianhui Lu, Yamin Liu, Zhenfei Zhang, Dingding Jia, Haiyang Xue, Jingnan He, Bao Li, and Kunpeng Wang. LAC: Practical Ring-LWE Based Public-Key Encryption with Byte-Level Modulus. Cryptology ePrint Archive, Report 2018/1009, 2018. https://eprint.iacr.org/2018/1009.

[LM09] Vadim Lyubashevsky and Daniele Micciancio. On bounded distance decoding, unique shortest vectors, and the minimum distance problem. In *Annual International Cryptology Conference*, pages 577–594. Springer, 2009.

[LMT16] A. Langley, Hamburg M., and S. Turner. Elliptic Curves for Security. RFC 7748, Internet Research Task Force, January 2016.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. *J. ACM*, 60(6):43:1–43:35, November 2013.

[MKVOV96] Alfred J Menezes, Jonathan Katz, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996. Fetched from http://cacr.uwaterloo.ca/hac/ (2020-02-04).

[NAB+17] Michael Naehrig, Erdem Alkim, Joppe Bos, Léo Ducas, Karen Easterbrook, Brian LaMacchia, Patrick Longa, Ilya Mironov, Valeria Nikolaenko, Christopher Peikert, et al. Frodokem. *Technical report, National Institute of Standards and Technology*, 2017.

[NIS16] NIST. Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process. https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/callfor-proposals-final-dec-2016.pdf, 2016.

[Pei14] Chris Peikert. Lattice cryptography for the internet. Cryptology ePrint Archive, Report 2014/070, 2014. https://eprint.iacr.org/2014/070.

[Pei16] Chris Peikert. How (not) to instantiate ring-LWE. In *International Conference on Security and Cryptography for Networks*, pages 411–430. Springer, 2016.

[Pla18] Rachel Player. *Parameter selection in lattice-based cryptography.* PhD thesis, PhD thesis, Royal Holloway, University of London, 2018.

[Reg05] Oded Regev. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC '05, pages 84–93. ACM, 2005.

[Roc09] Daniel S Roche. Space-and time-efficient polynomial multiplication, 2009.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Saa16] Markku-Juhani O. Saarinen. Arithmetic coding and blinding countermeasures for lattice signatures. Cryptology ePrint Archive, Report 2016/276, 2016. https://eprint.iacr.org/2016/276.

[Saa17] Markku-Juhani O. Saarinen. Hila5: On reliability, reconciliation, and error correction for ring-lwe encryption. Cryptology ePrint Archive, Report 2017/424, 2017. https://eprint.iacr.org/2017/424.

[Sch19] Peter Schwabe. Something with implementations. PQCRYPTO Summer School on Post-Quantum Cryptography 2017, June 2019. https://2017.pqcrypto.org/school/slides/pqimp.pdf.

[Sco17] Michael Scott. A note on the implementation of the Number Theoretic Transform. Cryptology ePrint Archive, Report 2017/727, 2017. https://eprint.iacr.org/2017/727.

[Sho94] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[Sig] Signal. Signal Documentation. https://signal.org/docs/. (Reviewed 2019-11-19).

[Sin15] Vikram Singh. A Practical Key Exchange for the Internet using Lattice Cryptography. Cryptology ePrint Archive, Report 2015/138, 2015. https://eprint.iacr.org/2015/138.

[SP19] Douglas R Stinson and Maura B Paterson. *Cryptography — Theory and Practice.* CRC Press, 2019. Fourth edition.

[Vai20] Vaillant, Loup. Monocypher — Boring crypto that simply works. https://monocypher.org, 2020. (Accessed 2020-02-16).

[VE85] Wim Van Eck. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers & Security*, 4(4):269–286, 1985.

[Wie14] Roelf J. Wieringa. *Design science methodology for information systems and software engineering.* Springer, 2014. 10.1007/978-3-662-43839-8.

[Wir] Wire Swiss GmbH. Wire Security Whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf. (Reviewed 2019-11-19).

[WP06] André Weimerskirch and Christof Paar. Generalizations of the Karatsuba Algorithm for Efficient Implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006.

[YGH17] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[Zoo16] Zoom Video Communications, Inc. Global Infrastructure and Security Guide. https://zoom.us/docs/doc/Zoom_Global_Infrastructure.pdf, January 2016. Fetched (2020-05-12).

[ZSS18] Raymond K. Zhao, Ron Steinfeld, and Amin Sakzad. FACCT: FAst, Compact, and Constant-Time Discrete Gaussian Sampler over Integers. Cryptology ePrint Archive, Report 2018/1234, 2018. https://eprint.iacr.org/2018/1234.

Appendix

Compiler options

We use CMake to generate the Makefiles for our projects, using the below file as CMakeLists.txt. A release-candidate is generated by running the following commands:

```
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
```

All the accompanying code used in this thesis may be found at https://github.com/simenbkr/rlwe-gke, commit `1795995`.

```

cmake_minimum_required(VERSION 3.15)
project(<projectname> C)

set(CMAKE_C_STANDARD 99)

if(NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release)
endif()

set(CMAKE_C_FLAGS_RELEASE "${CMAKE_C_FLAGS_RELEASE} -Wall -Wextra -O3 -fomit-frame-pointer -msse2avx -march=corei7-avx -s")

add_executable(<projectname> main.c <other files>)
target_link_libraries(<projectname> -static dgs oqs mpfr gmp m pthread)
```

Listing A.1: CMakeLists.txt for the compilation of the project binaries used to collect data. Similar makefiles were used for the other versions as well.

Appendix B
Raw data

This data is presented as a graph in Chapter 4. The underlying measurements stem from the code published at https://github.com/simenbkr/rlwe-gke, using commit id `1795995`. The raw data can be found in the subfolders of `create_charts`.

B.1 GAKE

B.1.1 ECDHGAKE

Participants	Mean CPU cycles	Standard deviation
3	8093472	1498506
4	14135599	682511
5	22836023	1682192
6	33648522	2780582
7	47668281	5837126
8	61881527	6385294
9	79643507	8746228
10	99178334	11269130
11	118717372	11682242
12	139865464	7194238
13	166818467	15236827
14	192118335	16539704
15	222760510	19979406
16	254992546	27853728
17	282003072	13733843
18	318596813	16721442
19	352190055	18359774
20	396270971	27978079

B.1.2 FALCON-AponGAKE

Participants	Mean CPU cycles	Standard deviation
3	321034266	38548090
4	423477523	45967385
5	517012130	45689436
6	621370685	51222242
7	730904006	59255768
8	856203031	74601518
9	944236732	69937583
10	1045762534	66782451
11	1154995350	69750982
12	1265623825	71680137
13	1403294315	96639183
14	1545537127	95142660
15	1635672180	91143605
16	1759372995	96584029
17	1858122813	95820400
18	1999984000	110338144
19	2110416504	97412756
20	2229646565	106198871

B.1.3 FALCON-ChoiGAKE

Participants	Mean CPU cycles	Standard deviation
3	340440508	38154217
4	448948478	45203272
5	551329491	46467151
6	663535366	54180245
7	777407858	59374787
8	905872236	76168189
9	996904974	66284594
10	1106181726	64780849
11	1220845950	67632512
12	1342046155	80230937
13	1499395734	92085817
14	1611283586	92493401
15	1728852828	88045003
16	1851523927	100494440
17	1967441882	109124736
18	2128124214	111988875
19	2227091481	103278750
20	2361726592	114336407

B.1.4 qTESLA-AponGAKE

Participants	Mean CPU cycles	Standard deviation
3	68185547	9281054
4	96201622	9671047
5	134274460	17454657
6	175559779	20728496
7	217841687	23300843
8	267584282	23508685
9	321666072	25278488
10	386083166	34678084
11	462504473	53517241
12	532960479	60328421
13	640988508	89707934
14	708705445	87539906
15	803019275	106768070
16	914094954	127233897
17	989413076	124468008
18	1054576419	109909841
19	1126641072	72198178
20	1252961186	79348041

B.1.5 qTESLA-ChoiGAKE

Participants	Mean CPU cycles	Standard deviation
3	65760924	7682345
4	96565902	11981431
5	129041170	13243488
6	171712430	18854481
7	219032410	25922297
8	259772282	16612510
9	322968718	31696826
10	383961352	36025736
11	454894405	57823304
12	531587672	68668141
13	631681231	94867159
14	709673468	85293744
15	798495205	110951967
16	905372576	115785491
17	992825943	125051343
18	1048375458	111726830
19	1117042311	76310603
20	1248092698	82887734

B.2 GAKE without keygen

B.2.1 ECDHGAKE

Participants	Mean CPU cycles	Standard deviation
3	8093472	1498506
4	14135599	682511
5	22836023	1682192
6	33648522	2780582
7	47668281	5837126
8	61881527	6385294
9	79643507	8746228
10	99178334	11269130
11	118717372	11682242
12	139865464	7194238
13	166818467	15236827
14	192118335	16539704
15	222760510	19979406
16	254992546	27853728
17	282003072	13733843
18	318596813	16721442
19	352190055	18359774
20	396270971	27978079

B.2.2 FALCON-AponGAKE

Participants	Mean CPU cycles	Standard deviation
3	143339171	8216654
4	186838744	12359066
5	231786161	11145337
6	276386079	11031563
7	322390574	10298195
8	370556526	13089670
9	419025731	20827660
10	473732265	16316756
11	523260826	13858485
12	579629755	36417906
13	614624116	32844640
14	688762066	21609540
15	752041563	24199271
16	798576606	24815887
17	857755031	28492796
18	917936912	32493345
19	979328270	28528355
20	1046790581	41129200

B.2.3 FALCON-ChoiGAKE

Participants	Mean CPU cycles	Standard deviation
3	142578068	8190200
4	184496132	10771402
5	231157464	12976642
6	273678840	9713205
7	318899892	9692351
8	366670609	10866243
9	413872715	20615615
10	468633594	16350840
11	517936198	13898550
12	570869396	25442262
13	606534978	10235305
14	677151088	19681977
15	741578873	27143423
16	788947326	27646676
17	843036382	25299825
18	903686135	31903366
19	962253517	25995367
20	1027236901	32673723

B.2.4 qTESLA-AponGAKE

Participants	Mean CPU cycles	Standard deviation
3	55736190	7820992
4	80937887	8862372
5	111504090	10239622
6	148308084	13957007
7	190868060	20293890
8	234668863	20895762
9	283235415	20866283
10	342288624	30099683
11	395903144	26307475
12	456609537	23465528
13	535199756	37141832
14	619625059	54850921
15	701591686	49518292
16	806959650	85742196
17	933077455	117716834
18	1025270308	120653521
19	1161871286	155335552
20	1237835766	140255629

B.2.5 qTESLA-ChoiGAKE

Participants	Mean CPU cycles	Standard deviation
3	56598536	8859394
4	84147529	14159968
5	109631300	10402715
6	144696175	11773214
7	189031556	20586055
8	230750438	21945803
9	278766475	19511671
10	339543441	31027041
11	396170015	28356629
12	451681094	20805308
13	528274397	37372856
14	613166107	47539721
15	694753471	53156239
16	801473909	86208020
17	918257328	111059152
18	1017920580	122781363
19	1143992930	140246135
20	1240085579	150435292

Simen Been Kristiansen

NTNU
Norwegian University of
Science and Technology